

## COMP205 IMPERATIVE LANGUAGES

### 9. EXPRESSIONS AND STATEMENTS

1. Unions
2. Expressions.
3. Operators.
4. Type equivalence.
5. Coercion, casting and conversion.

## UNIONS

- It is sometimes desirable to define a variable which can be of two or more different types according to different circumstances (overloading).
- In imperative languages such a data item is termed a *union*.
- The distinction between a union and a structure is that:
  - The members of a structure define different variables while,
  - The members of a union define different manifestations of the same variable.
- Thus space need only be allocated to accommodate the largest type specified in a union.

## C UNION EXAMPLE

```
#include <stdio.h>

void main(void) {
    typedef union data {
        int integer;
        char character;
    } DATA_T;
    DATA_T input;

    input.integer = 2;
    printf("input.integer = %d\n", input.integer);

    input.character = 'A';
    printf("input.character = %c\n", input.character);
}
```

## SOME FURTHER NOTES ON UNIONS

- In languages where it is left to the programmer to remember which type is to be assumed on each occasion that a union is used, we say that the union is *undiscriminated* (e.g. C).
- In languages where the compiler is able to automatically identify the type of a union on each occasion that it is used we say that the union is *discriminated*.
- Unions can be used to simulated the *variant record* concept found in Modula-2 etc.
- Unions are not specifically supported by Ada (although the same effect can be contrived using records and only using some of the fields as appropriate)

## EXPRESSIONS

- 1) Overview (operators and operands)
- 2) Boolean expression
- 3) Arity of operators
- 4) The C ternary operator
- 5) Processing of expressions
- 6) Precedence Rules
- 7) Associativity and (lazy evaluation)

## EXPRESSIONS

- Expressions are composed of one or more *operands* whose values are combined by *operators*.
- Typical operators include the maths operators +, -, \*, /.  
Example: `NUM_I + (2 * NUM_J)`
- Evaluation of an expression produces a value (an anonymous data item).
- Expressions can therefore be used to assign values to variables.

## BOOLEAN EXPRESSIONS

- **Boolean expressions** are expressions that use **Boolean** operators.
- Boolean expression return the values `true` or `false`.
- The first 6 of the above are sometimes referred to as **comparison** (or **relational**) operators and the remaining 3 as **logical** operators.

|               | C  | Ada | Pascal |
|---------------|----|-----|--------|
| Equals        | == | =   | =      |
| Not equals    | != | /=  | <>     |
| Less than     | <  | <   | <      |
| Greater than  | >  | >   | >      |
| < or equal to | <= | <=  | <=     |
| > or equal to | >= | >=  | >=     |
| Logical or    |    | or  | or     |
| Logical and   | && | and | and    |
| Logical not   | !  | not | not    |

## ARITY OF OPERATORS

- The number of **operands** that an operator requires is referred to as its **arity**.
- Operators with one operand are called **monadic** or **unary**.
- Operators with two operands are called **dyadic** or **binary**.
- Some programming languages (C, but not Ada or Pascal) support **ternary** operators (three operands):

```
<condition> ? <expression1> :
<expression2>
```

returns `expression1` if the condition is true and `expression2` otherwise.

- Operators can also be classified as **prefix**, **infix** or **postfix**.

## THE C TERNARY OPERATOR

```
main () {
    int number;

    scanf("%d",&number);

    printf("The number %d is an ",number);
    (number/2)*2 == number ? printf("even") :
                             printf("odd");
    printf(" number.\n");
}
```

```
int division(int num1, int num2) {
    return(num2 != 0 ? num1/num2 : 0);
}
```

## PROCESSING OF EXPRESSIONS

- Evaluation of an expression produces a value (an **anonymous data item**).
- Any sub-expression enclosed within brackets should be evaluated first (starting with the innermost brackets).
- Where more than one operator is contained in an expression (bracketed or otherwise) process the operators according to a sequence defined by:

- 1) Their **precedence**, and/or
- 2) Their **associativity**.

