Modern Programming Languages
Lecture # 40

Names

- Design issues:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Special Words

- There are two types of special words
  - Keyword
  - Reserved word
- A keyword is a word that is special only in a certain context
  - REAL X
  - REAL = 44.7
- Disadvantage: poor readability

- A reserved word is a special word that cannot be used as a user-defined name
  - Type

- **Type:** Determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
  - Value
- **Value:** The contents of the location with which the variable is associated
  - Binding

- **Binding :** A binding is an association, such as between an attribute and an entity, or between an operation and a symbol

  - Binding Time
    It is the time at which a binding takes place

Possible binding times

1. Language design time - e.g. bind operator symbols to operations
2. Language implementation time - e.g. bind fl. pt. type to a representation
   1. Compile time - e.g. bind a variable to a type in C or Java
   2. Load time- e.g.
   3. Runtime - e.g. bind a non-static local variable to a memory cell

Static and Dynamic Binding

- A binding is static if it occurs before run time and remains unchanged throughout program execution

- A binding is dynamic if it occurs during execution or can change during execution of the program

Type Bindings

- How is a type specified?
- When does the binding take place?
- If static, type may be specified by either an explicit or an implicit declaration
  - An explicit declaration is a program statement used for declaring the types of variables
  - An implicit declaration is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability

Dynamic Type Binding

- Specified through an assignment statement e.g. in SNOBOL
  - LIST = 'my name'
  - LIST = 4 + 5
- Advantage
  - Flexibility (generic program units)
- Disadvantages:
  - Type error detection by the compiler is difficult
  - High cost (dynamic type checking and interpretation)

Storage Bindings

  - At what time we associate the memory location with a name

Categories of variables by lifetimes

1. **Static Storage Binding**
   - Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
   - Example
     - In FORTRAN 77 all variables are statically bound
     - C static variables
   - Advantage
     - Efficiency (direct addressing), history-sensitive, subprogram support

- Disadvantage
    - Lack of flexibility (no recursion)

Stack Dynamic Variables:

- Bound to storages when execution reaches the code to which the declaration is attached. (But, data types are statically bound.) That is, stack-dynamic variables are allocated from the **run-time stack**.
- Example
    - E.g. a Pascal procedure consists of a declaration section and a code section.
    - E.g. FORTRAN 77 and FORTRAN 90 use SAVE list for stack-dynamic list.
    - E.g. C and C++ assume local variables are static-dynamic.
- Advantage
    - allows recursion; conserves storage
- Disadvantages
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)

Explicit Heap Dynamic Variables

- Allocated and de-allocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references
- Examples
    - Dynamic objects in C++ (via new and delete)
    - All objects in Java
- Advantage
    - Provide dynamic storage management
- Disadvantage
    - Inefficient and unreliable

Implicit Heap Dynamic Variables

- Allocation and de-allocation is caused by assignment statements
- Example
    - All variables in SNOBOL
- Advantage
    - Flexibility
- Disadvantages
    - Inefficient, because all attributes are dynamic
    - Loss of error detection

Modern Programming Languages Lecture 41

Type Checking

- Generalizes the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler generated code, to a legal type. This automatic conversion is called **coercion**
- A type error is the application of an operator to an operand of an inappropriate type
- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is strongly typed if type errors are always detected
- A programming language which does all the type checking statically then it is a strongly typed languages otherwise weakly typed language.

Strongly Typed Languages?

- FORTRAN 77 is not strongly typed
  - Parameters, EQUIVALENCE
- C and C++ are not
  - Parameter type checking can be avoided
  - Unions are not type checked
- Pascal is not
  - Variant records
- Modula-2 is not
  - Variant records
- Ada is, almost
  - UNCHECKED CONVERSION is a loophole
  - Coercion rules strongly affect strong typing; they can weaken it considerably (C++ versus Ada)
- Advantage of strong typing
  - Allows the detection of the misuse of variables that result in type errors

Type Compatibility

- Type compatibility by name means that the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive
  - Sub ranges of integer types are not compatible with integer types

Data Types

- Design Issues
  - What is the syntax of references to variables?
  - What operations are defined and how are they specified?

