GOOD LUCK EVERYONE!

Red: Yet to be Answered
Green: Not on the Test

Chapter 7 (10 Questions)

- What are unary, binary and ternary (or tertiary) arithmetic expressions?
    - **Unary operators:**
        - Only require one operand in order to perform the task or operation.
            - Ex. +, -, ++, --
                - + and - are used to show the sign of a number
    - **Binary operators:**
        - Require two operands in order to perform the task or operation.
            - Ex. +, -, *, /, %
    - **Ternary operators:**
        - Require three operands to perform the operation.
            - Ex. ?, &, :
                - These are conditional operands.
                - condition?expression1:expression2; (Source)
- What are infix, postfix, and prefix arithmetic expressions?
    - **Prefix:**
        - Arithmetic operator is before 2 operands
            - Ex. + A B
    - **Infix:**
        - Arithmetic operator is between 2 operands
            - Ex. A * B
    - **Postfix:**
        - Arithmetic operator is after 2 operands
            - Ex. A B +

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

- What are the design issues related to arithmetic expressions?
    - Operator precedence rules
    - Operator associativity rules
    - Order of operand evaluation
    - Operand evaluation side effects
    - Operator overloading
    - Type mixing in expressions
- When can associativity affect a computation?
    - Operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.
        - Associativity may affect a computation when it behaves differently in one language and another.
            - In APL, associativity is right to left, unlike typical left to right. (Sebesta Chapter 7 Slides)
- What is the ternary conditional operator?
    - Has three operands (condition?expression1:expression2;)
- What is a side-effect? What evaluation commentary can you offer about them?
    - Functional Side Effects: when a function changes a two-way parameter or a non-local variable.
    - Two options to solve the problem of operator evaluation order and side effects:
        - The language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects
        - Language definition could state that operands in expressions are to be evaluated in a particular order and demand implementers guarantee that order.
- What are the advantages and disadvantages of operator overloading?
    - **Advantage**: readability if operators were used sensibly. (Avoid method calls, expressions appear natural)
    - **Disadvantage**:
        - Users can define nonsense operations and readability may suffer, even when the operators make sense
        - Loss of compiler error detection (omission of an operand should be a detectable error)
- What do "narrowing" and "widening" mean in type conversions? Is one better than the other? Always?
    - **Narrowing**: one that converts an object to a type that cannot include all of the values of the original type EX. float to int
    - **Widening**: One in which an object is converted to a type that can include at least approximations to all of the values of the original type EX. int to float

    - Widening is safer than narrowing given that in the process of narrowing certain values cannot be accurately converted since the magnitude of the value is changed in the process. But it is not always safe since some precision may be lost.
    - **example:** integer-to-floating-point widening can result in the loss of two digits of precision since integers allow for a total of 9 decimal digits vs float that has a total of 7 decimal digits
- What does it mean to say that type "coercion" takes place?
    - Coercion is an implicit type conversion that is initiated by the compiler or runtime system.
    - When two operands of an operator are not of the same type (and that is legal in the language), the compiler must choose one of them to be coerced and generate the code for that coercion.
    - Disadvantage: they decrease in the type error detection ability of the compiler
- What is a mixed-mode expression?
    - Expressions that have operands of different types
- Which typically have higher precedence, arithmetic or relational operators?
    - Arithmetic operators typically have higher precedence. One should note however that operators with equal precedence operate left or right associativity based on the rules of the language. For example a = b = c would be parsed as a = (b = c) and similarly a + b - c would be parsed as (a+b) - c
- What is short-circuit evaluation? Is it commonly used in languages? When is it potentially a problem?
    - An expression in which the result is determined without evaluating all of the operands and/or operators
    - C, C++, and Java use short-circuit evaluation for the usual boolean operators.
    - All logic operators in Ruby, Perl, ML, F#, and Python are short circuit evaluated.
- What are compound and unary assignments?
    - **Compound:** A shorthand method of specifying a commonly needed form of assignment. Introduced in ALGOL: adopted by C and the C-based languages
        - EX. a=a+b can be written as a+=b
    - **Unary:** Combine increment and decrement operations with assignment in C-based languages.
        - sum = ++count (count increment, then assigned to sum)
        - sum = count++ (count assigned to sum, then incremented)
        - count ++ (count incremented)
        - -count++ (count incremented then negated)
- Why would you use an assignment as an expression (give an example) and why can this be bad?
    - In C based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

