Homework 2

Shreyas Jayant Havaldar
CS18BTECH11042
This document is generated by LaTeX

CS2433: PRINCIPLES OF PROGRAMMING LANGUAGES-II
IIT Hyderabad

February 24, 2020

Ch 8: Subroutines and Control Abstraction

Check Your Understanding

Question 3

Describe how to maintain the static chain during a subroutine call.
Static chain is maintained by the caller by computing the callee's static link and passing it as a hidden parameter as it depends on the lexical nesting depth of the caller. This way the reference to the lexically surrounding subroutine structure is maintained. The callee might nest directly inside the caller, the static link being made the caller's frame OR the callee scopes outwards, thus the caller dereferences it's own static link depending on the callee's scope and passes it as the required static link.

Question 5

What are the purposes of the stack pointer and frame pointer registers? Why does a subroutine often need both?
Stack Pointer: Points to the top of the stack.
Frame Pointer: Points to the bottom of the stack.
A subroutine often needs both because a stack pointer provides us with the first unused location where we can insert a new element whereas frame pointer provides us with the ability to access objects on stack by using the offset from itself. Frame pointer provides us with debugging abilities. As the size of the stack is variable, we need both of them when a subroutine is exited to ensure the stack is empty and the stack pointer points to the same location as the frame pointer.

Question 6

Why do RISC machines typically pass subroutine parameters in registers rather than on the stack?
RISC machines are designed such to have simpler subroutines that require accesses to the memory only to fetch instructions usually. Thus these subroutines have local variables, which when saved in the register are much faster to operate upon. Using of the stack to pass arguments is both unsafe due to the variability of the stack pointer and much slower.

Question 8

If work can be done in either the caller or the callee, why do we typically prefer to do it in the callee?
We prefer to do all the work in callee to put to best use abstraction. Doing all the work in the caller leads to unnecessary overhead of dealing with all the details of the task at hand which destroys the utility of abstraction.

Question 9

Why do compilers typically allocate space for arguments in the stack, even when they pass them in registers?
Space is allocated in the stack as no special method is required when memory is allocated to arguments of stack similar to other paramaters to handle them. A stack has much more memory storage capacity than the limited number of registers. Thus it is always preferred.

Question 11

How does an in-line subroutine differ from a macro?
The way the compiler approaches both of them is inherently different. A compiler parses the inline function whereas a macro is simply replaced or expanded with its definition whenever it is called by the preprocessor. Thus type issues or compilation errors might creep in and not get recognized when using macros but are always detected when using inline functions. A copy of the called routine keeps on getting added to the caller function which might lead to large overheads when used for bigger subroutines. This issue is not faced while using macros.

Question 19

Give an example in which it is useful to return a reference from a function in C++.
It might be useful to return references from a function in C++ to deal with I/O operations. For eg: >> and << operators return reference to their first argument.

Question 22

What are default parameters? How are they implemented?
The caller doesn't need to pass a default parameter during a subroutine call, a pre-established default value is used. They are implemented by the compiler by pretending that they have been actually provided and are loaded as arguments to the method into the stack or the register by generating a calling sequence.

Question 24

Explain the value of variable-length argument lists. What distinguishes such lists in Java and C# from their counterparts in C and C++?
In C or C++ we must use a collection of standard routines to access the extra arguments, to be written in the body of the function. Nowadays the necessary support is mostly built into the compiler. In C and Java they are passed in a type-safe manner, by requiring all trailing parameters to share a common type unlike C and C++ which are type unsafe.

Question 27

How does a generic subroutine differ from a macro?
Execution time and memory requirement is much higher for macros.
Macros also lack a return statement, which is provided by a subroutine.
Subroutines provide us with the ability to pass arguments are receive a return value but unlike macros cannot be used to define run time interface parameters.

Question 30

What does it mean for a generic parameter to be constrained? Explain the difference between explicit and implicit constraints.
If a generic parameter is constrained, it strives to provide all information that might be required by an user of abstraction through its interface. They ensure that operations allowed on a generic parameter are explicitly declared.
Implicit constraints imply that the generic parameter is treated as a form of overloading.
Explicit constraints require generic parameters to be instantiated explicitly before they are used.

Question 35

Explain how to implement exceptions in a way that incurs no cost in the common case(when exceptions don't arise).
The usual way of maintaining a linked-list stack of handlers is improved upon by capturing a correspondence between handlers and the protected blocks by generating a table at compile time. It is sorted on the starting address of the block of code and also maintains the address of the corresponding handler. If an exception occurs at run time a binary search is performed to find the handler. The cost in common case is 0 thus, though we encounter a overhead when an exception occurs.

