POPL II A : JAN20 : Homework2

Raj Patil : CS18BTECH11039

This document was written in LaTeX

Chapter 8: CHECK YOUR UNDERSTANDING

3>

This is maintained by the caller (as this work depends on the lexical nesting of the caller). The standard approach is for the caller to compute the callee's static chain links and pass it as a hidden parameter to the callee. This leads to 2 cases: if the callee is nested inside the caller, the static chain of the two is same. Otherwise if the callee is k scopes outward, the caller dereferences k times outward and passes the obtained link as the callee's static link.

5>

Both stack and frame pointers are necessary for a process due to the following. frame pointers store the last frame's base address so as to conveniently collapse the stack and this also allows us to access the callee saved registers with a small negative offset from the frame pointer. Similarly, the stack pointer points to the end of the stack's scope of valid variables and arguments to called routines can be easily accessed with a small positive offset.

Note: offsets' signs are corresponding to when the stack grows downwards.

6>

A subroutine with no local variables may not need a stack frame on a RISC machine. They also don't need to write to the memory and only need to read instructions from there. Consequently, taking arguments as registers, rather than incurring the overhead of book-keeping results in them being extremely fast.

8>

Whenever, a work can be done by both caller and callee, we prefer the callee as we will save space if the callee does as much work as possible. This is so because the tasks performed in the callee appear only once in the program but that done in the caller appear at every call site resulting, which is not favourable as a regular function is called multiple times.

9>

This kind of operation is the reflection of technology of the 1970's when register sets were significantly smaller and memory access was significantly faster as compared to processor speed, which not quite the case today.

11>

In-line expansion of a subroutine is doesn't affect the program's meaning and moreover, Macros aren't type safe. Furthermore, the arguments that have side-effects are handled correctly in the case of in-lines before the function body is entered and this is not the case with macros. All this with similar performance improvements(though slightly less than macros, due to their rash nature).

19>

Returning references from functions in C++ is useful for this case(other than the convenience factor): Some objects do not support copies(eg: file buffers) resulting in them not being able to be passed or returned by value. Of course, one can use pointers, but references are better when one has to do this on multiple occasions, as dereferencing for each manipulation of the object can be tiring.

22>

A default isn't necessarily provided by the caller and if it is not, the callee uses a pre-assigned, default value. The implementation is fairly straightforward: in any call with missing parameters, the compiler pretends as if the defaults were provided.

24>

Variable length argument lists allow us to pass an indeterminate(at compile time) number of arguments to functions giving the programmer more freedom. As C,C++ are statically typed, additional parameters are not type safe and the programmer has to use a collection of standard routines to access the extra arguments. C# and Java also support the usage of variable length argument lists but they do so in a type safe manner as compared to their parent languages, enforcing all trailing parameters to share a common type.

27>

The difference between macros and generic subroutines is similar to that between macros and inline subroutines(see question no. 11). Macros aren't type safe and their side-effects may not be handled properly as they don't obey scopes, naming and type rules that apply to generic subroutines(that's why templates are created for).

30>

The generic being an abstraction, the interface concerning its usage must provide all the information that must be known by the user of the generic. This is enforced in the design of several languages by constraining the parameters: to be more specific, the require the operations permitted on that generic to be explicitly declared.

Implicit constraints are employed by some languages (for instance C++ and modula3) where they do not enforce explicit definitions of the generic but still check how the parameters are used.

39>

The problem with setjmp and longjmp is that they are implemented by saving the current registers in the setjmp buffer and by restoring them in longjmp. There is no handlers' list and the stack growth is ignored rather than the usual unwinding that happens in normal exception handling mechanisms. This results in the problem that the register contents at the beginning of the handler do not reflect the effects of successfully completed part of the protected code, they were just saved before the code was run. This also results in the dirtying of the cache which has to be addressed by the usage of the volatile keyword in front of that variable.

40>

The volatile keyword is used to notify the C compiler that a variable may change spontaneously in the main memory and consequently instructs the compiler to flush the cache entries for that variable after each read and write. In this case, if a handler needs to see changes to a variable that may be modified by the protected code, then the programmer must specify the volatile keyword in that variables declaration.

42>

Coroutines are a form of sequential processing: only one executes at a time(like subroutines) but they switch between them more fluently compared to a common subroutine(the stack grows and collapses).

Threads are a form of concurrent processing: multiple threads can run at a time.

46>

Simulation of discretizable real-world events is discrete event simulation. It is used when one wants to test the behaviour of a program in response to real events but doing so is dangerous (example: failure control during a flight's systems mishap). This is one of the most important application of coroutines (via event-based programming) which allows to create an abstraction and experiment with it safely for finding loopholes.

