CS2040: POPL: Homework #1

Akilesh B

CS13B1042

Control Flow

PLP CYU:

2) We generally use the term operator for built-in functions that use special, simple syntax. Function calls consist of a function name followed by a parenthesized comma-separated list of arguments. Eg. my_func(A,B,C). Operators are simpler taking only one or two arguments. They get rid of parentheses and commas.
   Eg: a+b, -c.
   In C++, a+b is short for a.operator+(b). Some languages define their operators as syntactic-sugar for their "normal" looking functions.

12) The reasons why such initial values may be useful:
   a) Static variable that is local to a subroutine needs an initial value in order to be useful.
   b) For any statically allocated variable, an initial value that is specified in the declaration can be pre allocated in global memory by the compiler, avoiding the cost of assigning an initial value at run time.
   c) Accidental use of an uninitialized variable is most common programming errors. To prevent such errors, give every variable a value when it is first declared.

16) Initialization saves time only for variables that are statically allocated. Variables allocated in the stack or heap at run time must be initialized at run time. The problem of using an uninitialized variable occurs not only after elaboration, but also as a result of any operation that destroys a variable's value without providing a new one. Two such common operations are explicit de allocation of an object referenced through a pointer and modification of the tag of a variant record.

17) The two main reasons are
  a) Side effects: Depends on the order in which arguments are evaluated.

  b) Code improvement: Order of evaluation of sub expressions has an impact on both register allocation and instruction scheduling. In the expression a * b + f(c), it is desirable to call f before evaluating a*b, because if the product, if calculated first, would need to be saved during the call to f and f might want to use all the registers in which it might be easily saved.

18) Consider the expression (a<b) and (b<c). If a is greater than b, there is really no point in checking to see whether b is less than c as the overall expression must be false. In a similar manner, (a>b) or (b>c) , if a is greater than b there is no point in checking if b is greater than c, as overall expression must be true. Short circuit evaluation saves significant time in certain situations.

28) For precision purposes and accuracy. It will lead to an ambiguous situation and numerical imprecision. The problem with real-number sequences is that limited precision can cause comparisons to produce unexpected results when the values are close to one another.

Exercises:

Ex 6.1

No, they are not contradictory. When there are consecutive identical operators within an expression, associativity determines which sub expressions are arguments of which operators. It does not determine the order in which those sub expressions are evaluated.

For example, left associativity for subtraction determines that f(a) - g(b) - h(c) groups as
(f(a) - g(b)) - h(c) (rather than f(a)-(g(b)-h(c)), but it does not determine whether f or g is called first.

Ex 6.4

Yes, it is needed and let it be ~ (unary negation).
Prefix: / + ~ b sqrt - * b b * * 4 a c * 2 a

Postfix: b ~ b b * 4 a * c * - sqrt + 2 a * /

Ex 6.7

No, it gives an error. First of all &i is a compile time construct., the memory location of i gets translated at compile time to the address of i (i.e. &i). &(&i) cannot compile because it'd be the equivalent to the syntax: &(address of i). The address of address of i ??

Ex 6.8

Yes, this is more than a coincidence. Where a language with a reference model there will be copies of the same object. Manually it is very difficult for the programmer to keep track of the number of references to any given object, and specifically of whether a given object is no longer referenced at all. Garbage collection becomes more or less essential.

Ex 6.24

No, it's not convincing.

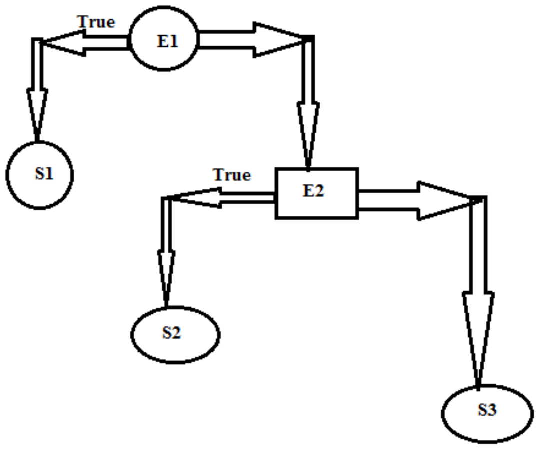
Alternative in C:

```
int  i, j;

for(i = 0; i < n; i ++){

        if(all_zero(A[i])){

                return i;

        }

}


int all_zero (int[] row){

        int i;

        for(i = 0; i < n; i ++ ){

                if(row[i]) return 0;
```

