Lecture 04: Data Types and Variables

All programming languages provide data types. A data type describes a set of data values and a set of predefined operations that are applicable to the data items belonging to the set.

This lecture contains:

- Primitive Data Types
- Variables
- Programmer-defined Data Types

4.1 Primitive Data Types

A *data type* describes a set of data values and a set of predefined operations that are applicable to the data items belonging to the set. A primitive data type is a data type that is not defined in terms of other data types.

Primitive data types are the building blocks for data representation in a programming language. Most programming languages support the following primitive data types: integer, real number, Boolean, character, string.

4.1.1 Integer

The *integer* data type includes the set of positive and negative whole numbers. Implementations range from 4 bytes to 8 bytes. Some languages have variations of the integer type. For example, Java implements the following variations: **byte**, **short**, **int**, and **long**. C++ implements the following variations: **short**, **int**, **long**, and **unsigned**.

Note: Most computers implement negative integers as 2's compliment or 1's compliment.

4.1.2 Real Number

You will recall from your elementary mathematics that the set of real numbers includes all (non-complex) numbers. In several languages (including C-based languages), real numbers are implemented as floating-point numbers and double precision floating-point numbers. Business-oriented languages such as COBOL, C#, and SQL (structured query language) have a **decimal** data type. SQL also has a **number** data type, which is essentially a real number. The language Pascal has a **real** data type.

4.1.3 Boolean

A *Boolean* data type allows for two possible values: *true* or *false*. This data type was first introduced in ALGOL 60, and has since been adopted by most programming languages. Earlier versions of C did not have a Boolean data type, but allowed numeric expressions to be treated as though they were Boolean. This was subsequently revised in C++ to incorporate a truly Boolean data type.

Even though technically speaking, only a bit is required to store Boolean values, for practical reasons, a byte is often used.

4.1.4 Character

The *character* data type relates to the representation of alphanumeric data. Traditionally, the coding systems were EBCDIC and ASCII. There is also the ISO 8859-1 coding system. These are all 8-bi8t systems. In more recent times, we have seen the introduction of the 16-bit Unicode system as implemented in Java, C#, and JavaScript.

4.1.5 Strings

A string is a sequence of characters. Some languages (e.g. Pascal) implement strings as primitive data types. Others (e.g. C++ and Java) implement strings as advanced data types. Some languages implement variable length strings (e.g. Java); others provide the flexibility of implementing fixed or variable length strings (e.g. Pascal and C++).

4.2 Variables

As you are aware, a variable is a name or identifier used to specify a data item. During the translation process, variables are translated to memory locations at execution-time, hence, the use of the symbol table (review lecture 3).

The generic term *identifier* is used to refer to variables, labels, or subprograms. To be clear, we will be explicit about the elements used. How identifiers are treated in a programming language is an issue that must be resolved early in the design of the language.

4.2.1 Variable Naming

In most languages, the convention is to have identifiers beginning with a letter, followed by a string of characters, including the underscore. Hyphens are usually not allowed, since they may be confused with the minus sign (-).

In some languages, external names are restricted to a specific length (e.g. C and RPG-400). A few languages also have length restrictions on internal identifiers (e.g. RPG-400). Languages such as Java, C#, Ada, and Pascal have no limit on identifier names.

In C-based languages, the so-called "camel" notation has replaced the more traditional notation. The "camel" notation begins with an identifier name with a lowercase letter, and capitalizes the first letter subsequent words in a multi-word name (e.g. **theBirthDate**). The more traditional approach is to capitalize the first letter of each word in the identifier (e.g. **TheBirthDate**). Neither convention is enforced by syntax, so observance is left to the discretion of the programmer (and rightly so).

4.2.2 Reserve Words

Every language has *reserve words*. These are special words that can only be used for their intended purpose; they cannot be used as identifiers. Reserve words include primitive data types and qualifiers.

