High Order Languages

Level HOL6, or: CS1 in 2 hours or less

Role of the compiler

• At the high-order language level, the view of the computer is a virtual machine that executes algorithms in the form of high-level language code, such as C++
• We already know that computers don’t really execute C++ code; in order to make C++ code executable, we must first run a program that translates the source code into machine language, or object code
• This program is, of course, the compiler

Why not a C++ machine?

• It is theoretically possible that a machine could be designed with C++ as its instruction set
• This is probably more trouble than it’s worth, since the circuitry for such a machine would be incredibly complicated (and thus, prone to error)

All compilers are not created equal

• Most of us are at least aware that there are different C++ compilers on the market, and that they offer different environmental features
• The differences between compilers go deeper than the interface, however
  – Visual C++ has a powerful integrated debugger; dev-cpp doesn’t
  – Dev-cpp handles string functions correctly according to the ANSI standard; Visual C++ doesn’t
• Keep in mind that a compiler is a program, and that different programs work differently

High order languages and platform independence

• A distinguishing characteristic of a high order language is its platform independence: you can, for example, write C++ code on a MacIntosh, then compile and execute the code on an Intel PC or IBM mainframe
• However, portability is somewhat relative: a program designed and written on a supercomputer may not execute correctly on a PC because of hardware differences (e.g. memory or bus capacity)

The Java Virtual Machine

• Java is a high order language that takes a unique approach to the semantic gap between level HOL6 and level ISA3
• Java source code is compiled to a pseudo machine language called Java Byte Code (JBC)
• The Java Virtual Machine (JVM) is an interpreter that runs JBC programs
• The JVM is the only platform-specific part of the Java scheme
The HOL6 virtual machine

- Conceptually, a high order language consists of:
  - A set of terms, symbols and rules for their use
  - A set of tools for describing algorithms
- At level HOL6, a computer system can be thought of as an engine for executing algorithms using these tools

High order languages

- There are several classes of general-purpose high order languages, including procedural, functional, declarative and object-oriented, to name four
- To simplify things, we will consider only the first of these

High order language characteristics

- Procedural languages (also known as imperative languages) share the following characteristics:
  - Most fundamental operations are concerned with storage and retrieval of data items
  - Programs consist of a sequence of statements that describe algorithms that manipulate data
- The next several slides describe other characteristics common to procedural languages

Program Structure

- In most procedural languages, functions consist of two discrete sections:
  - Data definition
  - Instructions
- Some languages are stricter than others in enforcing the separation of these sections; in C++, for example, data declaration statements can occur in the midst of other instructions
- But even in C++, variables must be declared before they can be used in statements

Simple Data Types

- Numeric types
  - Whole numbers: C and C++ have the int type and its variants; Pascal has integer
  - Floating-point numbers: C and C++ have data types float, double and long double; Pascal has real
  - In BASIC, all numbers are floating-point type by default; an integer variable is indicated by the use of the % symbol at the end of the variable name

Simple Data Types

- Alphanumeric: literal (usually ASCII) value of any single character; C, C++ and Pascal all call this data type char
- Logical: values true or false; C++ has bool, Pascal has boolean, C and BASIC don’t have an explicit logical type
Simple Data Types

- String: data consisting of one or more grouped alphanumeric characters
- Debatable whether string is really a "simple" type
  - Neither C nor standard Pascal contains this type explicitly; can use char arrays to get string functionality
  - C++ doesn’t include the type as part of the language proper, but it’s part of the standard set of libraries
  - In BASIC, a variable with a $ at the end of its name can store a string

Structured Data Types

- In general, structured data types are mechanisms for creating compound structures from simpler data types
- Structured data types include arrays, records, and files

Arrays

- Collections of some fixed number of contiguous memory blocks, each capable of holding data of the same type
- Each element accessible via subscript notation
- In Pascal, an array is declared as a new data type (sort of like enums in C++); in C/C++ an array is declared as a variable

Records

- Aggregate data type
- Allows construction of database model; individual fields, each containing one facet of data item
- In Pascal and C/C++, record structure is declared as a data type (in C/C++ a record is called a struct)

Files

- Similar to array, but not fixed in size
- May be restricted to sequential access
- In Pascal, C and BASIC, file is treated as specialized data container
- In C++, file is an input or output stream

Pointers

- Simple data type used as indirect reference to other data
- Basis for reference parameters in C++
- Not all high order languages support pointers (at least in terms of direct access)
Data Representation and Memory Allocation

- Literal values: explicit data value of any type; supported by all high order languages, although syntax for representation may vary
- Constant: data value assigned to an identifier; once assigned, can’t be changed
  - Explicitly supported in Pascal and C++
  - Usually handled via a #define directive in C
  - Not supported in BASIC

Variables

- Named memory blocks
- Hold values of specific type; can be assigned new values during course of program
  - In Pascal and C/C++, must declare variable prior to use
  - In BASIC, no formal declaration is made, but identifier may indicate data type (using % or $, for example)