Primitive Data Types

- Not defined in terms of other data types

1. Integer

- Almost always an exact reflection of the hardware, so the mapping is trivial
- There may be as many as eight different integer types in a language

2. Floating Point

- Models real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada)
  - type SPEED is digits 7 range 0.0..1000.0;
  - type VOLTAGE is delta 0.1 range -12.0..24.0;

3. Decimal

- For business applications (money)
- Store a fixed number of decimal digits
- Advantage
  - Accuracy
- Disadvantages
  - Limited range, wastes memory
- COBOL and ADA supports decimal
- C++ and java does not support decimal

4. Boolean

- Could be implemented as bits, but often as bytes
- Advantage
  - Readability

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects static or dynamic?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Concatenation
  - Substring reference
  - Pattern matching

- Examples
  - Pascal---Not primitive; assignment and comparisons supported only
  - Ada, FORTRAN 77, FORTRAN 90 and BASIC
    - Somewhat primitive
    - Assignment, comparison, catenation
    - Substring reference
  - C and C++
    - Not primitive
    - Use char arrays and a library of functions that provide these operations
  - SNOBOL4 (a string manipulation language)
    - Primitive
    - Many operations, including elaborate pattern matching
  - Java; string class (not arrays of char)

Ordinal Types (user defined)

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Enumeration Types - one in which the user enumerates all of the possible values, which are symbolic constants
- Design Issue
  - Should a symbolic constant be allowed to be in more than one type definition?
- Examples
  - Pascal
    - Cannot reuse constants
    - They can be used for array subscripts e.g. for variables, case selectors
    - NO input or output
    - Can be compared
  - Ada
    - Constants can be reused (overloaded literals)
    - Disambiguates with context or type_name
    - CAN be input and output
  - C and C++
    - Like Pascal, except they can be input and output as integers
  - Java does not include an enumeration types
  - C# includes them

- Evaluation (of enumeration types)
  - Useful for readability
    - e.g. no need to code a color as a number
  - Useful for reliability
    - e.g. compiler can check operations and ranges of values

- Subrange Type
  - An ordered contiguous subsequence of an ordinal type
  - Pascal
    - Subrange types behave as their parent types; can be used as for variables and array indices
    - E.g. type pos = 0 .. MAXINT;
  - Ada
    - Subtypes are not new types, they are just constrained existing types (so they are compatible)
    - Can be used as in Pascal, plus case constants
    - **E.g.** subtype POS_TYPE is
    - **E.g.** INTEGER range 0 ..INTEGER'LAST;
  - Evaluation (of enumeration types)
    - Aid to readability
    - Reliability - restricted ranges add error detection

- Implementation of user-defined ordinal types
  - Enumeration types are implemented as integers
  - Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

5. Arrays

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element
- Design Issues
  - What types are legal for subscripts?
  - Are subscripting expressions in element references range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?

6. Subscript Types

- FORTRAN, C, C++
  - int only
- Pascal
  - Any ordinal type (int, boolean, char, enum)
- Ada
  - int or enum (includes boolean and char)
- Java
  - integer types only

Four Categories of Arrays (based on subscript binding and binding to storage)

- **Static -** range of subscripts and storage bindings are static e.g. FORTRAN 77, some arrays in Ada
  - Advantage: execution efficiency (no allocation or de-allocation)
- **Fixed stack dynamic -** range of subscripts is statically bound, but storage is bound at elaboration time e.g. C local arrays are not static
  - Advantage: space efficiency
- **Stack-dynamic -** range and storage are dynamic, but fixed from then on for the variable's lifetime e.g. Ada declare blocks declare

  - STUFF : array (1..N) of FLOAT;
    begin
    ...
    end;
  - **Advantage:** flexibility - size need not be known until the array is about to be used

Modern Programming Languages Lecture 42

Records-(like structs in C/C++)

- **Description:**
  - Are composite data types like arrays
  - The difference between array and record is that array elements are of same data type where as record is a logical grouping of heterogeneous data elements.
  - Another difference is that access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
  - Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

- **Design Issues:**
  - What is the form of references?
  - What unit operations are defined?