4.2.3 Variable Declaration

Some languages require variable declaration before usage. The declaration alerts the compiler or interpreter of the identifier, and allows for easy loading of the symbol table. Languages such as Pascal and Oracle's PL-SQL represent the strictest form of this strategy.

4.2.3 Variable Declaration (continued)

Some languages do not require variable declaration prior to usage. Two examples of such languages are Basic and FoxPro. For these languages, constructing and maintaining the symbol table is more complicated — the ability to report errors relating to the use of variables is more difficult.

C-based languages provide a compromise, where a variable can be declared at the point where it is first introduced. However, the conventional wisdom is that it is a good habit to declare variables prior to their usage.

The syntax of Pascal-based languages (including Pascal and Oracle PL-SQL) require variable the variable-name prior to the data type of the variable; C-based languages require the data type prior to the variable-name. Figure 4.1 illustrates

Figure 4.1: Illustrating Variable Declaration in Pascal versus C++ and Java

```
{Pascal Variable Declaration follows. Note, this is how a Pascal comment is made:}
Var     MyName: String;

// Java declaration follows. Note, this is how a comment is made is C-based languages:
String MyName;

// C++ declaration follows. Note, this is how a comment is made is C-based languages:
string MyName;
char* MyName; // An alternate way of defining a C++ string
char MyName[n]; // An alternate way of defining a C++ string, assuming that n is previously defined
```

4.2.4 Binding of Attributes to Variables

There are two approaches to binding, as discussed below:

- **Static Binding:** Binding of attributes to variables occurs before runtime.
- **Dynamic Binding:** Binding of attributes to variables occurs at runtime.

Dynamic binding (also called *late binding*) is considered more flexible and efficient than static binding (also called *early binding*).

4.2.5 Strong Typing

A strongly types language is a language in which each variable-name has a single data type associated with it, and this is known at compilation time. Strongly typed languages tend to detect type violations rather early in the translation process. Pascal and Ada are strongly typed; C, C++, and Fortran are not. Java and C#, despite being C-based, are strongly typed; however, they facilitate type conversion through promotion or casting.

4.2.6 Data Conversion

Some languages allow for data type conversion implicitly from lower range to higher range (e.g. integer to real), and explicitly via casting. This is so in C-based languages. Other languages such as Pascal do not allow for type conversion. In the case of Python and Visual Basic, explicit data conversion functions are provided.

4.2.7 Variable Scope

The scope of a variable is the range of statements in which it is visible. Of interest is the scope rules that the language implements. The conventional wisdom is that a variable scope is confined to the programming block in which it was defined. This is referred to as *static scoping*.

Languages such as APL, SNOBOL4, LISP, Common LISP, and Perl support *dynamic scoping*, where the scope of the variable may change at runtime. This is quite problematic, and can significantly affect the performance of the program, if care is not taken.

On the surface, C and C++ appear to support static scoping. However, these languages do allow indirect access of variables outside of their scope via reference parameters and pointers.

4.2.8 Program Blocks

How does a programming language handle blocking? C-based languages use the left and right curly braces ({…}). Ada, Pascal, and PL-SQL use the keywords **begin** and **end**.

4.2.9 Named Constants

A constant is a variable that is bound to one value for the duration of the program execution. Named constants are useful in improving the readability of the program.

Pascal, C, and C++ allow for declaration of constants via the **const** keyword. Java requires the use of the **final** keyword. Figure 4.2 provides some examples.

Figure 4.2: Illustrating Constant Declaration in Pascal, C++, and Java

```
{Pascal constant declaration follows.}
Const  PI = 3.142;

// Java constant declaration follows.
double final PI = 3.142;

// C++ constant declaration follows
const double PI = 3.142;
```

4.3 Programmer-defined Data Types

A *programmer-defined data type* (also called *abstract data type*) is a data type that the programmer defines, using primitive data types and/or other programmer-defined data types as building blocks. Some languages are more flexible than others in this regard. Among these data types are the following: enumerated types, sub-range types, arrays, records, unions, pointers, etc.