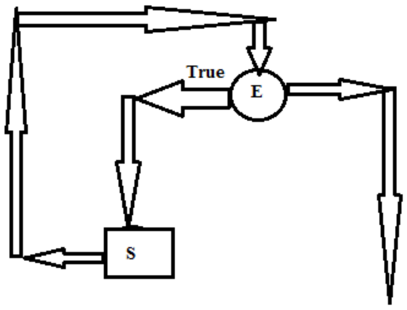
}}

PLCC: Ch #3

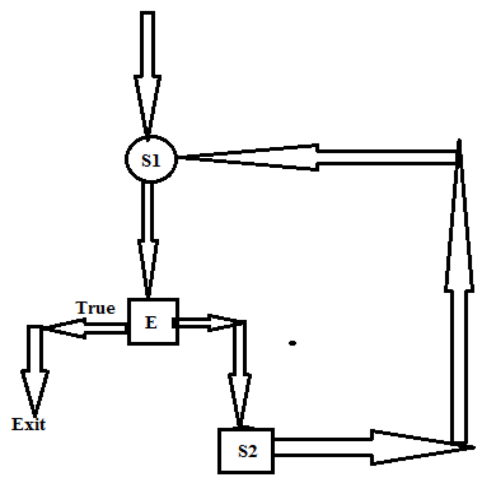
3.2 a)



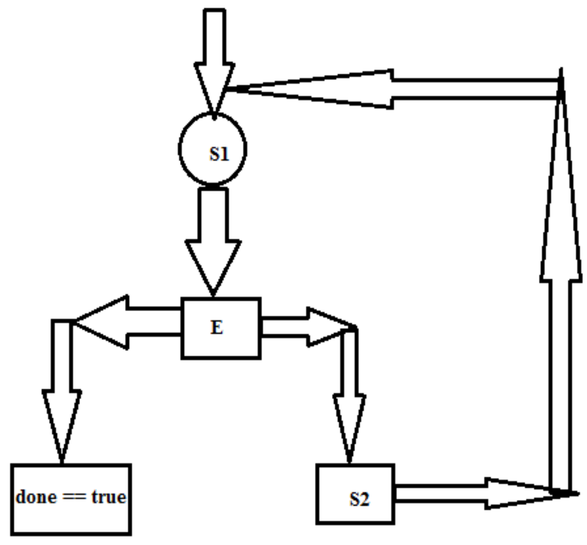
3.2 b)



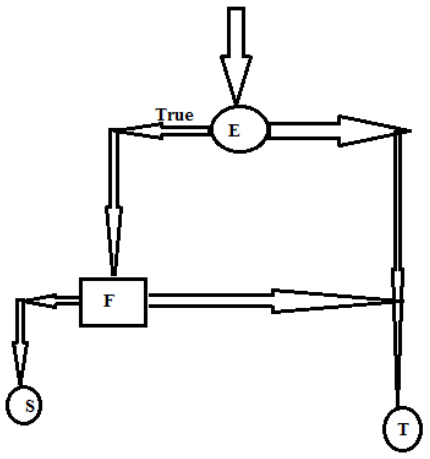
3.2 c)



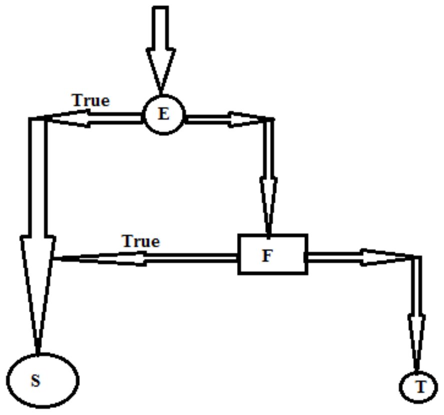
3.2 d)