- **Record Definition Syntax**
  - COBOL uses level numbers to show nested records; others use recursive definitions

  - Examples:
    - COBOL
      field_name OF record_name_1 OF ... OF
      record_name_n

    - Others (dot notation)
      record_name_1.record_name_2. ...
      .record_name_n.field_name

  - Better Approach:
    - According to readability point of view COBOL record definition syntax is easier to read and understand but according to writability point of view other languages record definition syntax is easier to write and less time consuming.

- **Record Field References**

  - Two types of record field references:
    - Fully qualified references must include all record names

    - Elliptical references allow leaving out record names as long as the reference is unambiguous

  - Pascal and Modula-2 provide a with clause to abbreviate references

- **Record Operations**

  - Assignment
    - Pascal, Ada, and C allow it if the types are identical
    - In Ada, the RHS can be an aggregate constant

  - Initialization
    - - Allowed in Ada, using an aggregate constant

  - Comparison
    - - In Ada, = and /=; one operand can be an aggregate constant

Pointers

- **Description:**
  - A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null)

- **Uses:**
  - Addressing flexibility
  - Dynamic storage management

- **Fundamental Pointer Operations:**
  - Assignment of an address to a pointer
  - References (explicit versus implicit dereferencing)

- **Design Issues**
  - What is the scope and lifetime of pointer variables?
  - What is the lifetime of heap-dynamic variables?
  - Are pointers restricted to pointing at a particular type?
  - Are pointers used for dynamic storage management, indirect addressing, or both?
  - Should a language support pointer types, reference types, or both?
  - Pointer arithmetic?

- **Problems with Pointers**

  - Dangling pointers
    - A pointer points to a heap-dynamic variable that has been deallocated

  - Lost Heap-Dynamic Variables
    - A heap-dynamic variable that is no longer referenced by any program pointer
    - The process of losing heap-dynamic variables is called memory leakage

- Pointers are like goto's - they widen the range of cells that can be accessed by a variable
- Pointers are necessary - so we can't design a language without them

- **Examples:**
  - **Pascal:** used for dynamic storage management only
    - Explicit dereferencing
    - Dangling pointers are possible (dispose)
    - Dangling objects are also possible

  - **Ada:** a little better than Pascal and Modula-2
    - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's scope
    - All pointers are initialized to null
    - Similar dangling object problem (but rarely happens)

  - **C/C++:** Used for dynamic storage management and addressing
    - Explicit dereferencing and address-of operator
    - Can do address arithmetic in restricted forms
    - Domain type need not be fixed (void *)
    - ```
      float stuff[100];
      float *p;
      p = stuff;
      ```
    - *(p+5) is equivalent to stuff [5] and p [5]
    - *(p+i) is equivalent to stuff[i] and p[i]
    - void * - can point to any type and can be type checked (cannot be dereferenced)

  - **Fortran90:** Can point to heap and non-heap variables
    - Implicit dereferencing
    - Special assignment operator for non dereferenced references

  - **C++ Reference Types**
    - Constant pointers that are implicitly dereferenced
    - Used for parameters
    - Advantages of both pass-by-reference and pass-by-value

  - **Java - Only references**
    - No pointer arithmetic
    - Can only point at objects (which are all on the heap)
    - No explicit deallocator (garbage collection is used)
    - Means there can be no dangling references
    - Dereferencing is always implicit

Unions

- **Description:**

  - A union is a type whose variables are allowed to store different type values at different times during execution
      ```

- **Design Issues for unions:**

  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?

- **Examples:**

  - FORTRAN
    - EQUIVALENCE
  - Algol has "discriminated unions"
    - Use a hidden tag to maintain the current type
    - Tag is implicitly set by assignment
    - References are legal only in conformity clauses
    - This runtime type selection is a safe method of accessing union objects
  - Pascal
    - Both discriminated and non-discriminated unions are used e.g.

```
type intreal = record tag : Boolean of
                 true  : (blint : integer);
                 false : (blreal : real);
               end;
```

    - Problem with Pascal's design is that type checking is ineffective

    - Reasons:
      - User can create inconsistent unions (because the tag can be individually assigned)

```
var blurb : intreal;
      x : real;
blurb.tagg := true;   { it is an integer }
blurb.blint := 47;    { ok }
blurb.tagg := false;  { it is a real }
x := blurb.blreal;    { assigns an integer
                        to a real }
```

      - The tag is optional!