4.3.1 Enumerated Types

Some languages allow for definition of *enumerated data types*. An enumeration is a list of allowed values for data items defined on this data type. Enumerated types are suited for situations where a finite list of legal values is required. This is supported in languages such as C++, Java, Pascal, and Ada. Figure 4.3 illustrates.

Figure 4.3: Illustrating Enumerated Types in Pascal, C++, and Java

```
{Pascal enumerated type definition for days of the week follows.}
Type   DaysOfWeek = (SUN, MON, TUE, WED, THU, FRI, SAT);

// Java enumerated type definition for days of the week follows
enum DaysOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT};

// C++ enumerated type definition for days of the week follows
enum DaysOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT};
```

4.3.2 Sub-range Types

A sub-range is a continuous stream of values between two stated limits. Pascal and Ada support this data type. Below is a Pascal example.

```
{Pascal declaration of a sub-range follows}
Type  Month = 1 .. 12;

{Having defined the sub-range, we can declare variables of it}
Var myMonth: Month;
```

4.3.3 Arrays

An array is a finite list of data items belonging to a particular base type. Most languages support arrays, some more efficiently than others.

Most languages require that the dimension(s) of the array be known before its declaration. This is true for Pascal, C, C++, RPG-400, etc. However, C++ allows the programmer to resize the array by allocating more memory for it.

4.3.3 Arrays (continued)

Java allows the programmer to declare the array without specifying its precise dimension(s). When the actual size is known, the array is then created and its size specified. Like C++, the array size can be modified. Java also supports heterogeneous arrays by allowing the programmer to declare and manipulate an array of the **Object** class.

Languages such as SAL (Scalable Application Language) and C99 support variable-length arrays.

Figure 4.4 shows array declarations in Pascal, Java, and C++

Figure 4.4: Illustrating Array Declarations in Pascal, C++, and Java

Figure 4.4a: Pascal BNF Definition of an Array

ArrayDefinition ::=

```
Arrray [<Sub-range>] of <DataType> (* Note, the square bracket is a required part of the syntax *)

(* Example follows *)
Type SaleList = Array[1..12] of Real;
…
Var Sale: SaleList;

(* Alternate definition follows *)
Var Sale: Array[1 .. 12] of Real;

(*Alternate definition follows *)
Type
        Month = 1 .. 12;
        SaleList = Array[Month] of Real;
…
Var Sale: SaleList;
```

Figure 4.4b: Java BNF Definition of an Array

ArrayDefinition ::=

```
<DataType>[ ] <ArrayName>;  // Note, the square bracket is a required part of the syntax

// Example follows
double [ ] Sale;  int n;  // The array is declared
…
Sale = new double[n]; // The array is being created, assuming n has a non-zero positive value
```

Figure 4.4c: C++ BNF Definition of an Array

ArrayDefinition ::=

```
<DataType> <ArrayName> [<Length>];

// Example follows
int n;
…
double [n] Sale;  // Note, in some implementations, n has to be an integer constant

// Alternate example follows
typedef double SaleList[12]; // The array is first defined as a data type
…
SaleList Sale;
```

4.3.4 Records

A *record* is an aggregation of related data items, possibly of different data types. The term used for *record* varies with different languages:

- Pascal, COBOL, and Ada use the term *record*
- RPG-400 implements records as *data structures*
- C and C++ implements records as *structures*
- C++ and Java implements records as *classes*

Figure 4.5 shows record declarations in Pascal, Java, and C++, while figure 4.6 shows how you could define an array of records in each language.

