Homework 1

Shreyas Jayant Havaldar
CS18BTECH11042
This document is generated by LaTeX

CS2433: PRINCIPLES OF PROGRAMMING LANGUAGES-II
IIT Hyderabad

February 8, 2020

Ch 6: Control Flow

Check Your Understanding

Question 2

What distinguishes operators from other sorts of functions?
Operators are specific pre-defined functions native to the language that we use directly in our programs. Functions need us to define function name, return type, arguments and the signature ourselves. Operators are rigid in the way of using them but functions can be defined with much more freedom. Operators are much simpler to use and can be directly used in our program.

Question 12

Given the ability to assign a value into a variable, why is it useful to be able to specify an initial value?
For variables allocated statically, the initial value can be allocated global memory at compile time itself preventing the run time allocation overhead. Initialization of variables also ensures that incorrect usage of the variables with garbage values does not occur.

Question 16

Why is it impossible to catch all uses of uninitialized variables at compile time?
It might occur that some variables are allocated memory dynamically in the heap or local to subroutine in stacks during run time. Thus these variables are not caught uninitialized at compile time.

Question 17

Why do most languages leave unspecified the order in which the arguments of an operator or function are evaluated?

This is done to give freedom to compiler designers to be able to optimize the compilation by adjusting the evaluation direction depending on the arguments as the order of evaluation impacts both instruction scheduling and allocation of registers for the operands. Example follows:

```
int d = 4;
int c = A[i];
p = c*7 - 77*d;
//Here right to left evaluation is faster as c needs to be retrieved from A
```

Question 18

What is short-circuit Boolean evaluation? Why is it useful?

Short circuit evaluation occurs when all the conditions provided in the check boolean of a conditional statement are not necessary to determine the output of the logical expression. The first statement's evaluation is sufficient to exhaust the logical possibilities. For example in a string of AND statements, if the first one is false the logical expression will always evaluate to FALSE and thus no further expression is checked and short circuit occurs as FALSE is returned. It is useful to reduce the computation time which is inherently unnecessary while evaluating long logical expressions. It is also important when checking of the following logical expressions depends on the value of variables in the expressions preceding them or and might go out of bound. Example follows:

```
    if ((a!=0) && (k/a > 2)) {
    //Do Something
    }

```

Question 28

Why do most languages not allow the bounds or increment of an enumeration-controlled loop to be floating-point numbers?

Floating point numbers are imprecise because of the way they are stored. We cannot ensure the equality of a floating point number to an exact real number always. Due to this imprecision we always check for equality between two floating point numbers using a range so that we can account for the error. If the bounds or increment of an enumeration controlled loops are allowed to be floating point numbers then it might happen that due to imprecision the bounds are exceeded because the condition is missed due to minor differences in bound values and counter values. Also the increment would thus be imprecise too and might lead to incorrect no of loop executions. The problem with real-number sequences is that limited precision can cause comparisons (e.g., between the index and the bound) to produce unexpected or even implementation-dependent results when the values are close to one another. Should for x := 1.0 to 2.0 by 1.0 / 3.0

Exercises

Question 6.1

We noted in Section 6.1.1 that most binary arithmetic operators are left- associative in most programming languages. In Section 6.1.4, however, we also noted that most compilers are free to evaluate the operands of a binary operator in either order. Are these statements contradictory? Why or why not?
No. These statements are not contradictory as both are independent. Associativity does not determine the order of evaluation of operands but simply attaches the operands to their respective operands. Likewise the evaluation of operands is undertaken by the compiler and gives a result which is individually independent of the associativity, and just the overall value of expression combines both these results to give us an final answer.

Question 6.4

Translate the following expression into postfix and prefix notation: $[b + sqrt(b \times b - 4 \times a \times c)]/(2 \times a)$ **Do you need a special symbol for unary negation?**
Yes. We need a symbol for unary negation to take care of -b. Let that symbol be !.
Postfix: b ! b b * 4 a * c * - sqrt + 2 a * /
Prefix: / + ! b sqrt - * b b * * 4 a c * 2 a

Question 6.7

Is &(&i) ever valid in C? Explain.
No. The unary & operator requires a lvalue(refers to memory location which identifies an object) as it's argument and returns a rvalue(refers to data value that is stored at some memory address). Thus the &i is a rvalue and cannot be operated upon by the & operator. Therefore the operation is not valid in C.