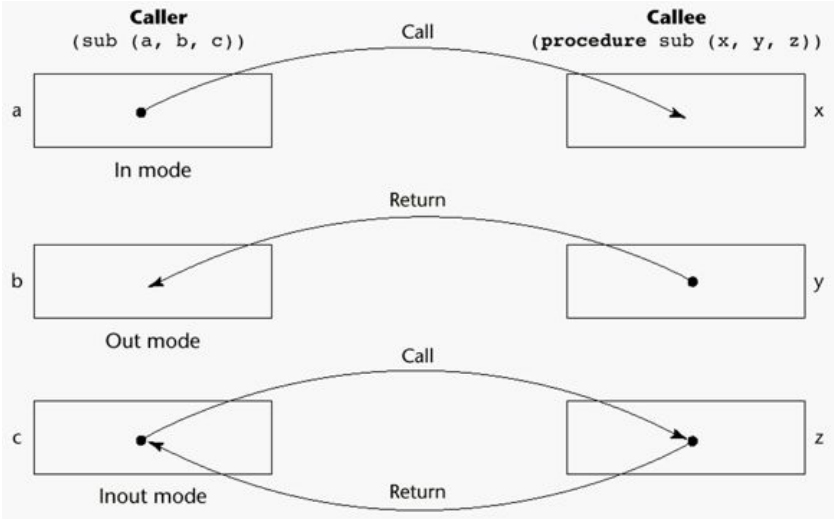
        - while ((ch = getchar()) != EOF) {...}
        - ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement
    - Disadvantage: The disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect. This type of side effect can lead to expressions that are difficult to read and under- stand. An expression with any kind of side effect has this disadvantage. Such an expression cannot be read as an expression, which in mathematics is a denota- tion of a value, but only as a list of instructions with an odd order of execution.

Chapter 9 (~7 Questions)

- What are the general characteristics of subprograms
    - A subprogram has a single entry point
    - A *subprogram call* is an explicit request that the subprogram be executed
        - The calling program is suspended during execution of the called subprogram
    - Control always returns to the caller when the called subprogram's execution terminates
- Distinguish a subprogram definition (including what is a prototype or header) from a call
    - A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
    - A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters. This is the **interface** to the rest of the system
    - A *subprogram declaration* provides the protocol, but not the body, of the subprogram.
    - The *parameter profile*(aka *signature*) of a subprogram is the number, order, and types of its parameters
    - The *protocol* is a subprogram's parameter profile and, if it is a function, its return type.
- What is the difference between a formal parameter and an actual parameter?
    - **formal parameter** — the identifier used in a method to stand for the value that is passed into the method by a caller
    - **actual parameter** — the actual value that is passed into the method by a caller. (Source)
- What does it mean to say that most languages use "positional parameters"?
    - The binding of actual parameters to formal parameters is done by position: The first actual parameter is bound to the first formal parameter and so forth.
        - Safe and effective

- What are keyword parameters? What are the restrictions associated with them?
    - Keyword parameters means that the name of a formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.
        - Advantage: Parameters can appear in any order, thereby avoiding parameter correspondence errors
        - Disadvantage: User must know the formal parameter's name
    - The only restriction with this approach is that after a keyword parameter appears in the list, all remaining parameters must be keyworded. This restriction is necessary because a position may no longer be well defined after a keyword parameter has appeared
- How do most languages handle the problem of a variable number of actual parameters?
    - Overloaded operators (I think the answer she is looking for is default parameters)
    - Default parameters
        - In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
        - In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)
    - Variable numbers of parameters
        - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **params**
        - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
        - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
        - In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a **for** statement or with a multiple assignment from the three periods
- What is the distinction between a procedure and a function and why does it matter?
    - **Procedures:** don't return a value. They are invoked solely for their side-effects: printing something, changing some data structure
        - Are collections of statements that define parameterized computations
        - Liken to providing the language with a new type of statement.
    - **Functions:** return a value. Are likened to user-defined operators
        - Structurally resemble procedures but are semantically modeled on mathematical functions
        - They are expected to produce no side effects
        - In practice, program functions have side effects
- What are the design issues for subprograms in section 9.3
    - Are local variables statically or dynamically allocated?
        - Local variables can be either static or stack dynamic.
    - Can subprogram definitions appear in other subprogram definitions?
    - What parameter-passing method or methods are used?