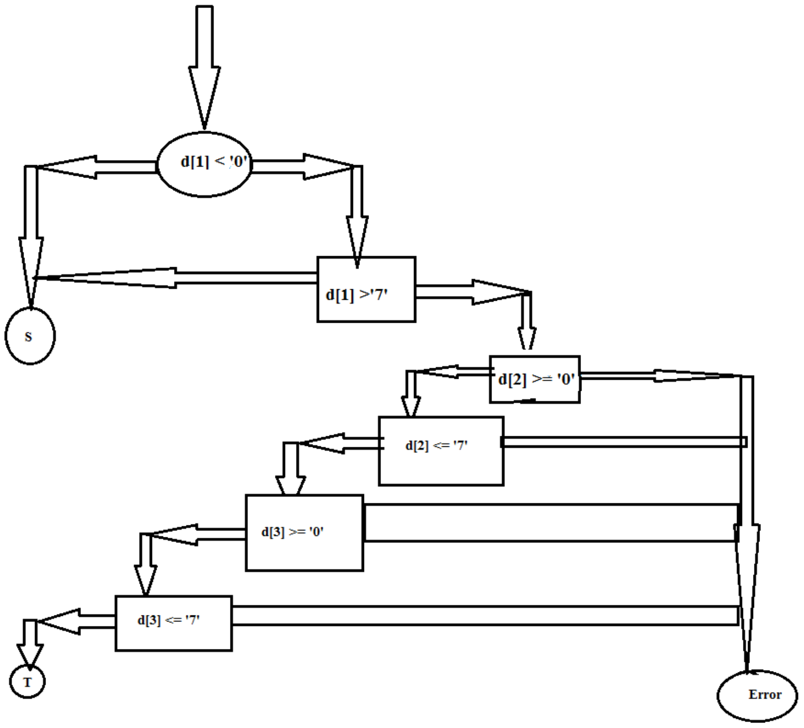
3.3 a)



3.3 b)



3.3 c)



Types

PLP Ch #7

CYU:

2) A language is said to be strongly typed if it generates an error or refuse to compile if the argument passed to the function does not closely match the expected type. A weakly typed language may produce unpredictable results or may perform implicit type conversion. A language is statically typed if it is strongly typed and type checking can be performed at compile time. C is weakly typed as the compiler allows operations such as assignment and comparison among variables of different types. C also allows value of a variable to be cast to another type.

7) C lacks a Boolean type. C expects an integer in place of a Boolean value, zero means false and anything else means true. Icon replaces Booleans with a more general notion of success and failure. If a<b does not mean "if the operations to the right evaluate to true" instead it means "if operations to the right succeed". In this case < operator succeeds if the comparison is true.

10) Composite literals are sometimes known as aggregates. They are particularly valuable for the initialization of static data structures, without them, program may waste time performing initialization at run time. Ada provides aggregates for all its structured types.

11) Type equivalence is divided into structural equivalence and name equivalence. Structural equivalence in a broad sense means two types are the same if they consist of the same components put together in the same way. Name equivalence is based on lexical occurrence of type definitions.

In most cases it is enough that a value’s type be compatible with that of the context in which it appears (equivalence not required in most cases). Type coercion and type casting can be done to achieve type compatibility.

15) Type conversion or explicit type cast is done when we wish to use a value of one type in a context that expects another. It is done explicitly by us as and when required. Type conversion in C is specified by using the name of the desired type, in parentheses as a prefix operator.

Non converting type casts is done when one needs to change the type of a value without changing the underlying implementation. A common example is in high performance numeric software which needs to re-interpret a floating point number as an integer or record. A change of type that doesn’t alter the underlying bits is called non converting type casts.

Type coercion refers to the automatic and implicit conversion to the expected type is done. A coercion requires run-time code to perform dynamic semantic check.

19) Type inference refers to the automatic deduction of the data type of an expression in a programming language.

Inference of sub range types

Inference of Composite types.

23) Most languages allow a value to be assigned to an entire record in a single operation. For small records, both copies and comparisons can be performed in-line on a field by field basis. For longer records, we can save significant code space by deferring to a library routine. A block_copy routine can take source address, destination address, and length as arguments, but the analogous block_compare routine would fail on records with different (garbage) data in the holes. One solution is to arrange for all holes to contain some predictable value (e.g., zero), but this requires code at every elaboration point. Another is to have the compiler generate a customized field-by-field comparison routine for every record type. Different routines would be called to compare records of different types.

26) The two main purposes are:

a) Unions allow the same set of bytes to be interpreted in different ways at different times, sometimes as an unallocated space, sometimes as a bookkeeping information, sometimes as user allocated data of arbitrary type.

b) To represent alternative sets of fields within a record. A record representing an employee might have several common fields and various other fields. Variant records allow us to specify that certain sets of fields should overlap one another in memory.

33) Rather than require the rows of an array to be adjacent, they allow them to lie anywhere in memory, and create an auxiliary array of pointers to the rows. Row pointer layout requires more space than contiguous layout but has following advantages:

a) It sometimes allows the individual elements of the array to be accessed more quickly.
b) It allows rows to have different lengths, without devoting space to holes at the end of the rows (ragged array).
c) It allows a program to construct an array from pre-existing rows (scattered through memory) without copying.

