Chapter 5: Names, Bindings, Scopes

1. What are the design issues for names?
Length, **special characters**, **case sensitivity**, **special words**(reserved words, keywords)
(“Are the name case sensitive?”, “Are special words reversed for words or keywords?”)

3. In what way are reserved words better than keywords?
Keywords have a special meaning in a language, and are part of the syntax.
Reserved words are words that cannot be used as identifiers (variables, functions, etc.), because they are reserved by the language.
Reserved words are better than keywords because the redefinition of keywords can be confusing, but not on reserved words.

4. What is an alias?
Alias is when more than one variable name can be used to access the same memory location.

7. Define binding and binding time.
A binding is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. The time at which a binding takes place is called binding time.

8. After language design and implementation [what are the four times bindings can take place in a program?]
Bindings can take place at language design time, language implementation time, **compile time**, **load time**, **link time**, or **run time.**

9. Define static binding and dynamic binding.
A binding is static if it first occurs before run time and remains unchanged throughout program execution.
A binding is dynamic if it first occurs during execution or can change during execution of the program.

10. What are the advantages and disadvantages of implicit declarations?
Advantages: Simple in naming conventions. In this case, the compiler or interpreter binds a variable to a type based on the syntactic form of the variable’s name.

Disadvantages: Although they are a minor convenience to programmers, implicit declarations can be detrimental to reliability because they prevent the compilation process from detecting some typographical and programmer errors.

11. What are the advantages and disadvantages of dynamic type binding?

Advantages:

Dynamic type binding allows any variable to be assigned a value of any type.

It provides more programming flexibility.

Disadvantages:

It causes programs to be less reliable, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings.

The cost of implementing dynamic attribute binding is considerable, particularly in execution time.

12. Define static, stack-dynamic, explicit heap-dynamic, and implicit heap- dynamic variables. What are their advantages and disadvantages?

Static: bound to memory cells before execution begins and remains bound to the same memory cell throughout the execution.(for example, using **'static'** to define a variable in C, static int i = 1;)

Stack-dynamic: storage bindings are created for variables when their declaration statements are elaborated. (For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.)

Explicit heap-dynamic: allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution. (for example, using **'new'** to define a variable in JAVA, List l = new ArrayList<String>();)

Implicit heap-dynamic: Allocation and deallocation caused by assignment statements. (for example, highs = [74, 84, 86, 90, 71];)

	Advantage	Disadvantage
Static	1. efficiency (addressing can be direct) 2. no run-time overhead is incurred for allocation and deallocation.	1. reduced flexibility (a language that has only static variables cannot support recursion) 2. storage cannot be shared among variables.
Stack-dynamic	Useful in most cases, supports recursion.	Run-time overhead of allocation and deallcation.

Explicit heap-dynamic	Useful to create dynamic data structure, like: linked list, trees that need to grow or shrink during execution time	Difficulty of using pointer and reference variables correctly; the cost of references to the variables; the complexity of the required storage management implementation (heap management).
Implicit heap-dynamic	The highest degree of flexibility, allowing highly generic code to be written.	1. run-time overhead of maintaining all the dynamic attributes (array subscript types and ranges, etc.) 2. loss of some error detection by the compiler.

13. Define lifetime, scope, static scope, and dynamic scope.

Lifetime: the lifetime of a variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

Scope: the scope of a variable is the range of statements over which the variable is visible.

Static scope: static scope is binding names to non-local variables.

Dynamic scope: Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.

15. What is the general problem with static scoping?

Too much access. Scope structure destroyed as program evolves.

7. Assume the following JavaScript program was interpreted using static-scoping rules. What value of x is displayed in function sub1? Under dynamic-scoping rules, what value of x is displayed in function sub1?

```
var x;
function sub1() {
  document.write("x = " + x + "<br />");
}
function sub2() {
  var x;
  x = 10;
  sub1();
}
x = 5;
sub2();
```

By using **static-scoping rules**, **the value of x is 5.**

Because the search for x begins in the procedure in which the reference occurs, sub1. But no declaration for x is found there. The search continues in the static parent of sub1, where the declaration of x is found, x = 5.