  - Ada
    - Discriminated unions are safer than Pascal & Modula-2
    - Reasons
      - Tag must be present
      - It is impossible for the user to create an inconsistent union (because tag cannot be assigned by itself)
      - All assignments to the union must include the tag value)

  - C and C++ have free unions (no tags)
    - Not part of their records
    - No type checking of references

- Java has neither records nor unions but aggregate types can be created with classes, as in C++

- Unions are potentially unsafe in most languages (except Ada)

Arithmetic Expressions

- **Description:**

  - Arithmetic evaluation was one of the motivations for the development of the first programming languages
  - Arithmetic expressions consist of operators, operands, parentheses, and function calls
  - 15 * (a + b) / log(x)

- **Design issues for arithmetic expressions:**

  - What are the operator precedence rules?

  - What are the operator associativity rules?

  - What is the order of operand evaluation?

  - Are there restrictions on operand evaluation side effects?

  - Does the language allow user-defined operator overloading?

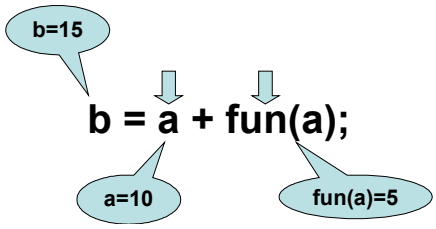
  - What mode mixing is allowed in expressions?

  - Conditional expressions?

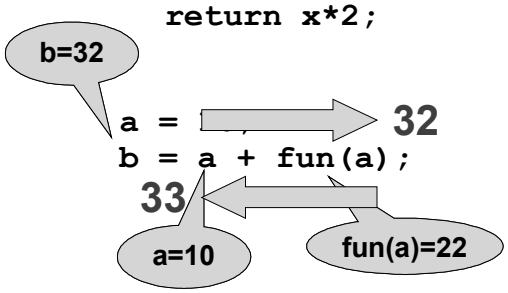
Modern Programming Languages Lecture 43

- **Arithmetic Expressions: Operators**
  - A unary operator has one operand
  - A binary operator has two operands
  - A ternary operator has three operands
- **Arithmetic Expressions: Operator Precedence Rules**
  - The operator precedence rules for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
  - Typical precedence levels
    - parentheses
    - unary operators
    - ** (if the language supports it)
    - *, /
    - +, -
- **Arithmetic Expressions: Potentials for Side Effects**
  - Functional side effects: when a function changes a two-way parameter or a non-local variable
  - Problem with functional side effects:
    - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(a);
```




```
int fun(int &x)
{
    x = x + 1;
    return x*2;
```



- **Functional Side Effects**
  - Two possible solutions to the problem
    - Write the language definition to disallow functional side effects
      - No two-way parameters in functions
      - No non-local references in functions
      - **Advantage:** it works!
      - **Disadvantage:** inflexibility of two-way parameters and non-local references
  - Write the language definition to demand that operand evaluation order be fixed
    - **Disadvantage:** limits some compiler optimizations

- **Operator Overloading**
  - Use of an operator for more than one purpose is called operator overloading
  - Some overloaded operators are common (e.g. '+' for int and float)
  - Some are potential trouble (e.g. '*' in C and C++)
    - * is also used for pointers
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
  - Can be avoided by introduction of new symbols (e.g., Pascal's div for integer division)
  - C++ and Ada allow user-defined overloaded operators
  - Potential problems
    - Users can define nonsense operations
    - Readability may suffer

- **Type Conversion**

  - Implicit Type Conversion
  - Explicit Type Conversion

- **Implicit Type Conversions**

  - A **narrowing conversion** is one that converts an object to a type that cannot include all of the values of the original type e.g., float to int
  - A **widening conversion** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., int to float
  - A **mixed-mode expression** is one that has operands of different types
  - **Coercion** is an implicit type conversion
  - **Disadvantage of coercions:**
    - They decrease in the type error detection ability of the compiler
  - In most languages, all numeric types are coerced in expressions, using widening conversions
  - In Ada, there are virtually no coercions in expressions

- **Explicit Type Conversions**
  - Called casting in C-based language
  - Examples
    - C: (int) angle
    - Ada: Float (sum)
  - Note that Ada's syntax is similar to function calls

- **Errors in Expressions**
  - Causes
    - Coercions of operands in expressions
    - Inherent limitations of arithmetic e.g., division by zero
    - Limitations of computer arithmetic e.g. overflow or underflow
  - Division by zero, overflow, and underflow are run-time errors (sometimes called exceptions)

Relational Expressions

- Use relational operators and operands of various types

- Evaluate to some boolean representation

- Operator symbols used vary somewhat among different languages (!=, /=, .NE., <>, #)

Boolean Expressions

- Operands are Boolean and the result is also a Boolean

- Operators

FORTRAN 77	FORTRAN 90	C	Ada
AND	and	&&	and
OR	or	||	or
NOT	not	!	not
	xor		

Boolean Expressions

One odd characteristic of C's expressions
a < b < c

Short Circuit Evaluation

- A and B

- A or B

- *Example*