The contiguous layout is a true multidimensional array, while the row-pointer layout is an array of pointers to array.

34) In row-major order, consecutive locations in memory hold elements that differ by one in the final subscript (except at the end of rows). A[2,4] is followed by A[2,5] . In column-major order, consecutive locations hold elements that differ by one in the *initial* subscript: A[2, 4] is followed by A[3, 4].

The difference between row- and column-major layout can be important for programs that use nested loops to access all the elements of a large, multidimensional array. On modern machines the speed of such loops is often limited by memory system performance, which depends heavily on the effectiveness of caching. When code traverses a small array, all or most of its elements are likely to remain in the cache through the end of the nested loops, and the orientation of cache lines will not matter. For a large array, however, lines that are accessed early in the traversal are likely to be evicted to make room for lines accessed later in the traversal. If array elements are accessed in order of consecutive addresses, then each miss will bring into the cache not only the desired element, but the next several elements as well. If elements are accessed *across* cache lines instead (i.e., along the rows of a Fortran array, or the columns of an array in most other languages), then there is a good chance that almost every access will result in a cache miss, dramatically reducing the performance of the code.

While column-major layout appears to offer no advantages on modern machines, its continued use in Fortran means that programmers must be aware of the underlying implementation in order to achieve good locality in nested loops. Row-pointer layout, likewise, has no performance advantage on modern machines (and a likely performance penalty, at least for numeric code), but it is a more natural fit for the "reference to object" data organization of languages like Java. Its impacts on space consumption and locality may be positive or negative, depending on the details of individual applications.

42) Garbage refers to objects, data of the memory of a computer system, which will not be used by any future computation by the system. A garbage is one to which no pointers exist. As computer systems have finite amounts of memory, it is frequently necessary to de allocate garbage and return it to the heap so the underlying memory can be reused.

Reference counts require a counter field in every heap object. For small objects such as cons cells, this space overhead may be significant. The ongoing expense of updating reference counts when pointers are changed can also be significant in a program with large amounts of pointer manipulation. Other garbage collection techniques, however, have similar overheads. Tracing generally requires a reversed pointer indicator in every heap block, which reference counting does not, and generational collectors must generally incur overhead on every pointer assignment in order to keep track of pointers into the newest section of the heap.

The two principal tradeoffs between reference counting and tracing are the inability of the former to handle cycles and the tendency of the latter to "stop the world" periodically in order to reclaim space. On the whole, implementors tend to favor reference counting for applications in which circularity is not an issue, and tracing collectors in the general case. The "stop the world" problem can be addressed with *incremental* or *concurrent* collectors, which interleave their execution with the rest of the program, but these tend to have higher total overhead. Efficient, effective garbage collection techniques remain an active area of research.

46) The three steps of **Mark and Sweep**

a) The collector walks through the heap, tentatively marking every block as "useless."
b) Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as "useful." (When it encounters a block that is already marked as "useful," the collector knows it has reached the block over some previous path, and returns without recursing.)
c) The collector again walks through the heap, moving every block that is still marked "useless" to the free list.

Stop and Copy

In a language with variable-size heap blocks, the garbage collector can reduce external fragmentation by performing storage compaction. Many garbage collectors employ a technique known as *stop-and-copy* that achieves compaction while simultaneously eliminating Steps 1 and 3 in the standard mark-and sweep algorithm. Specifically, they divide the heap into two regions of equal size.
All allocation happens in the first half. When this half is (nearly) full, the collector begins its exploration of reachable data structures. Each reachable block is copied into the second half of the heap, with no external fragmentation. The old version of the block, in the first half of the heap, is overwritten with a "useful" flag and a pointer to the new location. Any other pointer that refers to the same block (and is found later in the exploration) is set to point to the new location. When the collector finishes its exploration, all useful objects have been moved (and compacted) into the second half of the heap, and nothing in the first half is needed anymore. The collector can therefore swap its notion of first and second halves, and the program can continue. Obviously, this algorithm suffers from the fact that only half of the heap can be used at any given time, but in a system with virtual memory it is only the virtual space that is underutilized; each "half" of the heap can occupy most of physical memory as needed. Moreover, by eliminating Steps 1 and 3 of standard mark-and-sweep, stop-and-copy incurs overhead proportional to the number of non garbage blocks, rather than the total number of blocks.