By using **dynamic-scoping rules, the value of x is 10.** Because after the search of local declaration fails, the declarations of dynamic parent, or calling function, are searched. So we found x = 10 in sub1's dynamic parent function, sub2().

```
var x, y, z;
function sub1() {
  var a, y, z;
  function sub2() {
    var a, b, z;
    ...
  }
  ...
}
function sub3() {
  var a, x, w;
  ...
}
```

8. Consider the following JavaScript program

List all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used.

sub1: a(sub1), y(sub1), z(sub1), x(global)

sub2: a(sub2), b(sub2), z(sub2), y(sub1), x(global)

sub3: a(sub3), x(sub3), w(sub3), y(global), z(global)

11. Consider the following skeletal C program:

Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last function called? Include with each visible variable the name of the function in which it was defined.

```
void fun1(void);  /* prototype */
void fun2(void);  /* prototype */
void fun3(void);  /* prototype */
void main() {
  int a, b, c;
  ...
 }
void fun1(void) {
  int b, c, d;
  ...
 }
void fun2(void) {
  int c, d, e;
  ...
 }
void fun3(void) {
  ...
}
```

a. main calls fun1; fun1 calls fun2; fun2 calls fun3.

b. main calls fun1; fun1 calls fun3.

c. main calls fun2; fun2 calls fun3; fun3 calls fun1.

d. main calls fun3; fun3 calls fun1.

e.main calls fun1; fun1 calls fun3; fun3 calls fun2.

f.main calls fun3; fun3 calls fun2; fun2 calls fun1.

a

Variables	Declared in
d, e, f	fun3()
c	fun2()
b	fun1()
a	main()

b

Variables	Declared in
d, e, f	fun3()
b, c	fun1()
a	main()

c	
Variables	Declared in
b, c, d	fun1()
e, f	fun3()
a	main()

d	
Variables	Declared in
b, c, d	fun1()
e, f	fun3()
a	main()

e	
Variables	Declared in
c, d, e	fun2()
f	fun3()
b	fun1()
a	main()

f	
Variables	Declared in
b, c, d	fun1()
e	fun2()
f	fun3()
a	main()

Chapter 6: Data Types

3. What are the design issues for character string types?

Should strings be special type or primitive?

Should strings have static or dynamic length?

4. Describe the three string length options.

Static length: The length can be static and set when the string is created.

Limited dynamic length: allow strings to have varying length up to a declared and fixed maximum set by the variable's definition. may need a run-time descriptor for length (but not in C and C++).
Dynamic length: allow strings to have varying length with no maximum. need run-time descriptor; allocation/de-allocation is the biggest implementation problem.

9. Define static, fixed stack-dynamic, stack-dynamic, fixed heap-dynamic, and heap-dynamic arrays. What are the advantages of each?
Static: subscript ranges are statically bound and storage allocation is static (before run-time)
Advantage: efficiency (no dynamic allocation).

Fixed stack-dynamic: subscript ranges are statically bound, but the allocation is done at declaration time.
Advantage: space efficiency.

Stack-dynamic: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
Advantage: flexibility (the size of an array need not be known until the array is to be used).

Fixed heap-dynamic: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack).

Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times.
Advantage: flexibility (arrays can grow or shrink during program execution).

1. What are the arguments for and against representing Boolean values as single bits in memory?
For: Using a single bit instead of an entire byte will conserve memory

Against: Processors to have addresses for single bits. So up to 8 Booleans values can be combined into a single byte, which means bit-wise arithmetic to separate the separate values. Although this conserves memory, it takes more time (CPU time) to deal with.

7. What significant justification is there for the -> operator in C and C++?
Instead of writing as (*p).val, p->val is more concise. So it has better writability.

15. What are the arguments for and against Java's implicit heap storage recovery, when compared with the explicit heap storage recovery required in C++? Consider real-time systems.

For: Java's implicit heap storage recovery eliminates the dangling pointer problem. On the other hand, it also makes the Java programming easier. In C++, we have to use delete to manually release the resources we used. This is insecure because programmers may forget delete the resources then the trouble of dangling pointers will occur.

Against: Though Java's implicit heap storage recovery has many advantages, it doesn't work perfectly as we expected. Firstly, Java uses more memory to store the resources. Because Java's GC happens at an uncertain point, it has to store the useless resources for GC to release them. Also, sometimes, GC mechanism isn't efficient. Taking Generational Garbage Collection as an example. In a major GC, it often costs a lot of time.

21. In what way is static type checking better than dynamic type checking?

Firstly, static type checking happens at compile time, so it will cost less time in runtime for checking the type. **Secondly**, by using static type checking, some potential errors can be detected early. So there will be no fatal error in runtime.

Chapter 7: Expressions and Assignment Statements

7. Describe a situation in which the add operator in a programming language would not be commutative.

Take the program on the right side as an example.