Question 6.8

Languages that employ a reference model of variables also tend to employ automatic garbage collection. Is this more than a coincidence? Explain.
Yes. This is not merely a coincidence as languages which employ reference model of variable are forced to maintain multiple pointers to the same address in the heap and thus employ automatic garbage collection to ensure all the created pointers are taken care of. Each reference refers to an object in the heap memory and garbage collector reclaims the memory occupied by the object to prevent errors of memory leak. If an object is unreachable, which can be understood if there are no references to the object, the object is occupying unnecessary memory and can be then reclaimed by the automatic garbage collector without worrying about it's usage in the program as no references exist.

Question 6.24

Rubin [Rub87] used the following example (rewritten here in C) to argue in favor of a goto statement:

```
int first_zero_row = -1;
/* none */
int i, j;
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
if (A[i][j]) goto next;
}
first_zero_row = i;
break;
next: ;
}
```

The intent of the code is to find the first all-zero row, if any, of an n n matrix. Do you find the example convincing? Is there a good structured alternative in C? In any language?

No. The example is not at all convincing as we could easily use a check variable and implement the same logic depending on that. Example follows:

```
int first_zero_row = -1;
/* none */

int c = 0;
int i, j;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (A[i][j]) {
            c = -1; break;
        }
    }
    if (c==0) {
        first_zero_row = i;
        break;
    } else {
        c = 0;
    }
}
```

Ch 7: Data Types

Check Your Understanding

Question 2

What does it mean for a language to be strongly typed? Statically typed? What prevents, say, C from being strongly typed?

Strongly typed: The language prohibits the occurrence of an operation on any object whose type does not match the intended, depending on its own unique implementation and is verified at runtime. An type error occurs at runtime if we try to perform an operation on an object whose type does not support it.
Statically typed: The language is strongly typed but performs type checking at compile time. The type error if any is thrown at compile time. The compiler can statically assign the correct type to every expression. C is not a strongly typed language because it makes use of subroutines with variable no of parameters, unions and inter-operability of arrays and pointers. It does not perform any check at runtime.

Question 7

Give two examples of languages that lack a Boolean type. What do they use instead?
C90 and LISP.
C90 expects an integer with 0 meaning false and everything else representing true.
LISP uses an empty list for false and anything else is interpreted as true.

Question 10

What are aggregates?
Aggregates are built-up structured values of user-defined composite data types. Aggregates are usually found in object oriented languages where classes are defined to contain references to objects from other classes and an aggregate object contains other objects. Aggregates assign multiple values to elements of a record or an array, creating composite types. They help associate expressions and individual elements to each other.
Example in Ada follows:

```
type student is record
name : string (1..10);
age : integer;
dept : string (1..10)
end record;
s : student;
s := (name => "SJH", age => 19, dept => "CSE");
```

Question 11

What is the difference between type equivalence and type compatibility?
Type compatibility determines whether or not an object of a certain type can be used in a certain context. **Type equivalence** is more rigid as it requires both objects to be composed of same definitions and combined in the same manner to be structurally equivalent, or lexically identical for name equivalence. Most programming languages are flexible enough to allow for type compatibility and not enforce type equivalence in most contexts.

For example the operators like - allow operands to be compatible with any common type supporting subtraction. Likewise for subroutine calls the formal parameters and arguments must be compatible. The flexibility allowed in type compatibility differs from language to language with where *Ada* requires either equivalence or one being subtype of another or both equidimensonal arrays for 2 types to be compatible.