Generational Collection:

To further reduce the cost of collection, some garbage collectors employ a "generational" technique, exploiting the observation that most dynamically allocated objects are short-lived. The heap is divided into multiple regions (often two).When space runs low the collector first examines the youngest region (the "nursery"), which it assumes is likely to have the highest proportion of garbage. Only if it is unable to reclaim sufficient space in this region does the collector examine the next-older region. To avoid leaking storage in long-running systems, the collector must be prepared, if necessary, to examine the entire heap. In most cases, however, the overhead of collection will be proportional to the size of the youngest region only.
Any object that survives some small number of collections (often one) in its current region is promoted (moved) to the next older region, in a manner reminiscent of stop-and-copy. Promotion requires, of course, that pointers from old objects.

Ex 7.6

There's no difference between A mod B and A rem B if A is nonnegative and B is positive. If A is negative and B is positive, mod gives you the true mathematical modulo operation; thus, for example, if B is 5, here are the results of A mod 5 and A rem 5 for values of A:

```
A       = -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8
A mod 5 =   0  1  2  3  4  0  1  2  3  4  0  1  2  3  4  0  1  2  3
A rem 5 =   0 -4 -3 -2 -1  0 -4 -3 -2 -1  0  1  2  3  4  0  1  2  3
```

Note the pattern in the A mod 5 results. rem corresponds to the way the % operator works in C-style languages (but not Python or Ruby, apparently). It may be faster on some processors. If you have to deal with negative values for A, my hunch is that mod is much more likely to be useful, but there may be some uses for rem also. I don't think there's much use at all for mod or rem with a negative right-hand operand, so I wouldn't worry too much about the definition.

Ex 7.49

Releasing unused objects for garbage collection can be done efficiently using java weak reference.

A weak reference, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.

Ex 7.15

1000 + (3*10 + 7)*8 = 1296

Considering 8-byte integer values.

Ex 7.20

```
double *a[n];          // n element array of pointers to doubles
double (*b)[n];        //  pointer to n element array of doubles
double (*c[n])();      //  n element array of pointers to functions returning doubles
double (*d())[n];      //  function returning pointer to array of n doubles
```

Ex 7.17

a)

```
    r2 *:= 40
    r3 <<:= 2       //  left shift by 2 multiplies by 4
    r1 +:= r2
    r1 +:= r3
    r1 := *r1
```

b)

```
    r2  <<:= 2   //  left shift by 2 multiplies by 4
    r1 +:= r2
    r1 := *r1
    r3 <<:= 2    //  left shift by 2 multiplies by 4
    r1 +:= r3
    r1 := *r1
```

The code sequence (a) is likely to be faster than (b), because it performs only one load instead of two and loads are very slow.

Control Flow and Abstraction

PLP: Ch #8

CYU

3) In a language with nested subroutines and static scoping, objects that lie in surrounding subroutines and that are neither local nor global can be found by maintaining a static chain.

Each stack frame contains a reference to the frame of the lexically surrounding subroutine. This reference is called the *static link*. By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the *dynamic link*. The static and dynamic links may or may not be the same, depending on whether the current routine was called by it's lexically surrounding routine, or by some other routine nested in that surrounding routine.

5) Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

6) A subroutine with no local variables and nothing to save or restore may not even need a stack frame on a RISC machine. The simplest subroutines (eg. library routines to compute the standard mathematical functions) may not touch memory at all, except to fetch instructions: they may take their arguments in registers, compute entirely in (caller-saves) registers, call no other routines, and return their results in registers. As a result they may be extremely fast.

8) Most of the tasks, however, can be performed either by the caller or the callee. In general, we will save space if the callee does as much work as possible: tasks performed in the callee appear only once in the target program, but tasks performed in the caller appear at every call site, and the typical subroutine is called in more than one place.

9) Most CISC instruction sets include push and pop instructions that combine a store or load with automatic update of the stack pointer. The push instruction, in particular, was traditionally used to pass arguments to subroutines, effectively allocating stack space on demand. The resulting instability in the value of the sp made it difficult (though not impossible) to use that register as the base for access to local variables. A separate frame pointer made code generation easier and, perhaps more important, made it practical to locate local variables from within a simple symbolic debugger.