Question 39

Summarize the shortcomings of the setjmp and longjmp library routines of C.
setjump(jmp_buf buf) uses buf to remember current position and returns 0. and then *longjump(jmp_buf buf, i)* Restores current position and returns i. Zero indicates "normal" return; nonzero indicates "return" from a longjmp. The shortcoming is that the register contents at the beginning of the handler do not reflect the effects of the successfully completed portion of the code that is protected, code began to run before they were saved. Changes that were cached in registers will be lost, only the ones written through memory as visible to the handler.

Question 40

What is a volatile variable in C? Under what circumstances is it useful?
A variable whose value in memory changes spontaneously due to I/O or concurrent thread execution is a volatile variable. They are useful when we expect the handler to see the changes to the variable that might be modified by the code which is protected.

Question 42

What is the difference between a co-routine and a thread?
Co-routine: We have the control over switching of the co-routines and tasks can be multitasked cooperatively by pausing and resuming at set points even within single threads. They just follow sequential processing, only one coroutine executes at an instant.
Thread: The Operating System has the control over switching of the threads preemptively depending on its scheduling algorithm in the kernel. They follow concurrent processing, multiple threads may execute at the same time.

Question 46

What is discrete event simulation? What is its connection with coroutines?
It is a simulation in which the model is innately expressed in terms of events happening at specific times. Co-routines are used to simulate discrete events as we need to express interaction between different objects and each entity's process is represented by a co-routine.

Question 47

What is an event in the programming language sense of the word?
An event can be defined as something that occurs outside the program unpredictably, but a process needs to respond to. eg: Input to a GUI like *keypress*, *mouseclick* etc.

Exercises

Question 8.3

Using your favorite language and compiler, write a program that can tell the

order in which certain subroutine parameters are evaluated.

```cpp
//test.cpp
#include <iostream>
#include <stdlib.h>

using namespace std;

int one() {
    cout << "Parameter 1 evaluated \n";
    return 0;
}
int two() {
    cout << "Parameter 2 evaluated \n";
    return 0;
}
int three() {
    cout << "Parameter 3 evaluated \n";
    return 0;
}
int four() {
    cout << "Parameter 4 evaluated \n";
    return 0;
}

void func(int a, int b, int c, int d) {
}

int main() {
    func(one(), two(), three(), four());
    return 0;
}
// The program results in different values depending on whether the order was
   left to right or right to left
```

Commands executed:
g++ test.cpp
./a.out
OUTPUT:
Parameter 4 evaluated
Parameter 3 evaluated
Parameter 2 evaluated
Parameter 1 evaluated

I obtained consistent results for **right to left evaluation** for multiple executions using compiler: *g++ 7.4.0.*

Question 8.4

Consider the following (erroneous) program in C:

```c
void foo() {
```

```
int i;
printf("%d ", i++);
}
int main() {
int j;
for (j = 1; j <= 10; j++) foo();
}
}
```

Local variable i in subroutine foo is never initialized. On many systems, however, the program will display repeatable behavior, printing 0 1 2 3 4 5 6 7 8 9 . Suggest an explanation. Also explain why the behavior on other systems might be different, or nondeterministic.

On most systems, stack is created by initializing the space in stack with 0, and if the same space is not used for anything else beforehand, then i will be initialized to 0 and print 0 1 2 3 4 5 6 7 8 9 as the activation records for different instances of *foo*() will occupy the same space in stack for most of the languages.

But systems where space in stack is not initialized with 0 might lead to other outputs which cannot be uniformly determined. Also different instances of *foo*() might even have different space which could be another source of non-determinism.

Another source of error might be the compiler removing the variable i's value from the stack thus printing 0 0 0 0 0 0 0 0 0 0.

Question 8.8

Consider the following subroutine in Fortran 77:

```
subroutine shift (a, b, c )
integer a, b, c
a=b
b=c
end
```

Suppose we want to call shift(x, y, 0) but we don't want to change the value of y. Knowing that built-up expressions are passed as temporaries, we decide to call shift(x, y+0, 0). Our code works fine at first, but then (with some compilers) fails when we enable optimization. What is going on? What might we do instead?

Some compilers employ optimization such that the parameter *y+0* is reduced to simply y, thus the parameter is no more a built up expression and thus passed as *shift(x,y,0)* which defeats the purpose and leads to a modification in the value of y.

We should instead call *shift(x,y,y)* as then y is assigned to x, and y is assigned to y thus not affecting the original value of y.