Question 15

Explain the differences among type conversion, type coercion, and nonconverting type casts.
r is a float.
Type conversion is said to occur when the programmer wants to use a value of one type in a context where another type is expected. There is a change in the storage format which might result in loss of precision or truncation of excess bits. The conversion may or may not occur at run time. eg: n = (int) r
Coercion is said to occur when the language implicitly and automatically converts the value of a type into the expected type for the context. There is usually no loss of precision. eg: r = n
Non converting type casts is said to occur where the type of the value is changed without changing the low level hardware implementation, interpreting the same set of bits in a different manner. eg: n = cast_float_to_int(f)

Question 19

What is type inference? Describe three contexts in which it occurs.
Type inference is the determination of the type of the expression based on the types of the symbols involved in the expression.
Result of a function call has the same type as the type in the function's header.
Result of a comparison is Boolean.
Result of an assignment is of the same type as the left hand operand.

Question 23

Why is it easier to implement assignment than comparison for records?
Assignments are a single operation irrespective of the record size for all records. But comparisons between two records can be line by line, but comparisons on whole record blocks can fail due to garbage values in holes. These garbage values need to be taken care of for correct comparisons which need additional routines or have field by field comparison routine for every record type which is thus much more tedious than implementing assignment.

Question 26

Briefly describe two purposes for unions/variant records.

System Programs: Unions allow the same set of bytes to be interpreted in different ways to provide for different uses. Memory management makes use of unions for storage space to interpret it as un-allocated, book-keeping or user-allocated memory space.
Alternative sets of fields: A record for a student may have his/her ID, name, major etc. as several common fields depending on if the student is a day scholar, residential or long distance student of the institute.

Question 33

Discuss the comparative advantages of contiguous and row-pointer layout for arrays.
Contiguous layout usually requires much lesser space and provide for more direct access to the individual elements in multidimensional arrays by directly referring to the indices.
Though the row-pointer layout requires more memory space to account for all pointers, it allows the formation of a ragged array by allowing each element's row to be of different length and preventing devotion of space to holes, offsetting some memory space. The array can also be constructed from pre-existing rows without copying overhead. It also allows individual elements to be accessed faster sometimes on machines with slower multiplication operations.

Question 34

Explain the difference between row-major and column-major layout for contiguously allocated arrays. Why does a programmer need to know which layout the compiler uses? Why do most language designers consider row-major layout to be better?
The elements of a row are contiguous in memory space in row-major order, the elements of a column are contiguous if it is a column major order. For eg: a[1][1] is followed by a[1][2] in memory space in row major, whereas a[1][1] is followed by a[2][1] in memory space in column major. It is necessary for the programmer to know which layout the compiler uses so that he/she can optimize his program for cache resuse and computation while dealing with large 2-d arrays. For eg: Nested loops in matrix multiplication. The inner loop needs to be across the row for row major and across the column for column major for better performance. Most language designers consider row-major to be better because it is easier to define the multidimensional arrays as array of subarrays which constitute the rows.

Question 42

Discuss the advantages and disadvantages of the interoperability of pointers and arrays in C.
Advantages: Pointer based array traversal eliminated redundant address calculations on the older compilers of C. Pointer based referral to array also allows for dynamic allocation of memory in the heap. Arrays are always passed as pointers so it might be natural to refer to them as pointers throughout the program. Pointer arithmetic and operations scale their results depending on size of referenced arrays and thus provide for tools for working with arrays.

Disadvantages: Both are not naturally same during declaration so the initialization and difference in allocation needs to be kept in mind. Declaration of pointer variables allocates space for pointers but not for the array. This needs to be kept in mind while using pointers interchangably with arrays.

Question 45

What is garbage? How is it created, and why is it a problem? Discuss the comparative advantages of reference counts and tracing collection as a means of solving the problem.

Garbage is an object or collection of objects that are no longer useful. It is created when all the references to that object are destroyed and the object just wastes space in the memory. The object cannot be reached by using any no of pointers and thus is problematic as it simply eats up memory space which could have been otherwise used judiciously. The aggregation of garbage leads to accumulation of lots otherwise useful memory space. Reference counts maintains the count of the number of pointers that refer to the object. When the count is 0, the object memory space is reclaimed. It's advantage is that it is simple to maintain. Tracing collection defines useful objects in a better way as some objects might not be useful even when some pointers refer to them. So if an object can be traced to by valid pointers starting from outside heap it is said to be useful. This brings about even better utilization of memory.