    - Are the types of the actual parameters checked against the types of the formal parameter?
    - If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
    - Are functional side effects allowed?
        - No, because of the problems of side effects of functions that are called in expressions as described in chapter 5, parameters to a function must always be in-mode.
    - What types of values can be returned from functions?
        - Most imperative languages restrict the type that can be returned by their function. C allows any type except arrays and functions. C++ is like C but it allows user-defined types or classes to be returned. Python and Ruby are the only languages among the imperatives that are able to return values of any type
    - Can subprograms be overloaded?
        - Yes, Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism
    - Can subprograms be generic?
        - Yes, Parametrically polymorphic subprograms are often called generic subprograms (C++, Java 5.0 +, C# 2005 +, F# provide a kind of compile-time parametric polymorphism)
    - If the language allows nested subprograms, are closures supported?
        - There are some languages that allow nested subprograms that do not support closures. Nearly all functional languages, most scripting languages and at least 1 primarily imperative language (C#) that support closures. But if a language does not allow nested subprogram then they will definitely not support closures because they do not need them
- (Anywhere the review says "what are the design issues" you should be able to discuss them and typical approaches)
- What are the two most common kinds of storage binding for local variables in subprograms?
    - Stack-Dynamic
        - **Advantages**: Support for recursion. Storage for locals is shared among some subprograms
        - **Disadvantages**: Allocation/ Deallocation, initialization time. Indirect addressing. Subprograms cannot be history sensitive.
    - Static
        - **Advantages and Disadvantages** are opposite of those for stack-dynamic local variables
- Which one of these supports recursion and why?

    - Stack-Dynamic storage binding supports recursion, because storage bindings are created for variables when their declaration statements are executed at runtime
- When would you want to use the other kind?
    - When you need history sensitivity in a subprogram
- Know the meaning of the three semantic models of parameter passing: in, out, and in-out
    - 

- Know the details of implementation for pass-by-value, pass-by-result, pass-by-value-result and pass-by-reference
    - **Pass-by-value (in Mode)**: The value of the actual parameter is used to initialize the corresponding formal parameter
        - Normally implemented by copying
        - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
            - **Disadvantages (if by physical move):** additional storage is required (stored twice) and the actual move can be costly (for large parameters)
            - **Disadvantages (if by access path method):** must write-protect in the called subprogram and accesses cost more (indirect addressing)
    - **Pass-by-Result (Out Mode)**: When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
        - Requires extra storage location and copy operation
        - Potential Problems:
            - sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1.
            - sub(list[sub], sub); Compute address of list[sub] at the beginning of the subprogram or end?

    - **Pass-by-Value-Result (InOut Mode):** A combination of pass-by-value and pass-by-result.
        - Sometimes called *pass-by-copy*. Formal Parameters have local storage
        - **Disadvantages:** Those of pass-by-result and pass-by-value
    - **Pass-by-Reference (InOut Mode):** Pass an access path.
        - Also called *pass-by-sharing*
        - **Advantage:** Passing process is efficient (no copying and no duplicated storage)
        - **Disadvantages:** Slower access (compared to pass-by-value) to formal parameters
            - Potential for unwanted side effects(collisions)
            - Unwanted aliases (access broadened)
- What are the costs and advantages associated with each of these?
    - **Pass-by-Value (In Mode):**
        - Disadvantages (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
        - Disadvantages (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)
    - **Pass-by-Result (Out Mode):**
        - Potential problems:
            - sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1
            - sub(list[sub], sub); Compute address of list[sub] at the beginning of the subprogram or end?
    - **Pass-by-Value-Result (Inout Mode):**
        - Disadvantages:
            - Those of pass-by-result
            - Those of pass-by-value
- Know that pass-by-name is essentially the same as using a macro definition
    - **Pass-by-Name (InOut Mode):** By textual substitution
        - Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
        - Allows Flexibility in late binding
        - Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated.
- What are the details of doing overloading of subprograms?
    - An **Overloaded Subprogram** is one that has the same name as another subprogram in the same referencing environment
        - Every version of an overloaded subprogram has a unique protocol
    - C++, Java, C#, and Ada include predefined overloaded programs

    - In Ada, the return type of an overloaded function can be used to disambiguate calls(thus two overloaded functions can have the same parameters)
    - Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name.
- What are the details and constraints on using default parameters in subprograms?
- Describe the problem of passing multidimensional arrays as parameters.
    - If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function
        - Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
        - Disallows writing flexible subprograms
    - Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters
- Sections 9.7 and 9.8 may come up indirectly in questions related to C++

Chapter 11 Review Questions (~7 Questions)