The assignment statement x = x + fun(&x) is not commutative.

If we evaluate x first and then fun(&x), the result would be 7. However, If we calculate fun(&x) first and then evaluate the statement x + fun(&x), the result would be 12. Because in the function fun(int *i), the value of original x has been changed to 8. This is might not what we are expected.

```
int fun(int *i) {
  *i += 5;
  return 4;
 }
void main() {
  int x = 3;
  x = x + fun(&x);
 }
```

8. Describe a situation in which the add operator in a programming language would not be associative.

In JAVA, the maximum of Integer is 2^31 - 1. In the statement int v = a - b + c, if a, b, c are very large number that close to 2^31 - 1, we cannot calculate a + c first. Because it will result in overflow. However if we calculate a - b first, the whole express can be correctly computed.

20. Consider the following C program:
What is the value of x after the assignment statement in main, assuming
a. operands are evaluated left to right.
b. operands are evaluated right to left.

```
int fun(int *i) {
  *i += 5;
  return 4;
 }
void main() {
  int x = 3;
  x = x + fun(&x);
 }
```

a. 7
b. 12 (this is because in the function fun(&x) we already changed the value of x)

Chapter 8: Statement-Level Control Structures

1. Describe three situations where a combined counting and logical looping statement is needed.

a. A list of values is to be added to a SUM, but the loop is to be exited if SUM exceeds some prescribed value.

b. A list of values is to be read into an array, where the reading is to terminate when either a prescribed number of values have been read or some special value is found in the list.

c. The values stored in a linked list are to be moved to an array, where values are to be moved until the end of the linked list is found or the array is filled, whichever comes first.

4. What are the pros and cons of using unique closing reserved words on compound statements?
Advantage: Readability. When one sees an endif or endwhile in a program written by someone else, it is clear which block is ending.

Disadvantage: Writability. Complicating the language by increasing the number of keywords.

Chapter 9: Subprograms

5. Consider the following program written in C syntax:

```
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void main() {
  int value = 2, list[5] = {1, 3, 5, 7, 9};
  swap(value, list[0]);
  swap(list[0], list[1]);
  swap(value, list[value]);
}
```

For each of the following parameter-passing methods, what are all of the values of the variables value and list after each of the three calls to swap?

a. Passed by value
b. Passed by reference
c. Passed by value-result

a. the variables value and list didn't change after each of the swap function. value = 2, list = {1, 3, 5, 7, 9}
b. after the first swap: value = 1, list = {2, 3, 5, 7, 9}; after the second swap: value = 1, list = {3, 2, 5, 7, 9}; after the third swap: value = 2, list = {3, 1, 5, 7, 9}
c. after the first swap: value = 1, list = {2, 3, 5, 7, 9}; after the second swap: value = 1, list = {3, 2, 5, 7, 9}; So far, the result is as the same as if we pass the parameters by reference. However, in the third swap function, we have two different result.

1. If we pass back from left to right. Then value will be 2 and list[value] will be list[2]. So b will send back to list[2]. At last, value = 2, list = {3, 2, 1, 7, 9}

2. If we pass back from right to left. Then list[value] will be list[1]. So b will send back to list[1] and then a will send back to value. At last, value = 2, list = {3, 1, 5, 7, 9}

7. Consider the following program written in C syntax:

For each of the following parameter-passing methods, what are the values of the list array after execution?

a. Passed by value
b. Passed by reference
c. Passed by value-result

```
void fun (int first, int second) {
  first += first;
  second += second;
}
void main() {
  int list[2] = {1, 3};
  fun(list[0], list[1]);
}
```

a. list = {1, 3}
b. list = {2, 6}

c. list = {2, 6}

Eillustrates call by value, value-result, reference

```
begin
   integer n;
   procedure p(k: integer);
      begin
      n := n+1;
      k := k+4;
      print(n);
      end;
   n := 0;
   p(n);
   print(n);
   end;
Note that when using call by reference, n and k are aliased.
Output:

call by value:       1 1
call by value-result: 1 4
call by reference:    5 5
```

15. What are at least two arguments against the use of pass-by-name parameters?

First, Pass-by-name parameters are complex to implement and inefficient.
Also, it adds significant complexity to the problem, thereby lowering its readability and reliability.

Chapter 14: Exception Handling and Event Handling

1. What did the designers of C get in return for not requiring subscript range checking?

The designers of C choose not to check the subscript ranges because the cost of such checking was not worth the benefit of detecting the errors. However, in some kind of compiler, user can choose to turn on subscript range checking.