Question 8.14

Consider the following declaration in C:

```
double(*foo(double (*)(double, double[]), double)) (double, ...);
```

Describe in English the type of foo.
foo is a function which takes a pointer to a function taking a double and a pointer to double returning a double and a double returning a pointer to a function taking a double and any function that is returning a double.

Question 8.15

Does a program run faster when the programmer leaves optional parameters out of a subroutine call? Why or why not?
No. The program doesn't speed up because implicitly the optional parameters are assigned their default values all the same. Irrespective of whether the optional parameters are included in a subroutine call or not, the paramters are loaded into the subroutine.

Question 8.29

Describe a plausible implementation of C++ destructors or Java try . . . finally blocks. What code must the compiler generate, at what points in the program, to ensure that cleanup always occurs when leaving a scope?
A plausible implementation of destructors in C++ or equivalently try ... finally in Java is to provide, for each scope a specified chunk of code while compiling to perform all the cleanup operations and then jump a pre-determined specific address in the program to continue forward.
The compiler must generate memory reclamation code to ensure that if a subroutine exits all its member methods and data members do not waste memory. It must generate this cleanup code at points in program where the program control jumps out of the subroutine to a new location altogether, eg: return statements. It must also pass the next address to jump to after the cleanup is performed and stack frame is emptied for that method.

Ch 9: Data Abstraction and Object Orientation

Check Your Understanding

Question 1

What are generally considered to be the three defining characteristics of object-oriented programming?
Encapsulation: Enables the programmer to group data and the subroutines that operate on the data together in one place, and to hide the irrelevant details from the users of an abstraction
Inheritance: Acquiring the properties of another class to use for oneself.
Dynamic Method Binding: Determination of methods to be used is dependent on the object rather than the variables.

Question 3

Name three important benefits of abstraction
The programmer doesn't need to ponder about every fine details together therefore reducing the conceptual load.
By limiting the scope of individual programming components, bugs are restricted thus providing fault containment.
Program components can be worked upon and manipulated independently without significantly affecting unrelated external components, thus establishing independence.

Question 6

What is the purpose of the "private" part of an object interface? Why is it required?
All the information that is not required by the programmer to use the object correctly, or the information required for compiler to generate executable code is put into the private part of the object interface, especially in programming languages that use a value model of variables, instead of a reference model. It is required for the compiler to generate code to allocate space to hold an instance of a class as it must be aware of the size of that instance and also to expand any in-line method calls primarily expansion of the smallest, most frequently used methods of an object-oriented program contributes largely towards the overall performance of the program.

Question 7

What is the purpose of the :: operator in C++?
:: is a scope resolution operator with the main purpose of helping identify the class to which the method belongs, by using it in the header.
Used to access global variables when there exist local variables of the same name.
Often used in the declaration of the derived class to refer to the base class or classes.
Also used to refer to the namespace in the program.
To refer to the data members of a specific class.

Question 10

What are constructors and destructors?
When an object of the class is created the constructor is implicitly called. The constructor is used to initialize the data members of the class. There can be 0, 1, or multiple constructors differentiated by their number and type of arguments.
Destructors facilitate manual storage reclamation, especially in absence of automatic garbage collection. They greatly help in storage management and error checking. Their main purpose is to deallocate the resources used by the object.

Question 14

Explain the significance of the *this* parameter in object-oriented languages.
The *this* keyword points to the instance or the object we are currently referring to. It helps prevent repetition of code for every instance of the module. A single instance of each subroutine is instead created and the address stored and passed as the hidden parameter *this*.

Question 16

Explain the distinctions among private, protected, and public class members in C++.
Public: Visible everywhere the class declaration is in scope.
Protected: Visible everywhere in the methods of that class itself, and all its descendants.
Private: Visible only in the methods of the class itself.

Question 20

How do inner classes in Java differ from most other nested classes?
In C family of languages, inner classes are allowed to only access the static members of the outer class, because they have only a single instance. Nesting purpose here is simply as a means of information hiding.
Java allows inner classes to access any arbitrary member of their outer class. Thus each and every instance of the inner class belongs to an instance of the outer class. Multiple instances of the outer class still leads to correct nesting as each inner class contains a hidden pointer unless it's declared to be static, in which case it behaves like C++ inner classes. Java permits inner classes to be nested within methods as well where it additionally has the access to variables of it's outer method.