```
index := 1;
while (index <= length) and
      (LIST[index] <> value) do
  index := index + 1
```

C, C++, and Java*:* use short-circuit evaluation for the usual Boolean operators (&& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)

Ada: programmer can specify either (short-circuit is specified with 'and then' and 'or else')

FORTRAN 77: short circuiting is there, but any side affected place must be set to undefined

Problem with Short Circuiting

(a > b) || (b++ / 3)

Short-circuit evaluation exposes the potential problem of side effects in expressions

Modern Programming Languages Lecture 44

Control Structure

- Def: A *control structure* is a control statement and the statements whose execution it controls

- *Levels of Control Flow*
  1. Within expressions
  2. Among program units
  3. Among program statements

- *Overall Design Question*

What control statements should a language has, beyond selection and pretest logical loops?

Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the1960s about the issue
- One important result: It was proven that all flowcharts can be coded with only two-way selection and pretest logical loops

Selection Statements
Design Issues

1. What is the form and type of the control expression?

2. What is the selectable segment form (single statement, statement sequence, compound statement)?

3. How should the meaning of nested selectors be specified?

Single-Way Selection Statement

FORTRAN IF
- IF (boolean_expr) statement

Problem
- *C*an select only a single statement;
- To select more, a goto must be used

FORTRAN example:

```
   IF (.NOT. condition) GOTO 20
     ...
     ...
20 CONTINUE
```

ALGOL 60 if:

```
if (boolean_expr) then
  begin
  ...
  end
```

Two-way Selector

***ALGOL 60* if:**

```
if (boolean_expr)
  then statement  (the then clause)
  else statement  (the else clause)
```

- The statements could be single or compound

Nested Selectors

Example (Pascal)

```
if ... then
if ... then
    ...
else ...
```

- Which then gets the else?

- Pascal's rule: else goes with the nearest then

Nested Selectors

ALGOL 60's solution - disallow direct nesting

```
if ... then          if ... then
  begin                begin
  if ...               if ... then ...
    then ...             end
    else ...           else ...
  end
```

FORTRAN 77, Ada, Modula-2 solution – closing special words

Example (Ada)

```
if ... then          if ... then
  if ... then          if ... then
    ...                  ...
  else                   end if
    ...              else
```

```
  end if              ...
end if              end if
```

- *Advantage:* flexibility and readability

- Modula-2 uses the same closing special word for all control structures (END)

- This results in poor readability

Multiple Selection Constructs

Design Issues

1. What is the form and type of the control expression?

2. What segments are selectable (single, compound, sequential)?

3. Is the entire construct encapsulated?

4. Is execution flow through the structure restricted
   to include just a single selectable segment?

5. What is done about un-represented expression
   values?

Early Multiple Selectors

1. FORTRAN arithmetic IF (a three-way selector)
   - IF (arithmetic expression) N1, N2, N3

   Bad aspects:
   - Not encapsulated
     (selectable segments could be anywhere)
   - Segments require GOTO's

Modern Multiple Selectors

1. Pascal case (from Hoare's contribution to ALGOL W)