11) The major difference between inline functions and macros is the way they are handled. Inline functions are parsed by the compiler, whereas macros are expanded by the C++ preprocessor.
Inline functions are like regular functions except that it writes a copy of the compiled function definition whereas macros are implemented with text replacement.

```
#define DOUBLE(X) X*X

 int y = 3;
 int j = DOUBLE(++y);
```

Because of the text replacement, what actually happens is that the DOUBLE(++y) expands to ++y * ++y, which equals 4*5, giving us 20. This problem would not occur if DOUBLE were implemented as an inline function. Inline functions only evaluate their arguments once, so any side effects of evaluation happen only once.

19) The object oriented features of C++ and its operator overloading, makes reference returns particularly useful. Without references << and >> would have to return a pointer to their stream. It is highly useful for function reference.

22) A default parameter is one that need not necessarily be provided by the caller, is it is missing, then a pre established default value will be used.

In any call in which actual parameters are missing, the compiler pretends as if the defaults had been provided; it generates a calling sequence that loads those defaults into registers or pushes them onto the stack, as appropriate.
On a 32-bit machine, put (37) will print the string “37” in an 11-column field (with nine leading blanks) in base-10 notation. Put(37, 4) will print “37” in
a four-column field (two leading blanks), and put(37, 4, 8) will print “45”
(37 = 458) in a four-column field.

24)
Within the body of a function with a variable-length argument list, the C or
C++ programmer must use a collection of standard routines to access the extra arguments. Like C and C++, C# and recent versions of Java support variable numbers of parameters, but unlike their parent languages they do so in a type-safe manner, by requiring all trailing parameters to share a common type.

27) The difference between macros and generics is much like the difference between macros and in-line subroutines generics are integrated into the rest of the language, and are understood by the compiler, rather than being tacked on as an afterthought, to be expanded by a preprocessor. Generic parameters are type checked. Arguments to generic subroutines are evaluated exactly once. Names declared inside generic code obey the normal scoping rules. In Ada, which allows nested subroutines and modules, names passed as generic arguments are resolved in the referencing environment in which the instance of the generic was created, but all other names in the generic are resolved in the environment in which the generic itself was declared.

30) In its pure form the table-based approach requires that the compiler have access to the entire program, or that the linker provide a mechanism to glue sub tables together. If code fragments are compiled independently, we can employ a

hybrid approach in which the compiler creates a separate table for each subroutine, and each stack frame contains a pointer to the appropriate table.

39) Setjmp takes as argument a buffer into which to capture a representation of the program’s current state. This buffer can later be passed to longjmp to restore the captured state. Setjmp has an integer return type: zero indicates “normal” return; nonzero indicates “return” from a longjmp. The usual programming idiom looks like this:

```
if (!setjmp(buffer)) {
/* protected code */
} else {
/* handler */
}
```

When initially called, setjmp returns a 0, and control enters the protected code. If longjmp(buffer, v) is called anywhere within the protected code, or in subroutines called by that code, then setjmp will appear to return again, this time with a return value of v, causing control to enter the handler. Setjmp and longjmp are usually implemented by saving the current machine registers in the setjmp buffer, and by restoring them in longjmp.

40) A volatile variable is one whose value in memory can change “spontaneously,” for example as the result of activity by an I/O device or a concurrent thread of control. C implementations are required to store volatile variables to memory whenever they are written, and to load them from memory whenever they are read. If a handler needs to see changes to a variable that may be modified by the protected code, then the programmer must include the volatile keyword in the variable’s declaration.

42) Coroutines are a form of sequential processing: only one is executing at any given time (just like subroutines AKA procedures AKA functions -- they just pass the baton among each other more fluidly.

Threads are (at least conceptually) a form of concurrent processing: multiple threads may be executing at any given time. (Traditionally, on single-CPU, single-core machines, that concurrency was simulated with some help from the OS -- nowadays, since so many machines are multi-CPU and/or multi-core).

46) A discrete event simulation is one in which the model is naturally expressed in terms of events (typically interactions among various interesting objects) that happen at specific times. Discrete event simulation is usually not appropriate for the continuous processes.

47) An *event* is something to which a running program (a process) needs to respond, but which occurs outside the program, at an unpredictable time. The most common events are inputs to a graphical user interface (GUI) system: keystrokes, mouse motions, button clicks. They may also be network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation.

Ex 8.14)

foo is a function taking a (pointer to a function taking a double and a pointer to a double that returns a double) and a double that returns a pointer to a function taking a double and anything that returns a double.

Recall the rule for C: start at the identifier; work as far right as possible, without escaping parentheses, then work as far left as possible without escaping parentheses, then move out one level and repeat.