Question 46

Summarize the differences among mark-and-sweep, stop-and-copy, and generational garbage collection.

Mark and Sweep: One of the earliest garbage collection algorithms designed to collect garbage in 2 steps:
Mark: Visiting all reachable objects and marking them as visited.
Sweep: Sweeping through all available objects in memory, and then adding the unmarked ones to the free list of objects to reallocate. Disadvantage is that that collection overhead is directly proportional to the size of memory, which may be huge in some cases.
Stop and Copy: Dividing the heap into semispaces, and copying reachable objects between semispaces during collection. As only reachable objects are the ones visited, the overhead of copying is not proportional to the size of memory, but is rather small. Other advantage being that reachable objects are placed contiguously when copied and thus are compacted, taking less memory.
Generational garbage collection: Dividing a program's heap into regions or generations containing objects of different ages, focusing the effort of garbage collection on the youngest objects because provably young objects tend more to become garbage. Advantages being that the collection references are localized does not disrupt the reference locality of the program and obviously that collecting a even smaller region takes lesser time and thus does not disrupt interactive users.

Exercises

Question 7.6

Ada provides two "remainder" operators, rem and mod for integer types, defined as follows [Ame83, Sec. 4.5.5]: *Integer division and remainder are defined by the relation A = (A/B)*B + (A rem B) , where (A rem B) has the sign of A and an absolute value less than the absolute value of B . Integer division satisfies the identity (-A)/B = -(A/B) = A/(-B) . The result of the modulus operation is such that (A mod B) has the sign of B and an absolute value less than the absolute value of B ; in addition, for some integer value N , this result must satisfy the relation A = B*N + (A mod B)*
Give values of A and B for which A rem B and A mod B differ. For what purposes would one operation be more useful than the other? Does it make sense to provide both, or is it overkill? Consider also the % operator of C and the mod operator of Pascal. The designers of these languages could have picked semantics resembling those of either Ada's rem or its mod. Which did they pick? Do you think they made the right choice?
The example which gives contrasting answers for same values of A and B is:
A = -6; B = 7
A rem B = -6
A mod B = 1

The rem operator is more useful when we want to refer to the sign of A and it holds significance in the operations that follow. Rem maintains the negative sign of A if any in the remainder we obtain.
The mod operator seems more intuitive by its definition as N is the quotient for A/B and the result obtained from mod operation can be obtained to the quotient*B to get A back.
It is overkill as the other can always be obtained from the one language provides functionality for as follows:
(A mod B) = (A rem B) + B

The % operator in C and mod in Pascal both follow the definition of Ada's mod. It seems like the correct choice to me because the remainder being the same sign of the divisor seems more intuitive and fits easily into the basic equation A = N*B + (A mod B).

Question 7.15

Consider the following C declaration, compiled on a 32-bit Pentium machine:

```
struct {
int n;
char c;
} A[10][10];
```

If the address of A[0][0] is 1000 (decimal), what is the address of A[3][7] ? Size of the struct is 4 + 4 = 8.

Following the row major format that C uses, address of A[3][7] is $= 1000 + (3 \times 10 \times 8) + (7 \times 8) = 1296$.

Question 7.17

Suppose A is a 1010 array of (4-byte) integers, indexed from [0][0] through [9][9]. Suppose further that the address of A is currently in register r1 , the value of integer i is currently in register r2 , and the value of integer j is currently in register r3. Give pseudo-assembly language for a code sequence that will load the value of A[i][j] into register r1 (a) assuming that A is implemented using (row-major) contiguous allocation; (b) assuming that A is implemented using row pointers. Each line of your pseudocode should correspond to a single instruction on a typical modern machine. You may use as many registers as you need. You need not preserve the values in r1 , r2 , and r3 . You may assume that i and j are in bounds, and that addresses are 4 bytes long. Which code sequence is likely to be faster? Why?