Figure 4.5: Illustrating Record Declarations in Pascal, C++, and Java

Figure 4.5a: Pascal BNF Definition of a Record

```
RecordDefinition ::=
Type <RecordName> =
Record
<VariableDeclaration>;
{<VariableDeclaration>;}  (*Zero or more data items *)
End;

(* Example follows *)
Type StudentRecord =
Record
        ID_Number: Integer;
        LastName:     String;
        FirstName:    String;
        Major:  String;
        GPA:          Real;
End;
…
Var     CurrentStud:   StudentRecord;
```

Figure 4.5: Illustrating Record Declarations in Pascal, C++, and Java (continued)

Figure 4.5b: C++ BNF Definition of a Record

```
StructureDefinition ::=
struct [<StructureName>]
{
<VariableDeclaration>;
*<VariableDeclaration>;  // Zero or more data items
}[*<VariableName>] ;

// Example follows
typedef char* String;
struct StudentRecord
{
        int ID_Number;
        String LastName;
        String FirstName;
        String Major;
        double GPA;
};
// …
 StudentRecord CurrentStud;
```

Figure 4.5c: Java BNF Definition of a Record

```
ClassDefinition ::=
public|private|protected class <ClassName>
{
<VariableDeclaration>;
*<VariableDeclaration>; // Zero or more data items
 public|private|protected class <ClassName> ([<ParameterList>]) // Constructor
{ *<Statement>; }
*<MethodDefinition>  // Zero or more methods
}

// Example follows
public class StudentRecord
{
        int ID_Number;
        String LastName, FirstName, Major;
        double GPA;

        public class StudentRecord ( ) // Constructor
        { … };

        //  Other methods …
};
// …
 StudentRecord CurrentStud;
```

Figure 4.6: Illustrating Array of Records in Pascal, C++, and Java

Figure 4.6a: Pascal Array of Student Records

```
{Assuming the declarations of figure 4.5a}
Type StudentList = Array [1 .. 20] of StudentRecord;
…
Var ThisList: StudentList;
```

Figure 4.6b: C++ Array of Student Records

```
//Assuming the declarations of figure 4.5b
typedef StudentRecord StudentList[20];
…
StudentList ThisList;
```

Figure 4.6c: Java Array of Student Records

```
//Assuming the declarations of figure 4.5c
StudentRecord [ ] ThisList;
…
ThisList = new StudentRecord[20];
```

4.3.5 Unions

A *union* is a record-like structure that stores alternate definitions for certain data items. An item from the union can be referenced from the body of the program by an alternate name. Both Ada and C++ support unions.

4.3.6 Pointers

A *pointer* is a data type that allows variables defined on it to be references to memory locations. Pointers are useful in implementing dynamic lists, and are for more efficient than static arrays.

Languages such as Pascal, C, and C++ support pointers by requiring explicit pointer declarations. Java implicitly supports pointers by simply implementing all objects as reference variables (that reference memory locations). Java provides additional flexibility by allowing the programmer to define and manipulate a list (static or dynamic) of instances of the **Object** class, thus facilitating a potentially heterogeneous list.

By way of illustration, suppose that we desire to construct a dynamic list of student records. The best way to do this in Pascal is via a linked list of student records. The best way to do it in Java is via a linked list, vector, or array-list of student objects (array-list and vectors are essentially dynamic arrays). C++ provides the most flexibility: you could achieve this list in C++ via a pointer to student records, a vector of student records, a linked list of student records, or a linked list of student objects (using a class but possibly two).

4.3.6 Pointers (continued)

For the purpose of comparison, figure 4.7 illustrates how this dynamic list of student records could be done via the linked list approach in Pascal, C++, and Java.

Figure 4.7: Illustrating Pointer Declarations in Pascal, C++, and Java

Figure 4.7a: Pascal BNF Definition of a Pointer and Construction of Linked List

```
PointerDefinition ::=
Type <PointerName> = ^<DataType>;

(* Example of a linked list of student records follows *)
Type
   StudentL = ^StudentRec;
   StudentRec =
   Record
        ID_Number:   Integer;
        LastName:    String;
        FirstName:   String;
        Major:       String;
        GPA:         Real;
        Next:        StudentL;
    End;
…
Var     First, Last: StudentL;
```