Question 22

What are extension methods in C#? What purpose do they serve?
When we need to implement extension of functionalities, but inheritance is not feasible because of various reasons like restrictions on base class or preexisting chunks of code which utilise the same class name we can use extension methods in C. They do not support dynamic method binding or private member access but are a way around to invoke the method as if it were a member of the class *this* refers to. The extension method must be static. Example follows:

```
static class A {
    public static int convert(this string str) {
        return int.Parse(str);
    }
}
```

Question 23

Does a constructor allocate space for an object? Explain
No. A constructor does not allocate space. The role of the constructor is to initialize the data members automatically at the beginning of the object's lifetime, in the space that has already been allocated. It does not

Question 25

Why is object initialization simpler in a language with a reference model of variables (as opposed to a value model)?
In a reference model, every object must be created explicitly, and thus we can ensure that the correct constructor is called. In a value model, then object creation may occur implicitly as a result of elaboration. Thus the language must either allow uninitialized objects to begin their lifetime, or we must be provided with a way to choose an appropriate constructor for every elaborated object to prevent incorrect calls. Thus a reference model is much simpler and less error prone.

Question 28

Summarize the rules in C++ that determine the order in which constructors are called for a class, its base class(es), and the classes of its fields. How are these rules simplified in other languages?
C++: The constructors for any base classes will be executed, outermost first, before the constructor for the derived class. Also the rules follow that, if a class has members that are themselves objects of some class, then the constructors for the members will be called before the constructor for the object in which they are contained. Thus C++ requires every object to be initialized before it is used. Thus the derived class never sees the inherited fields inconsistently. Other languages have simpler rules.
Java: Also requires that a constructor for a base class be called before the constructor for a derived class simplified by using the keyword *super*. If we want something else, we must call new explicitly within the constructor of the surrounding class.
C: Members of type struct are initialized by setting all of their fields null.
Eiffel: For a class containing members of an expanded class type, that type is required to have a single constructor, with no arguments to call this constructor when the surrounding object is created.
All these languages simplify the rules by simply arranging calls to the constructor for each newly created object automatically, but not to call constructors for base classes automatically. They just initialize base class data members null.

Question 29

Explain the difference between initialization and assignment in C++
In the context of Object Oriented Programming, while using objects whether the object is declared by assignment or is initialized first has impact on the performance of the program.

Direct assignment leads to the additional cost of creating a temporary object to be the target of the operation involved in the assignment. Thus the eventual code includes the excessive zero argument constructor and destructor for the temporary object and a copy constructor to move the temporary constructor into the required object. Initializing first and then assigning value does not cause creation of a temporary constructor and is much faster.

Question 31

Explain the difference between dynamic and static method binding (i.e., between virtual and nonvirtual methods).
The decision of whether a member method to be called depends on the type of variable or the class of the object being referred to is applied by the type of method binding. Static method binding resolves overloaded methods during compile time, whereas dynamic method binding does the same at run time.

Question 35

Explain the connection between dynamic method binding and polymorphism.
Dynamic method binding is nothing but runtime ploymorphism. Dynamic method binding is central to the concept of polymorphism as it allows definitions in derived class to override the ones in the base class, thus allowing member methods to be truly used for multiple purposes.

Question 40

What is an abstract (deferred) class?
A class is said to be abstract if has one or more abstract member methods(virtual methods in C++). Abstract classes are used as a base class for concrete classes to derive from them and define every abstract method that it has inherited from the abstract class. They enable us to write code to call methods of objects of base class assuming the concrete methods are invoked at run time.

Question 43

Explain the importance of virtual methods for object closures.
Object closures are used to encapsulate a method with context for later execution relying completely on dynamic method binding. As the virtual functions are member functions in base class that are redefined in the derived class, they help us achieve this closure.

Exercises

Question 9.14

Compare Java final methods with C++ nonvirtual methods. How are they the same? How are they different?

The end-goal of both of these is to prevent overriding of a function in the base class. But they differ in their approach towards solving this problem. Java explicitly disallows declaration of methods with the same signature in which the base class has declared that method final. This is permitted in C++. It just helps recognise the correct function to call when there are confusion due to inheritance. They are not dispatched and do not override the clashing members, which is polar opposite of the approach Java takes.

Question 9.17

What happens to the implementation of a class if we redefine a data member? For example, suppose we have:

```
class foo {
public:
int a;
char *b;
};
...
class bar : public foo {
public:
float c;
int b;
};
```

Does the representation of a bar object contain one b field or two? If two, are both accessible, or only one? Under what circumstances?
The representation of the bar object has 2 b fields, but only one is accessible. Which one is accessible is determined by whether the language follows static or dynamic data member binding.

Question 9.21

If foo is an abstract class in a C++ program, why is it acceptable to declare variables of type foo*, but not of type foo ?
If we declare a *foo* we must initialize or instantiate it explicitly. If we declare a *foo it is acceptable because, we can use it to point to instances of classes that inherit from foo but are not abstract and thus can be implicitly instantiated.