```
case expression of
  constant_list_1 : statement_1;
  ...
  constant_list_n : statement_n
end
```

Design Choices

1. Expression is any ordinal type
   (int, boolean, char, enum)
2. Only one segment can be executed per

execution of the construct
3. In Wirth's Pascal, result of an un-represented control expression value is undefined
(In 1984 ISO Standard, it is a runtime error)

- Many dialects now have otherwise or else clause

The C and C++ switch

```
switch (expression)  {
   constant_expression_1 : statement_1;
   ...
   constant_expression_n : statement_n;
   [default: statement_n+1]
}
```

Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)

- Trade-off between reliability and flexibility (convenience)

- To avoid it, the programmer must supply a break statement for each segment

Ada's case is similar to Pascal's case, except:

1. Constant lists can include:
   - Subranges e.g., 10..15
   - Boolean OR operators
     e.g. 1..5 | 7 | 15..20

2. Lists of constants must be exhaustive
   - Often accomplished with others clause
   - This makes it more reliable

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses (ALGOL 68, FORTRAN 77, Modula-2, Ada)

Example (Ada)

```
if ...
  then ...
elsif ...
  then ...
elsif ...
  then ...
  else ...
```

end if

- Far more readable than deeply nested if's
- Allows a boolean gate on every selectable group

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion; here we look at iteration, because recursion is a unit-level control

- *General design Issues for iteration control statements are:*
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Counter-Controlled Loops

Design Issues

1. What is the type and scope of the loop variable?

2. What is the value of the loop variable at loop termination?

3. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?

4. Should the loop parameters be evaluated only once, or once for every iteration?

1. *FORTRAN 77 and 90*

- Syntax: DO label var = start, finish [, stepsize]

  - Stepsize can be any value but zero
  - Parameters can be expressions

- Design choices:
  1. Loop variables can be INTEGER, REAL, or DOUBLE
  2. Loop variable always has its last value
  3. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
  4. Loop parameters are evaluated only once

FORTRAN 90's 'Other DO'

- Syntax:
  [name:] DO variable = initial, terminal [, stepsize]
  …
  END DO [name]

- Loop variable must be an INTEGER

2. *ALGOL 60*

- Syntax: for var := <list_of_stuff> do statement
  where <list_of_stuff> can have:
  - List of expressions
  - expression step expression until expression
  - expression while boolean_expression

```
for index := 1 step 2 until 50,
        60, 70, 80,
        index + 1 until 100 do
```

(index = 1, 3, 5, 7, ..., 49, 60, 70, 80,
81, 82, ..., 100)

ALGOL 60 Design choices

1. Control expression can be int or real; its scope is whatever it is declared to be

2. Control variable has its last assigned value after loop termination

3. Parameters are evaluated with every iteration, making it very complex and difficult to read

4. The loop variable cannot be changed in the loop, but the parameters can, and when they are, it affects loop control

3. Pascal

- Syntax:
  for variable := initial (to | downto) final do
  statement

- Design Choices:
  1. Loop variable must be an ordinal type of usual scope
  2. After normal termination, loop variable is undefined
  3. The loop variable cannot be changed in the loop; the loop parameters can be changed, but they are evaluated just once, so it does not affect loop

control

4. *Ada*

- Syntax:

```
for var in [reverse] discrete_range loop
  ...
end loop
```

Ada Design choices

1. Type of the loop var is that of the discrete range; its scope is the loop body (it is implicitly declared)
2. The loop var does not exist outside the loop
3. The loop var cannot be changed in the loop, but the discrete range can; it does not affect loop control
4. The discrete range is evaluated just once

5. C

- Syntax:

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
- The value of a multiple-statement expression is the value of the last statement in the expression
  e.g.