## PRECEDENCE RULES

- **Precedence rules** indicate the order in which a sequence of operators in an expression should be evaluated.
- Operators have a **precedence level** associated with them.
- Given two operators with different precedence levels the **lower-level** operator will be processed prior to **higher-level** operator.
- We say that low-level operators **bind** their operands more strongly than high-level operators.
- Operators on the same level have the same precedence.

## PRECEDENCE TABLE

| Level | Operator                    |
|-------|-----------------------------|
| 2     | +, - (unary plus and minus) |
| 3     | ** (exponential)            |
| 4     | *, /, rem                   |
| 5     | +, -                        |
| 6     | Comparison operators        |
| 7     | Logical not                 |
| 8     | Other logical operators     |
| 9     | := (assignment)             |

```
#include<stdio.h>
```

```
void main(void) {  
int numA = 2;  
int numB = 4;  
  
printf("%d\n", numA+numB*numB);  
printf("%d\n", numB*numB+numA);  
printf("%d\n", (numA+numB)*numB);  
printf("%d\n", numA*2==numB);  
printf("%d\n", numA+2==numB*2);  
}
```

```
18  
18  
24  
1  
0
```

- Note that C Boolean expression returns a value (usually 1) to represent true, and 0 to represent false.
- Remember there is no type Boolean in C

## ASSOCIATIVITY

- Operators on the same level have the same *precedence*.
- The order of execution of operators on the same level is controlled by *associativity*:
  - *Left-associative* operators are evaluated from left to right.
  - *Right-associative* operators are evaluated from right to left.
- If an operator is *non-associative* it cannot be used in a sequence.
- Some language include operators that have no specific associativity (that is not to say that they are non-associative --- they have to be processed in some order).

## THE AND OPERATOR AND LAZY EVALUATION

- Consider the following C expression:

```
(q != 4) && (x == q+p*4) && (x >= 0)
```

- This sequence will be evaluated from left to right (&& is left associative) until the end of the sequence is reached or one of the operands for the && operators fails.
- This process of not needlessly evaluating each operand is termed lazy evaluation.

```
#include<stdio.h>
```

```
int lazyEval(int, int, int);
```

```
void main(void) {  
printf("%d\n", lazyEval(2,1,9));  
printf("%d\n", lazyEval(-2,1,-7));  
printf("%d\n", lazyEval(2,1,8));  
printf("%d\n", lazyEval(4,1,17));  
}
```

```
int lazyEval(int numP, int numQ, int numX)  
{  
return((numP != 4) && (numX == numQ+numP*4)  
      && (numX >= 0));  
}
```

```
1  
0  
0  
0
```

## LAZY EVALUATION Cont.

- This form of lazy evaluation is not supported Ada.
- In the case of the Ada and operator no order of evaluation (association) is specified, i.e. all operands for the and operator will be evaluated regardless of whether this is necessary or not.
- The effect of lazy evaluation, as illustrated above, can be achieved in Ada by writing:

```
(Q /= 4) and (X = Q+P*4) then (X >= 0)
```

- Lazy evaluation is an important concept in functional languages.

## TYPE EQUIVALENCE

- 1) Coercion
- 2) Casting
- 3) Conversion

## COERCION

- Operators require their operands to be of a certain type (similarly expressions require their arguments to be of the same type).
- In some cases it may be appropriate, when a compiler finds an operand that is not quite of the “right” type to convert the operand so that it is of the right type.
- This is called *coercion*.
- C supports coercion, Ada does not.

## CASTS

- It is sometimes necessary for a programmer to force a coercion.
- This is referred to as a *cast*.
- This is supported in C (but not Ada).
- Example:

```
void main(void) {  
    float j = 2.0;  
    int i;  
  
    i = (int) (j * 0.5);  
}
```

## CONVERSION

- Where casting is not supported the most common alternative is *explicit conversion*. This is supported by Ada.
- Ada conversions have the following form:

$$T(N)$$

where T is the name of a numeric type and N has another numeric type. The result is of type T.

- The distinction between conversion and casting is that the latter relies on the compilers coercion mechanism to achieve the end result while a conversion obtains the desired result explicitly through a call to a conversion function.

## SUMMARY

1. Expressions.
2. Operators.
3. Statements.
4. Type equivalence.
5. Coercion, casting and conversion.