- What are two kinds of abstractions in programming languages?
    - Two kinds of abstractions in programming languages are process abstraction and data abstraction.
- Define the concept of an ADT.
    - Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations. (Source)
- What purpose does the Ada **use** clause have? Why does Java not need a **use** clause?
    - The use type clause makes both infix and prefix operators visible without the need for renames clauses. You enhance readability with the use type clause because you can write statements using the more natural infix notation for operators. (Source)
    - Part 2 still missing
- From where are C++ objects allocated?
    - For object allocation C++ has three storage types:
        - Automatic Storage: It is also known as stack memory
        - Static Storage: It is usable for namespace scope objects and local statics
        - Free Storage Heap: It is used for dynamically allocated objects.
- In what different places can the definition of a C++ member function appear?
    - You can place them inline, it could also be outside or inside the definition of the class
- What is the purpose of a C++ constructor?

    - A constructor in C++ is a special member function and C++ allows the user to include it in the class definition. It initializes the object of the class and has the same name as the class. Whenever the object of the class is created, constructor is automatically invoked, It constructs the data member value of the class, hence it is called a constructor.
- Where are all Java methods defined?
    - All Java methods must be defined in a class. A Java method can be overloaded and overridden. If overriding occurs then the Java compiler cannot incline methods.If we make a method private then it is hidden from the client.
- Why does Java not have destructors?
    - In Java, the garbage collector automatically deletes the unused objects to free up the memory. Developers have no need to mark the objects for deletion, which is error-prone and vulnerable to the memory leak. So it's sensible Java has no destructors available (Source)
- What is a friend function? Why does Java not have them?
    - A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
    - Java has packages which bring some friend functionality
    - Java (mostly) discards non-accessible members. If friendship needs to be taken into account then the resolution is more complicated and less obvious.(Source)
- Explain the dangers of C's approach to encapsulation.
    - The main problem is that the biggest part of encapsulation is done via hiding, rather than protection. This is achieved through definition hiding: a header file is preprocessed (which is a synonym for copy-pasted) into the implementation file. Anyone with this header file will be able to access any method or public variable of the client related to the header, leaving apart any "static" method / variable. (Source)
- What are arguments for and against the C++ policy on inling of methods?
    - **Advantages :** It does not require function calling overhead.
        - It also saves overhead of variables push/pop on the stack, while function calling.
        - It also saves overhead of return calls from a function.
        - It increases locality of reference by utilizing instruction cache.
        - After in-lining compilers can also apply interprocedural optimization if specified. This is the most important one, in this way compilers can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..
    - **Disadvantages** : May increase function size so that it may not fit on the cache, causing lots of cache misses.

        - After in-lining function if variables number which are going to use the register increases then they may create overhead on register variable resource utilization.
        - It may cause compilation overhead as if some body changes code inside an inline function then all calling locations will also be compiled.
        - If used in a header file, it will make your header file size large and may also make it unreadable.
        - If somebody used too many inline functions resultant in a larger code size than it may cause thrashing in memory. More and more page faults bringing down your program performance.
        - It's not useful for embedded systems where large binary size is not preferred at all due to memory size constraints. (Source)

Chapter 12 Review Questions (~4 Questions)

- Define the three characteristic features of object-oriented languages.
    - Encapsulation, Inheritance and Polymorphism.
- What is the difference between a class variable and an instance variable?
    - The difference between instance and class variables is Class variables only have one copy that is shared by all the different objects of a class, whereas every object has its own personal copy of an instance variable. So, instance variables across different objects can have different values whereas class variables across different objects can have only one value.
- Explain one disadvantage of inheritance.
    - Inherited functions work slower than normal function as there is indirection.
    - Improper use of inheritance may lead to wrong solutions.
    - Often, data members in the base class are left unused which may lead to memory wastage.
    - Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.
    - Creates interdependencies among classes that complicate maintenance
- What exactly does it mean for a subclass to have an is-a relationship with its parent class?
    - “Is-a” implies inheritance. The derived class has all of the public methods and properties of the parent class (plus some of its own). This is useful because other parts of the application can accept parameters of the parent class type, and since all derived classes “is-a” that type, you can pass any one of those to the method. It gives you polymorphism without having to write a separate (but virtually identical) method for every conceivable child type, or without using “object”, “void *”, etc., which throws type safety out the window. (Source)