Moving right from foo we find an argument list, so foo is a function, specifically a function of two parameters, neither of which is named. The first parameter to foo is a pointer to a function. For the sake of clarity, call that function F; if it had had a name the name would have appeared between the * and the following right parenthesis. F's first argument is a double; its second argument is an array of doubles, of unspecified length. The second parameter to foo is also a double.

Moving left from foo we find another *, so foo returns a pointer. To figure out the type of the thing to which it points, we move out a level of parentheses. Moving right from there we find another argument list, so foo returns a pointer to a function. That function, call it G, takes variable-length argument lists: the first parameter is a double; the others (zero or more of them) can be of any type. Finally, moving left, G returns a double. Because no function body is provided, this is a declaration of foo, but not a definition. The body must be provided elsewhere.

Ex 8.4)

In most language implementations, the activation records for different instances of foo will occupy the same space in the stack. Local variable i is never initialized (that's why the program is erroneous), but if the stack space has not been used for anything else in the meantime it may "inherit" a value from the previous instance of the routine. If the stack is created from space filled by zeros by the operating system, and if the space occupied by foo's activation record is not used for anything else before the first time foo is called, then i will start with the value zero in the first iteration of the loop.

Ex 8.15)

The optional parameters are not going to play any role in speeding the program as they are copied on to the stack.
Not with most machines and compilers. The code for a call with optional parameters is exactly the same as the code for a call in which all parameters are specified explicitly, with their default values. If one used a separate arguments pointer (as supported in hardware on the VAX), it might be possible, for non-recursive routines, to use a static build area, prepared at compile time, that contains pre-computed default values for optional parameters.

Ex 8.3)

```
#include <stdio.h>
static void foo(int a, int b, int c) {}
static int argument(int n)
{
   printf("Argument %d.\n", n);
   return n;
}
int main(void)
{
   foo(argument(0), argument(1), argument(2));
   return 0;
}
```

If you pass a function simple arguments, the compiler might evaluate them last-to-first so that it can easily put them on the stack in the order required for calling subroutines on that platform.

<u>Object oriented</u>

PLP: Ch #9

CYU:

1) Inheritance, Polymorphism and Encapsulation.

3) Abstraction is meant for removing code duplication and also for supporting polymorphism. Code reusability is an important advantage.

6) Private part is required for data hiding. It can be accessed only by a few members and so data is more secured.

7) Scope resolution operator is used when we have a local and global variable of the same name and we want to use the value of the global variable in a local scope.

10) Constructors are used for assigning the variables of an object with initial values. Destructors are used for destroying the object when it is no longer used.

14) The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Only member functions have a this pointer.

16) A public member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function.

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class would be private.

A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

20)
If a nested class in Java is declared to be static, it behaves as in C++ and C#, with access to only the static members of the surrounding class.

Java classes can also be nested inside methods. Such a *local* class not only has access to (all) members of the surrounding class; it also has (a copy of) any final parameters and variables of the method in which it is nested. Because final objects cannot be changed, copies are indistinguishable from the original, and the implementation does not need to maintain a reference (a static link) to the frame of the method in which the class is nested. Non-final objects are inaccessible to the local class.

Inner and local classes in Java are widely used to create object closures.

22) Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

23) The constructor allocates space for the object in stack. It is crucial that you always define a constructor for every class you define, and that the constructor initialize every data member of the class. If you don't define your own constructor, the compiler will automatically define one for you, and believe me, it won't do what you want. The data members will be initialized to random, unrepeatable values, and while your program may work anyway, it might not the next time you recompile. We should always make sure we delete what we allocate else it will lead to memory leak.

25) With a reference model for variables every object is created explicitly, and it is easy to ensure that an appropriate constructor is called.

The reference model of variables is arguably more elegant than the value model, particularly for object-oriented languages, but generally requires that objects be allocated from the heap, and imposes an extra level of indirection on every access.

28) Every object be initialized before it can be used.
If the object’s class (call it *B*) is derived from some other class (call it *A*), C++ insists on calling an *A* constructor before calling a *B* constructor, so that the derived class is guaranteed never to see its inherited fields in an inconsistent

state. When the programmer creates an object of class *B* (either via declaration or with a call to new), the creation operation specifies arguments for a *B* constructor. These arguments allow the C++ compiler to resolve overloading when multiple constructors exist.