```
for (i = 0, j = 10; j == i;  i++) ...
```

- If the second expression is absent, it is an infinite loop

-C Design Choices

1. There is no explicit loop variable
2. Irrelevant
3. Everything can be changed in the loop
4. Pretest
5. The first expression is evaluated once, but the other two are evaluated with each iteration

- This loop statement is the most flexible

6. C++

- Differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the function in which it is defined)

7. Java

- Differs from C++ in two ways:
  1. Control expression must be Boolean
  2. Scope of variables defined in the initial expression is only the loop body

Logically-Controlled Loops

- Design Issues
  1. Pre-test or post-test?
  2. Should this be a special case of the counting loop statement (or a separate statement)?

Examples

1. Pascal has separate pretest and post-test logical loop statements (while-do and repeat-until)

2. C and C++ also have both, but the control expression for the post-test version is treated just like in the pretest case (while - do and do - while)

3. Java is like C, except the control expression must be Boolean (and the body can only be entered at the beginning- Java has no goto)

4. Ada has a pretest version, but no post-test

5. FORTRAN 77 and 90 have neither

User-Located Loop Control Mechanisms

- Design issues

1. Should the conditional be part of the exit?
2. Should the mechanism be allowed in an already controlled loop?
3. Should control be transferable out of more than one loop?

1. C , C++, and Java - break

- Unconditional; for any loop or switch; one level only (Java's can have a label)
- There is also a continue statement for loops; it skips the remainder of this iteration, but does not exit the loop

2. FORTRAN 90 - EXIT

- Unconditional; for any loop, any number of levels
- FORTRAN 90 also has a CYCLE, which has the same semantics as C's continue

Examples:

1. Ada - conditional or unconditional exit; for any loop and any number of levels

```
for ... loop
  ...
  exit when ...
  ...
end loop
```

Example (Ada)

```
LOOP1:
  while ... loop
            ...
    LOOP2:
          for ... loop
    ...
    exit LOOP1 when ..
    ...
end loop LOOP2;
  ...
  end loop LOOP1;
```

Unconditional Branching

- All classical languages have it

- Problem: readability
  Edsger W. Dijkstra,
-Go To Statement Considered Harmful,
  CACM, Vol. 11, No. 3, March 1968, pp. 147-148
- Some languages do not have them e.g. Modula-2 and Java
- Advocates: Flexibility
  KNUTH D E, Structured programming with go to statements, ACM Computing Surveys 6, 4 (1974)

Conclusion

Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability!

Connection between control statements and

program verification is intimate

- Verification is impossible with goto's
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

Modern Programming Languages Lecture 45

Actual/Formal Parameter Correspondence:

1. Positional
2. Keyword

e.g. SORT(LIST => A, LENGTH => N);

Default Parameter Values

```
procedure SORT(LIST : LIST_TYPE;
               LENGTH : INTEGER := 100);
...
SORT(LIST => A);
```

Parameters and Parameter Passing

Semantic Models

- in mode, out mode, in-out mode

Conceptual Models of Transfer

1. Physically move a value

2. Move an access path

Implementation Models

1. Pass-by-value (in mode)

- Either by physical move or access path

2. Pass-by-result (out mode)

- Local's value is passed back to the caller

- Physical move is usually used

- Disadvantages: Collision

Example:

```
procedure sub1(y: int, z: int);
            ...

sub1(x, x);
```

3. Pass-by-reference (in-out mode)

i. Actual parameter collisions

e.g.

```
procedure sub1(a: int, b: int);
```

```
...
sub1(x, x);
```

ii. Array element collisions

e.g.

```
sub1(a[i], a[j]);  /* if i = j  */
```

iii. Collision between formals and globals

Root cause of all of these is: The called subprogram is provided wider access to Non-locals than is necessary

Type checking parameters

- Now considered very important for reliability

FORTRAN 77 and original C: none

Pascal, Modula-2, FORTRAN 90, Java, and Ada: it is always required

- ANSI C and C++: choice is made by the user

Implementing Parameter Passing

ALGOL 60 and most of its descendants use the run-time stack

- Value copied to the stack; references are indirect to the stack

- Result : same

- Reference : regardless of form, put the address on the stack

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way

- These two are in conflict with one another!

Good programming =>
limited access to variables, which means one-way whenever possible

Efficiency =>
pass by reference is fastest way to pass structures of significant size

Concluding Remarks

- Different programming languages for different problems
  Imperative, Declarative, Functional, Object-Oriented, Scripting
- Readability – maintainability

- Side effects
- Mainly because of the assignment statement and aliasing
- Reduces readability and causes errors
- Operand evaluation order
- Data types and data sizes
- Static-Dynamic activation records
- Necessary for recursion