    - “Derived class” IS-A child class of “Base Class”
- What is multiple inheritance?
    - An inheritance which allows a new class to inherit from two or more classes
        - Advantage: Sometimes convenient and valuable
        - Disadvantage: Language and implementation complexity(in part due to name collisions). Potential inefficiency-dynamic binding costs more with multiple inheritance
- What is an overriding method?
    - Overriding method is a method that purpose is to provide an operation in the subclass that is similar to one in the parent class, but is customized for objects of the subclass.
- Describe a situation where dynamic (late) binding is a great advantage over its absence.
    - There is a base class, ‘A’, that defines a method ‘draw’ that draws some figure associated with the base class. A second class, ‘B’, is defined as a subclass of ‘A’. Objects of this new class also need a ‘draw’ method that is like that provided by ‘A’ but a bit different. With overriding, we can directly modify ‘B’'s draw function. But without it, we either make a specific function in ‘A’ for ‘B’ and inherit it.
- What is a virtual method?
    - Virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature.
- What is an abstract method? What is an abstract class?
    - An abstract method is one that does not include a definition (it only defines a protocol).
    - An abstract class is one that includes at least one pure virtual method. An abstract class cannot be instantiated. Declare an interface without declaring a full set of implementations for that interface. The interface specifies the abstract operations supported by all objects derived from that class. It is up to the derived classes to supply implementations for those abstract operations. Its representation extends only to the features those classes of objects have in common
- Compare the class entity access controls of C++ and Java.
    - Java has packages, which are groups of classes, which can access each other’s default, public, and protected entities, similar to the friend function of C++.
    - Java also has nested classes that can be accessed by the class they’re nested in. C++ has private, public, and protected inheritance, as well as friend functions.
        - Private methods and variables can only be accessed within the class they’re in.
        - Protected methods and variables can be accessed by the class they’re in as well as child classes.

        - Public has the same access as protected and can be accessed outside as well.
        - Friend functions allow the access of private data and methods from classes that are friends of that class.

Fortran

- Be prepared to discuss the pros and cons of your experience programming in Fortran.

C++ (6-7 Questions)

- How does (or does not) C++ illustrate the preservation of readability in transitioning from one language to another?
- What about reliability and writability? Especially what about orthogonality (a partial measure of simplicity)
- What is the zero-overhead rule?
    - The zero-overhead principle is a guiding principle for the design of C++. It states that: *What you don't use, you don't pay for* (in time or space) and further: *What you do use, you couldn't hand code any better.*
    - In other words, no feature should be added to C++ which would make any existing code (not using the new feature) larger or slower, nor should any feature be added for which the compiler would generate code that is not as good as a programmer would create without using the feature.(Source)
- What is the "scope-resolution operator"?
    - The scope resolution operator is :: (2 colons)
    - It allows access to global variables
    - In C++, global variables are declared outside the main function, therefore they are out of scope in main. If a user wants access to global variables, use ::
    - Inside the main function, users can create local variables of the same name as global variables. The scope resolution operator also lets the user distinguish that they are using the global variable instead of local
- What is the most correct form of programmer-controlled primitive type coercion?
- What does the friend keyword do? Why is it useful? How does it break our usual assessment of readability, reliability, etc.?
    - A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

- What guarantee does C++ make regarding constructors? What does this commitment reflect about the history of the language?
    - Guarantees that a constructor will be called on class instantiation.
- What can be overloaded in C++?
    - Functions and operators
- Why are const reference parameters useful in C++?
    - With a constant reference as a parameter, you can pass by reference without giving the function write access. This allows the function to not have to spend resources copying the parameter's value onto its stack frame
- How are pointers different from references? When would you use one over the other?
    - Pointers contain the memory address of another variable. References contain the actual value of another variable. You would use a pointer when you want to pass by reference and a reference when you want to create an alias for a variable.
- Why use copy constructors (you can provide deep vs. shallow copy) but why do that?
    - A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype.
    - **Shallow:**
        - A shallow copy of an object copies all of the member field values.
    - **Deep:**
        - Copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. (Source)
- What does the "inline" keyword do?
    - The compiler essentially pastes the function code into the function call
- Explain what the virtual keyword does and why it is important
    - The virtual keyword is used to modify a method, property, indexer, or event declared in the base class and allow it to be overridden in the derived class
        - It is important because it allows for the user to still access the base class' function.
- What does it mean to make a member function "pure virtual"?
    - When the base class virtual function is never used, remove the body and set it.
        - EX. class Base{ public: virtual void show() = 0; }
        - The = sign has nothing to do with assignment and the value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a function will be pure, i.e., **will have no body**.