Discrete because it can't be used with continuous happenings such as the growth of a crystal or studying fluid behaviour(unless you get down to the level of a particle).

47>

Events (w.r.t. the programming esoterics) refers to an asynchronous activity that is detected by the program (maybe I/O or a request to a server by a client). Asynchronous being the key-word here : the user shouldn't be made to wait for an event to happen and then take action(synchronous behaviour), especially in the case when response times are important(GUI applications).

Chapter 8: EXERCISES

8.3>

consider the following C program:

 # include <stdio.h>

int main(){

```
    int a =10;
    printf("% d % d % d % d\ n",a++,++a,-a,a-);
    return 0;
}
```

the following output is received:

```
rajp152k@Raj:~ /links/sem_4/POPL2/A_POPL2/hw2/8ex$ gcc 3.c
rajp152k@Raj:~ /links/sem_4/POPL2/A_POPL2/hw2/8ex$ ./a.out
9 10 10 10
```

note that we did not see 10 first and a possible execution order begins with a– being evaluated first. The point is, the compiler chooses the order of evaluation in the case of C and C++ and its not always fixed(like from left to right).

8.4>

Repeatable behaviour maybe observed on most systems as the stack grows and collapses on the same set of virtual addresses leading to the actual increment being performed on the same address explaining the constant difference between the outputs. As for the first output being 0, this may just be a compiler preference which initializes integers to 0 and then doesn't reinitialize it again detecting that the last time that address was in the stack, it was already an int(possible explanation). This being a compiler's criterion to decide, may result in different results on different machines dependent on the compiler being used or in fact, the architecture itself.

8.8>

The reason it works without code optimizations is simple: the reference/value object in the function refers to a temporary and not the object itself resulting in the newly referred temporary being assigned the 0 and not the y. But in the case of compiler optimizations, the new temporary may not be created as one can see

that the result of the argument (y+0) is the same and recalculating it would be wasteful, resulting in y being changed, regardless.

a very simple workaround is to call shift(x,x,y)

this will keep the value of y unchanged and x will hold the value of y as, in the function definition, c(the third argument) occurs in the second statement and not the first one.

resulting in the following execution order:

x=x

x=y

with the value of y being unchanged.

8.15>

Yes, it does run faster, as the parameters are by default initialized by the the default arguments and when we pass optional parameters, an overhead of reassigning the variable in the local scope is incurred(talking in terms of loads and stores to the main memory).

8.29>

talking about C++ destructors:

one possible implementation could be:

- check if the object was passed by reference to that subroutine, in that case, do not destroy

- if it was passed by value, destroy the object

- if it was created in the local scope and one doesn't intend to return it, destroy

- if it was created in the local scope and one intends to return it, return a zero-constructed object of the same class with some sort of further exception handling mechanism so as to be type safe.

1>

The characteristics fundamental to object oriented programming are: encapsulation, inheritance and dynamic binding method.

3>

The important benefits of abstraction are structured as follows:

a) the user doesn't have to put in much technical thought behind the implementation and the concepts are handled by the abstraction mechanisms one at a time. One builds a smaller abstraction and then builds upon that for a more complex object, breaking down the thought in steps.

b) easy fault finding due to the bug being easily observed to be isolated in a contained section.

c) independence among program components : promotes reusability because one code segment is easily adaptable to other and is an opportunity to handle boiler-plate efficiently.

6>

Private members are used to restrain access to an attribute (or some subsidiary procedures) of a class to only certain specific publicly accessible(global scope) procedures. They are useful as they restrict the ways in one can modify that entity hence making errors back trackable which is useful during exception handling(unwinding the stack).

7>

The :: is a scope resolution operator. It aids in recognizing what context does an identifier belong to (to be specific.. by specifying a namespace).

10>

A constructor is invoked upon the creation of a new object and it is simply a an initialization routine defined by the programmer. The programmer can also explicitly specify a destructor that is invoked automatically upon the destruction of an object. However, it is not needed to be defined and a default is anyway called when the object

under concern goes out of scope.

14>

The "this" keyword is useful when one decides to use externally defined modules in the class definition rather than placing that module code in the class definition itself. The latter approach is ignorant and not recommended as they would just be address translated execution procedures and instead the this keyword allows in accessing the attributes that fall under the scope of the object to which the module will be attached.

16>

Private members can only be accessed by the class in which they are defined and friend class objects.

Protected members are accessible in the class that defines them, ones derived from that class, and friend classes as well.

Public members are accessible by anyone.