Executable statements in high order languages

- Three basic types of instructions:
  - Data movement
  - Data manipulation
  - Program control

Data movement statements

- Three kinds of data movement can occur in a program, classified by data source destination:
  - Internal to internal
  - External to internal
  - Internal to external
- Data can move from one external (to the program) location to another, but since this by definition occurs outside the program, we won’t consider it here

Internal to internal data movement

- Data already resides in memory; we’re just moving it from one location to another
- Usually accomplished via an assignment statement:
  - X := 4;  [Pascal]
  - X = 4;  // C or C++
  - LET X = 4  //BASIC
- In each case, the element on the left of the assignment operator is a valid identifier; in the first two cases, the identifier must be a previously-declared variable, while the element on the right can be a literal value, constant, or other valid expression

External to internal data movement: input

- In most of our example languages, input is accomplished using function calls
  - Pascal: read and readln
  - BASIC: INPUT & READ
  - C: scanf, gete, getchar, gets
- C++ has operator >> (as well as various functions)
  - but this operator has a function behind it, as you already know if you have ever overloaded this operator
**Internal to external data movement:**

- Similar to input:
  - Pascal provides write and writeln
  - C has printf, puts and putc
  - BASIC has PRINT
  - C++ has << and various functions
- In Pascal and BASIC, the I/O functions are built into the language; in C and C++, they are not
- All of the above allow for I/O redirection so that data can be read from or written to an alternate location, such as a file

---

**Data manipulation statements**

- Operations on string data, such as division into substrings and concatenation (combining strings) are not explicitly built into any of our example languages
- Library functions associated with C and C++ provide these operations

---

**Data conversion operations**

- High order languages can be classified as strongly typed (like Pascal) or weakly typed (like C and C++)
- In a strongly typed language, conversion of stored data from one type to another requires an explicit casting operation
- In C and C++, explicit casting can be done, but implicit type coercion/conversion is a common occurrence

---

**Data manipulation statements**

- For arithmetic calculations, all the example languages have the four basic arithmetic operators: +, -, *, / which can be applied to numeric data
- In Pascal, the / operator can only be applied to real numbers; operators DIV and MOD are used for integer division
- In C/C++, the % operator is used for modulus

---

**Control structures**

- By default, all high order language programs proceed sequentially
- Control structures alter the flow of program logic; the two major types of control structures are:
  - Conditional (branching) structures
  - Iteration (looping) structures
Conditional structures

- Simple branching statements occur in all of our example languages:
  - C and C++ have if and if/else
  - Pascal and BASIC have if/then and if/then/else
- In all cases, nesting of these structures is allowed; again, there is syntactic variation among languages

Conditional structures

- For multiway branching, all of the example language support some variation of the case statement
  - C and C++: switch/case with optional default
  - Pascal: case statement has no default clause
  - BASIC: several variations, including one that allows testing a range of values (not available in any of the other examples)

Iteration constructs

- Several looping constructs supported by high order languages, including:
  - Pretest loops
    - C/C++ while, Pascal while/do, BASIC do while/loop
  - Post-test loops
    - C/C++ do/while, Pascal repeat/until, BASIC do/loop
  - Count-controlled loops
    - Specialized for or pretest loop
    - All example languages support some form of for or for/next loop

Modularity constructs

- Modularity refers to the ability to compartmentalize program tasks to facilitate debugging and maintenance
- In general, a program block is a set of statements with a marked beginning and end
  - Block delineated by { } in C/C++
  - Pascal uses begin/end
  - BASIC uses more primitive structure: block begins with a LABEL and ends with a RETURN or EXIT statement

Subprograms

- A subprogram is a named program block usually containing code that performs some specific algorithm
- Data are supplied to subprograms in the form of parameters:
  - Formal parameters: specified in block definition
  - Actual parameters: passed from calling program (aka arguments)

Subprograms and parameter passing

- High order languages that support parameter passing use positional matching of actual and formal parameters
- Strongly-typed languages won’t allow data type mismatches; weakly-typed languages will, at least some of the time
Functions Vs. Procedures

• In most high-order languages, a function is subroutine that returns a single value
• C/C++ refers to all subprograms as functions, but only value-returning functions are truly functions in the broadest definition of the term
• A subprogram that does not return a value (but which may or may not permanently alter parameters passed to it) is referred to as a procedure — so a void C/C++ function is a procedure

Parameter passing methods

• Pass by value: a copy of the actual parameter is passed to the subprogram; any change to the formal parameter has no effect on the value of the actual parameter
• Pass by reference: also known as pass by address; the formal parameter and the actual parameter both refer to the same block of memory, so changes to the formal parameter are also changes to the actual parameter
  • In Pascal, pass by reference is accomplished using VAR parameters
  • In C++, the & symbol appended to the formal parameter’s data type makes it a reference parameter
  • In C, there are no reference parameters as such, but the same thing can be accomplished by using pointers as formal parameters