(a)

```
mul r2 r2 40
/* multiplying value of i with 40 = number of bytes in a row, to reach the
   correct row */
mul r3 r3 4
/* multiplying value of j with 4 = the size of an int element to reach the
   correct column */
add r1 r1 r2
add r1 r1 r3
// reaching the required location A[i][j] in the array
load r1 *r1 // loading the element at the address r1
```

(b)

```
mul r2 r2 4 // multiplying value of i with 4 = the size of pointer, to reach
   correct row pointer
add r1 r1 r2 // finding the address to the row i by adding to the address of A
load r1 *r1 // loading the address of the row i into r1
mul r3 r3 4
/* multiplying value of j with 4 = the size of an int element in the array,
   using bit shift operator */
add r1 r1 r3 // loading the address of the element at column index j
load r1 *r1 // loading the element at the address r1
```

(a) should be faster because it performs only one load operation and eliminates the need for loading twice to save the row pointer and the eventual pointer to the final required element.

Question 7.20

Explain the meaning of the following C declarations: double *a[n]; double (*b)[n]; double (*c[n])(); double (*d())[n];

double *a[n] : an array of n pointers to doubles
double (*b)[n]: pointer to an array of n doubles
double (*c[n])(): an array of n pointers to functions returning doubles
double (*d())[n]: a function returning pointer to an array of n doubles

Question 7.49

Learn about weak references in Java. How do they interact with garbage collection? Describe several scenarios in which they may be useful.

According to the documentation provided by Oracle and baldeung.com, weak reference objects are objects which do not prevent their referents from being made finalizable, finalized, and then reclaimed. Weak references are most often used to implement canonicalizing mappings.

A weak reference is definitively a reference to an object made such that it is not strong enough by itself to make the object to remain in memory. So, weak references let the garbage collector decide an object's reach-ability and whether the object in question should be kept in memory or not. It has neither strong or soft references pointing to it. It can only be reached through travelling across weak references.

java.lang.ref.WeakReference class is to be imported to deal with and create weak references. Weak reference objects are inherently connected to garbage collection. Let the garbage collector determine at a certain point in time that an object is weakly reachable. Then it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references. Simultaneously it will declare all of the formerly weakly-reachable objects to be finalizable also en-queuing those newly-cleared weak references that are registered with reference queues. Thus this approach clears up much more memory and leads to efficient memory management.

Apart from canonicalized mapping, when the maps hold only one instance of a particular value, weak references are oft-used in *WeakHashMap* class. The map interface is implemented by storing every key value as a weak reference, with the key-value pairs extending the *WeakReference* class. When the garbage collector removes a key, the entity mapped to the key is removed as well. Example follows:

```
1  import java.lang.ref.WeakReference;
2  class Weak
3  {
4      public void foo()
5      {
6          System.out.println("Hello");
7      }
8  }
9
10 public class Ref
11 {
12     public static void main(String[] args)
13     {
14         // Creating Strong Reference
```

```
15          Weak obj1 = new Weak ();
16          obj1.foo();
17
18          /* Creating Weak Reference to Weak-type object to which obj1 is also
    pointing. */
19          WeakReference<Weak> weakref = new WeakReference<Weak>(obj1);
20
21          /* Now, Weak-type object to which obj1 was pointing earlier is not
    available for garbage collection. But will be only be garbage collected
    when JVM needs memory. */
22          Obj1 = null;
23
24          / We can also retrieve back the object which has been weakly
    referenced. It also succesfully calls the method */
25          Obj1 = weakref.get();
26
27          Obj1.foo();
28      }
29  }
```

Another situation in which they are used are when solving the Lapsed Listener problem. All the listeners are held as strong references by the subjects to be able to get notified as any event occurs. The listener can't detach from the subject and thus cannot be collected as garbage as a strong reference is still available through the subject, leading to memory leak. So to partially solve this problem the listener can be made to be only weakly referenced from the subject. Thus they can be collected automatically without being explicitly dereferenced from the subject.