20>

In languages like C++ and C# , only static members of the outer class are allowed to access the class members as the corresponding inner class has only one static instance. However, Java takes it up a notch and allows an inner class to access any members of its encompassing class; each instance of the inner class belongs to the outer class and if there are many instances to an outer class, each one of them will have its own inner class that can access its members.

22>

Extension methods can be perceived as class(or even pertaining to a specific instance if you wish) extensors.

They (from C# 3.0 onwards) are used to extend the functionality of an abstraction when inheritance is not an option. This maybe due to several reasons such as the class being sealed(C# terminology) or the case when it will be very cumbersome to change all the variables that occur in the class definition when inheritance is possible.

23>

No, a Constructor does not allocate the space for an object. It just initializes the already allocated space i.e. the space signature is already allocated, but the contents can be set according to the programmer's needs via defining a suitable constructor (a class definition can have multiple constructors).

25>

When working with references, the constructor has to be explicitly called because a reference in itself is of no significance without the binding and this allows us to check easily whether the constructor was called or not. But in a value based model, the variables start out either unitialized or initialized to a default(0 for ints in C, for instance) and this implicit nature makes it difficult to keep track of whether the desired initialization has happened or not.

28>

When dealing with inheritance, the constructor of the base class is called before the constructor of the derived class so that the latter does not see any discrepancies in its inherited members. However, to correctly call the constructor for the base class, one needs to provide the constructor's signature which would be the violation of the notion of abstraction that object orientation desires to maintain. C++ does this by allowing the header of the derived class to specify the base class's constructor's arguments.

C++ asks for this explicitly, but in the case of other languages, say Java, if the programmer wishes to achieve a higher level of abstraction by not passing the signature, the zero-constructor for the base class is called by default(possible as Java uses a reference model and can simply initialize a reference to null). If the programmer does want to specify the specifics for the base constructor, it can be done using new.

29>

In the simplest terms, when initializing, only a constructor(the one defined in the class definition) is called. When assigning, the constructor(the one defined in the class definition) is called to create a temporary and then a copy-constructor(usually automat-

ically defined by the compiler itself) is called to copy the contents from the temporary to the object being assigned. Note that assignment can also be done using a move-constructor and this is usually the case when dealing with references that always have to point to something, whereas failure to do so might result in undefined behaviour. For instance thread objects and mutexes (from C++11 and onwards) have move-constructors defined which are invoked when assigning them a new thread of execution or a low-level synchronization primitive respectively.

31>

Note that one can easily observe by the nomenclature itself that static refers to compile-time or (early) binding and dynamic refers to run-time or late binding.

A difference to note that dynamic binds require an entity(an object or something that can be referenced to) to exist at the time of binding and hence need to established during run-times. This is not the case with static binding. This is a notion most commonly discussed when discussing about the mechanisms of Java, so one could consider this example: static binds work for type class definitions and dynamic binds work for object-class definitions.

Also note that, obviously, static binds are faster than dynamic binds(as in establishing the bind).

In regards to virtual/non-virtual methods, in languages that use static binding by default (eg: C++), one can define a dynamically bound method by using the virtual keyword (the counterpart being a non-virtual(real) method).

35>

When we talk about dynamic method binding, what is given is subtype polymorphism: the ability to use a derived class in the context where the base class is expected (assuming public permissions to public members of the base class). This allows us to define a function for the base class and pass on a derived class at run time, for instance.

40>

When the class definition under concern contains at least one pure

virtual function, it is said to be an abstract class. This is because, no instances can be created with that class definition only and one needs those virtual methods to be bound to a defined function in a derived class. So only instances of the derived classes will make sense. The name is pretty intuitive, the class definition is abstract and not definitive, it changes depending on which derived class one looks at.

43>

object closures for the case of abstract classes can be only be achieved by the use of dynamic binds, and hence virtual methods. They allow the class under concern to refer the virtual method to the correct context for later execution.

Chapter 9: Exercises

9.14>

C++ non-virtual functions can still be over-ridden by writing a definition for the same signature in some derived class but that is not the case when using the final methods in Java: they cannot be overridden. Java clearly forbids the declaration of functions in derived classes with the same signature as a final keyworded method.

9.17>

No, one can't override base class members, the compiler simply won't allow this and a compilation error will be raised. If one needs to have one variable represent multiple data types, unions can be used. But one is rarely so desperate to make the same variable to mean different things and this is not good practice - should be avoided if possible

9.21>

foo being an abstract class, it is possible to declare foo pointers but not instances because a foo pointer can point to one of its derived class instances where the virtual-function is bound to a definitive method resulting in allowed method. But there is no use case where

declaring a foo instance would make sense, it is just pointless.