Binding and Variables

1. Definitions

- **Attributes**:
    - It is a set of properties used to describe a program entity, e.g., variable and function.
    - Example:
        - Array: Name, Element type, Index type, Index lower bound, Index upper bound, and address.
        - Variable: Name, Type, and Value.

- **Descriptors**:
        - It is where the values of the attributes of an element are stored.

- **Binding**:
    - It is the process of assigning a value to the attribute of an element.
    - **Binding Time**:
        - At what time a value is assigned an attribute.
        - There are two types of binding:
            - **Static**:
                - A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
                - Static binding occurs at:
                    - Compile time:
                    - Implementation time:
                        - Range of values for an integer.
                    - Language definition:

            - Possible operators on strings: || and +.
        - Dynamic:
            - A binding is *dynamic* if it first occurs during execution or can change during execution of the program.
            - Dynamic binding occurs at:
                - Run time:
                    - Assign a value to a variable.

- **Stability**:
    - Is the assignment of a value fixed or modifiable?

2. Variables

- Most conventional programming languages can be viewed as ***abstraction*** of an underlying Von Neumann architecture:
    - Memory **cell** with and **address** and a **value**.

- Four Semantic Attributes of Variables
  In general, the semantics of variables in programming languages is often described in terms of four attributes.
    - **Name**: It is the name used to refer to the variable.
    - **Type**: A description of the set of permissible values for a variable.
    - **Scope**: The region of program text over which a variable is known.
    - **l-value (or location):** A location in memory associated with the variable.

- **r-value**: Typically, an indirect attribute of a variable; the value stored in the memory location associated with a variable.

3. Type

- Definitions:
    - **Type checking** is the activity of ensuring that the operands of an operator are of compatible types
    - **Casting or Coercion**: 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.
    - A type error is the application of an operator to an operand of an inappropriate type.
    - Static type checking: If all type bindings are static.
    - Dynamic type checking: If all type bindings are dynamic.

- Strongly Typed Languages:
    - A Language is strongly typed if type errors are always detected.Some languages are ***strongly type:*** Ada, C, C++, Java
    - Others are not: LISP

- Most allow you to create complex:
    - ADTs, Records, Structures, Classes

4. Scope

- **Definitions**:

- The *scope* of a variable is the range of statements over which it is visible
- The **scope** rules of a **language** determine how references to names are associated with variables.

- **Example**:

```
#include <stdio.h>
main(  )  {
  int x, y;
  printf("Please enter a value for x: ");
  scanf( "%d", &x );
  printf("Please enter a value for y: ");
  scanf( "%d", &y );
  { // This block used to swap x and y
    int temp;
    temp = x;
    x = y;
    y = temp;
  }
  printf( "x:%d and y:%d\n", x, y );
  // printf( "temp:%d\n", temp );
}
```

If you try to execute the last statement in Visual C++, you will get the following error:

"error C2065: 'temp' : undeclared identifier"

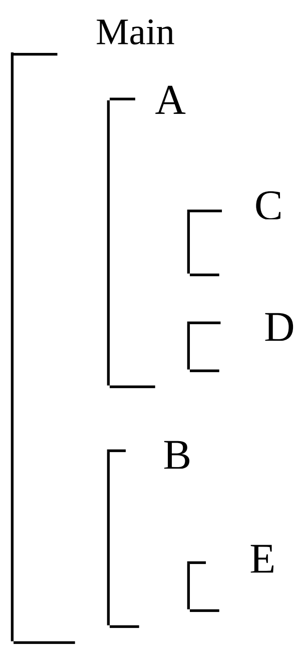
- **Non-local variables**:
  - The non-local variables of a program unit are those that are visible but not declared in the program.
  - Example:

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

```
float increase_factor = 10.5;

main()  {
  float x;
  printf("Please enter a value for x: ");
  scanf( "%f", &x );
  printf( "The input value is: %f\n",x);
  // increase_factor variable is used but not declared in
  // main function.
  printf( "The increased value is: %f\n",x *
increase_factor);
}
```

- Searching:
    - Search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name



- Scoping types:
    - Static scope binding: The scope of a variable is defined by examining the program code. You do not need to run the program.