Like C++, Java insists that a constructor for a base class be called before the constructor for a derived class. The syntax is a bit simpler, however; the initial line of the code for the derived class constructor may consist of a “call” to the base class constructor: super(*args*);
(C# has a similar mechanism.). Super is a Java keyword that refers to the base class of the class in whose code it appears. If the call to super is missing, the Java compiler automatically inserts a call to the base class’s zero-argument constructor (in which case such a constructor must exist).
Because Java uses a reference model uniformly for all objects, any class members that are themselves objects will actually be *references*, rather than “expanded” objects (to use the Eiffel term). Java simply initializes such members to null.
If the programmer wants something different, he or she must call new explicitly within the constructor of the surrounding class. Smalltalk and (in the common case) C# and Eiffel adopt a similar approach. In C#, members whose types are structs are initialized by setting all of their fields to zero or null. In Eiffel, if a class contains members of an expanded class type, that type is required to have
a single constructor, with no arguments; the Eiffel compiler arranges to call this constructor when the surrounding object is created.
Smalltalk, Eiffel, and CLOS are all more lax than C++ regarding the initialization of base classes. The compiler or interpreter arranges to call the constructor (creator, initializer) for each newly created object automatically, but it does *not* arrange to call constructors for base classes automatically; all it does is initialize base class data members to default (zero or null) values. If the derived class wants different behavior, its constructor(s) must call a constructor for the base class explicitly. Objective-C has no special notion of constructor: programmers must write and explicitly invoke their own initialization methods.

29) Initialization: A newly created object is first given a value. There is no previous value to overwrite (destroy).

Assignment: An existing object is given a different value (copied from another object). Previous value of the existing object is overwritten (destroyed).

```
void f()
{
```

```
int x = 1; /* Initialization. */
int y = x; /* Initialization. */
y = 2; /* Assignment. */
x = y; /* Assignment. */
y = g(x); /* Assignment (y = temp; temp is */
} /* a temporary on the stack frame of */
/* f(). See return statement below. */
int g( int a) /* Initialization ( int a = x). */
{
int t = a+1; /* Initialization. */
return 2*t; /* Initialization ( int temp =2* t, */
} /* where temp is as above. */
```

31) Here are few important difference between static and dynamic binding

a) Static binding in Java occurs during Compile time while Dynamic binding occurs during Runtime.

b) Private, final and static methods and variables uses static binding and bonded by compiler while virtual methods are bonded during runtime based upon runtime object.

c) Static binding uses Type (Class in Java) information for binding while Dynamic binding uses object to resolve binding.

d) Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

35) Dynamic method binding introduces polymorphism (specifically, *subtype* polymorphism) into any code that expects a reference to an object of some base class. The ability to use a derived class in a context that expects its base class is called subtype polymorphism.

40)
In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract. Abstract classes are used to provide an interface for its sub classes. Classes inheriting an abstract class must provide definition to the pure virtual function, otherwise they will also become abstract class.

43) Object closures can be used in an object-oriented language to achieve roughly the same effect as subroutine closures in a language with nested subroutines: namely, to encapsulate a method with *context* for later execution. It should be noted that this mechanism relies, for its full generality, on dynamic method binding.

Ex 9.14

C++ non-virtual methods are not dispatched, and do not override anything.

Java final methods are dispatched, and can override methods in their classes' super classes.

However, they are the similar in the respect that neither C++ non-virtual methods or Java final methods can be overridden. They are also similar in the sense that if you have some object whose static type is the type in question, the runtime system does not **need** to dispatch the method call.

EX 9.17

The representation of the bar object contain one b field. Base class data member b is overridden in derived class.
To prevent overriding of base class data members by the derived class, declare them as virtual.

Ex 9.21

Abstract class cannot be instantiated but pointers and references of abstract class type can be created.
Abstract classes cannot be instantiated, means we can't create an object to Abstract class. We can create Subclasses to Abstract classes.

An Abstract class may or may not have abstract methods, abstract method in the sense a method can declared without any body implementation is called abstract method. So in that case JVM does not know how much memory it has to allocate for that abstract method because abstract method does not have body implementation. So JVM will not able to allocate memory for the abstract methods when the time of creating instance to Abstract class. So JVM unable to create the instance to Abstract class. So that we can't create the object for the Abstract class.

It is also possible that we can create an Abstract class with all concrete methods, that is without any abstract methods also. In that case also we can't create the instance for the Abstract class. Why because the abstract keyword simply indicates to JVM that the class cannot be instantiated.