Figure 4.7b: C++ BNF Definition of a Pointer and Construction of Linked List

```
PointerDefinition ::=
<DataType> *

// Example of a linked list of student records
typedef char* String;
struct StudentRec
{
        int ID_Number;
        String LastName;
        String FirstName;
        String Major;
        double GPA;
};
struct StudentL
{
  StudentRec myRec;
  StudentL* Next;
};
// …
StudentL First, Last;
```

```
PointerDefinition ::=
<DataType> *

// Alternate example of a linked list of student records
typedef char* String;
struct StudentL
{
        int ID_Number;
        String LastName;
        String FirstName;
        String Major;
        double GPA;
        StudentL* Next;
};
// …
StudentL First, Last;
```

Figure 4.7: Illustrating Pointer Declarations in Pascal, C++, and Java (continued)

Figure 4.7c: Java BNF Definition of a Pointer and Construction of Linked List

```
public class StudentRec
{
        int ID_Number;
        String LastName, FirstName, Major;
        double GPA;

        public class StudentRec ( ) // Constructor
        { … };

        //  Other methods  to input, modify, and format information for output respectively
};

public class StudentNode
{
        StudentRec Info;
        StudentNode Next;

        public class StudentNode ( ) // Constructor
        { … };

        //  Other methods to input, modify, and format information for output respectively
};

public class StudentL
{
        StudentNode First, Last
        int Length;
}

// Notice the indirect approach required in Java as opposed to Pascal and C++
```

4.3.7 Other Advanced Data Types

There are several other abstract data types (ADTs) that programmers often wish to define and manipulate. The previous subsection mentioned array-lists, vectors, and linked lists. These three data types fall in this category. Other commonly used advanced data types include sets, stacks, queues, trees, and graphs. Programming languages that are to be used in rigorous programming scenarios (particularly in scientific and/or business environments) should support these features. Languages such as Pascal, C, C++, Java, C#, and Ada support these features or their equivalents. Of this group, Java and C++ are particularly impressive in their provision of flexibility with respect to these ADTs (albeit in very different ways).

4.4 Summary and Concluding Remarks

It's time to summarize what has been covered in the lecture:

- Most programming languages support the following primitive data types: integer, real number, Boolean, character, string. In some languages, *string* is treated as an advanced data type.
- An *identifier* is a programmer-ascribed that is used to refer to a program component (variable, subprogram, or label). A *variable* is a data item that has been defined typically on a specific data type.
- A general rule for an identifier is that it should not contain whitespace(s). Additionally, different programming may have other rules relating to identifiers.
- All programming languages have special *reserve words* which have predefined meanings. Identifiers are not allowed to be given reserve-word-names.
- In C-based languages (e.g. Java, C++, and C#), the data type is specified ahead of the identifier during variable declaration; in other languages (e.g. Pascal), the order is reversed; still other languages do not require variable declaration at all (e.g. Python).
- Some languages are strongly typed (e.g. Java and Pascal) while others are minimally typed (e.g. C).
- Some languages facilitate implicit data conversion from a lower ranger to a higher range whenever necessary, and explicit conversion for other scenarios (e.g. C++); other languages support explicit conversion only (e.g. Python), and some do not allow data conversion at all.
- Most languages appear to support static scoping; however, there are others that support dynamic scoping. Languages that support pointers (e.g. C and C++) tend to tacitly support dynamic scoping via pointers.
- Most languages support the concept of declaring *constants*.
- Among the well-known ADTs are the following: enumerated types, sub-range types, arrays, records, unions, pointers, classes, array-lists, vectors, linked lists, sets, stacks, queues, trees, graphs, etc. These ADTs are supported differently in different languages.

When considering a programming language for usage, special attention should be given to how these matters are handled (especially how ADTs are treated), as this often affects the effectiveness of the language for the problem domain in question.

4.5 Recommended Readings

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapter 5.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapters 5 & 6.

[Webber 2003] Webber, Adam B. 2003. *Modern Programming Languages: A Practical Introduction*. Wilsonville, Oregon: Franklin, Beedle & Associates. See chapter 6.