- Dynamic scope binding:
  - The scope of a variable is defined at run-time.
  - Example:

```
MAIN {
        - declaration of x
        SUB1 {
                - declaration of x -
                ...
                call SUB2
                ...
        }
        SUB2 {
                ...
                - reference to x -
                ...
        }
        ...
        call SUB1
        …
}
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

- **Static scooping**: reference to x is to MAIN's x
- **Dynamic scooping**: reference to x is to SUB1's x

5. References

- Definition: A pointer or reference: it is when the r-value of a variable is used to access another variable.
- Used for dynamic storage management and addressing
- Example:

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

```
main()  {

        int x = 5;
        int *x_pointer;  // Pointer to an integer.

        printf( "The value of x before is: %d\n", x );
        x_pointer = &x;  // Now x_pointer can access x.
        *x_pointer = 6;  // Indirectly change x.
        printf( "The value of x is after: %d\n", x );
        printf( "The memory cell pointed to by x_pointer: %d\n",
                *x_pointer );
}
```

- A C/C++ example:

```
#include <stdio.h>
#define size  10

main()  {

        int choices[size];
        int *choices_ptr;  // Points to the array.
        int i=0;
        choices_ptr = choices;
        //initialize the array
        for (i=0;i<size;++i)
                // the following is equivalent to choices[i]=i*i;
                *(choices_ptr+i) = i*i;
        // Print the array
        for (i=0;i<size;++i)
                printf("choices[%d]= %d\n", i, choices[i]);
}
```

- **Java:**
    - No pointer arithmetic
    - Can only point at objects

- No explicit deallocator (garbage collection is used)

6. Routines

- We will use the term ***routine*** to mean:
    - Subprograms: Assembler
    - Subroutines: FORTRAN
    - Procedures: Pascal, Ada
    - Functions: C, LISP
    - Methods: Java, C++

- Routine Parts:
    - Declaration:
        - The Specification of the name, list of formal parameters, and any return type.
    - Body
        - The list of statements within the definition of the routine.
    - Invocation:
        - Statement used to call the routine.

- Routine Attributes:
    - Name
    - Scope: It is similar to variable scope.
    - Type: It is defined by the routine header: Name of the routine, the types of the parameters, and the type of the returned type.
    - L-value: It is the memory area where the body of the routine is stored.
    - R-value: It is the body of the routine.

- Routine Parameters:
  - Formal parameters:
    - It is the set of parameters that appear in the routine's definition.
  - Actual parameters:
    - It is the set of parameters that appear in the routine's call.
  - Some programming languages have **positional** method and **named** for binding actual parameters to formal parameters in routine calls.

- Routine **Signature**:
  - This specifies the types of the parameters and the return type.

- Activation record:
  - Data objects associated with **local variables** (including any parameters)
  - The relative position (offset) of the data object in the activation record.
  - Return pointer: It is the address where execution must resume in the calling routine.

7. Aliasing and Overloading

- **Overloading**: Method overloading is commonly used to create several methods with the same name that perform similar tasks:

```
public int square (int side);
public double square(double side)
```

- Java enables methods of the same name to be defined as long as they have different signatures.
- A C++ Example:

```
#include <iostream.h>

int max(int x, int y){
    return (x>y?x:y);
}


float max(float x, float y){
    return (x>y?x:y);
}

void main(){

    float a = 4.5;
    float b = 3.4;
    cout << max(3,6) << endl;
    cout << max(a,b) << endl;
}
```

- Aliasing:
    - Two names are aliases if they refer to the same entity at the same program points.
    - It is related to variables.
    - Example:

```
#include <iostream.h>
void main(){

        int i =4;
        int * p_i = &i;
        int * p_ii= &i;

        cout << "The value of i: " << i << endl
                << "The value of &p_i: " << * p_i << endl
                << "The value of &p_ii: " << * p_ii << endl ;
```

}

8. Generics and Templates

- Generic routines allow the same code to be used for multiple data types:
    - ADT Stack, Sorting, searching, etc.

- Called templates in C++

- Generic types bound to actual types by instantiation at compile time