Anatomy of a procedure call

• When a procedure (or function) in a high order language is called, storage is allocated from the computer’s runtime stack
  • Referred to as the “runtime” stack because the allocation takes place during program execution (not allocated at compile time)
  • A stack is something like an array that is accessible only at one end:
    • A push operation stores a value at the end (top) of the stack
    • A pop operation retrieves a value from the top of the stack
  • This storage scheme is called LIFO (last in, first out)

Anatomy of a procedure call

• Activation record or stack frame: the collection of items pushed on the stack when a procedure is called; includes
  • Return value (if procedure is a function)
  • Actual parameters
  • Return address
  • Storage for local variables

Example

```c++
#include <iostream.h>
void printBar(int n)
{
    for (int i=0; i<n; i++)
        cout << "*
    cout << endl;
}
int main()
{
    int numPts;
    cin >> numPts;
    for (int i=0; i<numPts; i++)
    {
        cin >> value;
        printBar(value);
    } // rai (return address) = return 0;
```

Snapshots of runtime stack for example program

<table>
<thead>
<tr>
<th>Main program</th>
<th>Run-time stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>numPoints</td>
<td>?</td>
</tr>
<tr>
<td>value</td>
<td>?</td>
</tr>
<tr>
<td>i</td>
<td>?</td>
</tr>
</tbody>
</table>

Program start

First call to printBar function
Procedure deallocation process

- Items on run-time stack are deallocated in reverse order of allocation:
  - Local variable storage deallocated
  - Return address popped
  - Parameters deallocated
  - Return value (if function) deallocated
- Program uses return address to determine where program execution should resume; will be the statement immediately following the procedure call

Example with reference parameters

```c
#include <iostream.h>

int main()
{
    int a, b;
    cout << "Enter value 1: ";
    cin >> a;
    cout << "Enter value 2: ";
    cin >> b;
    order(a, b);
    // ra2
}
```

Recursion

- In mathematics, a recursive definition of a function is one that uses the function itself
- For example, the factorial function can be defined as:
  \( f(n) = n(f(n-1)) \)
- In a high order language, a recursive procedure is a procedure that calls itself

Example: recursive factorial program

```c
#include <iostream.h>

int fact(int n)
{
    if (n <= 1)
        return 1;
    return n * fact(n-1);
    // ra2
}
int main()
{
    int num = 4;
    cout << "4! = " << fact(4) << endl; // ra1
    return 0;
}
```

Runtime stack after last recursive call

As long as \( n > 1 \), the function calls itself with an argument of \( n-1 \)

Each recursive call pushes a new activation record on the runtime stack

When \( n = 1 \), the non-recursive case is reached and the deallocation process commences
Calling sequence

Each solid arrow represents a function call
The dotted arrows represent returns, with the returned value displayed

Recursive thinking

- The microscopic view of recursion, illustrated by the previous example, is useful in understanding how recursion works at a lower level of abstraction
- However, to write a recursive function, you need to think macroscopically; forget about the run-time stack, and assume it’s possible to make a recursive call

Mathematical induction and recursion

- Inductive proof requires two key elements:
  - Establish the basis (show the theorem is valid for n=1)
  - Assuming the theorem is valid for n, prove it for n+1
- Designing a recursive function requires similar reasoning:
  - Compute the value for the basis step
  - Assuming the function for n-1, write it for n
- The key factor is that the problem gets smaller with each recursive call

Computing the expansion of binomial coefficients

- Examples:
  \((x+y)^1 = x+y\)
  \((x+y)^2 = x^2+2xy+y^2\)
  \((x+y)^3 = x^3+3x^2y+3xy^2+y^3\)
  \((x+y)^4 = x^4+4x^3y+6x^2y^2+4xy^3+y^4\)
  Etc.
- Mathematically, the binomial coefficient of \(b(n,k)\) to the power \(n\) and term \(k\) is:
  \(b(n,k) = b(n-1,k) + b(n-1,k-1)\) for \(0 \leq k \leq n\)

Pascal’s Triangle

The coefficient of the kth term for power n is the sum of the kth term’s coefficient and the (k-1)th term’s coefficient for power n-1

Recursive computation of binomial coefficients

```c
int BC (int n, int k)
{
    int y1, y2;
    if (k==0 || n==k)
        return 1;
    else
    {
        y1=BC(n-1, k);
        y2=BC(n-1, k-1);
        return y1+y2;
    }
}
```
Call tree for initial call of BC(3,1)

Mutual recursion

- Functions may be recursive indirectly
- Suppose procedure a calls procedure b, which then calls procedure a: the two procedures are mutually recursive
- In C/C++, the use of function prototypes facilitates this possibility, since a function must be declared before it can be called