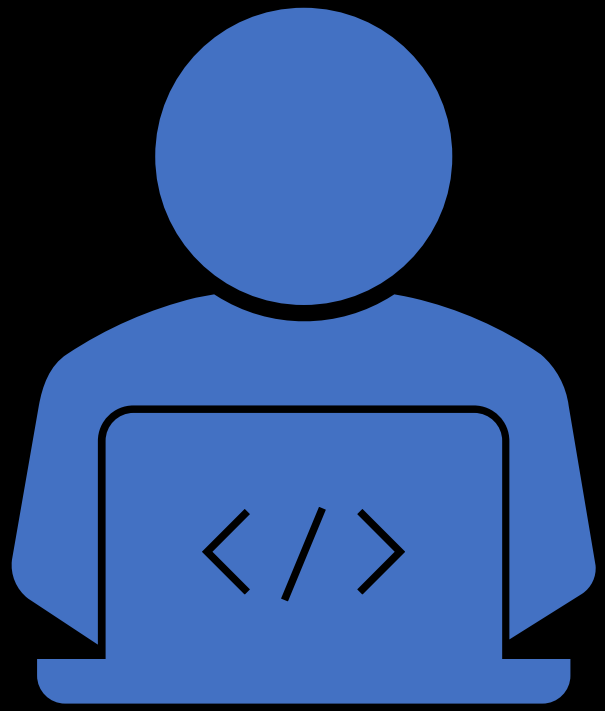


CENG204 - Programming Languages Concepts

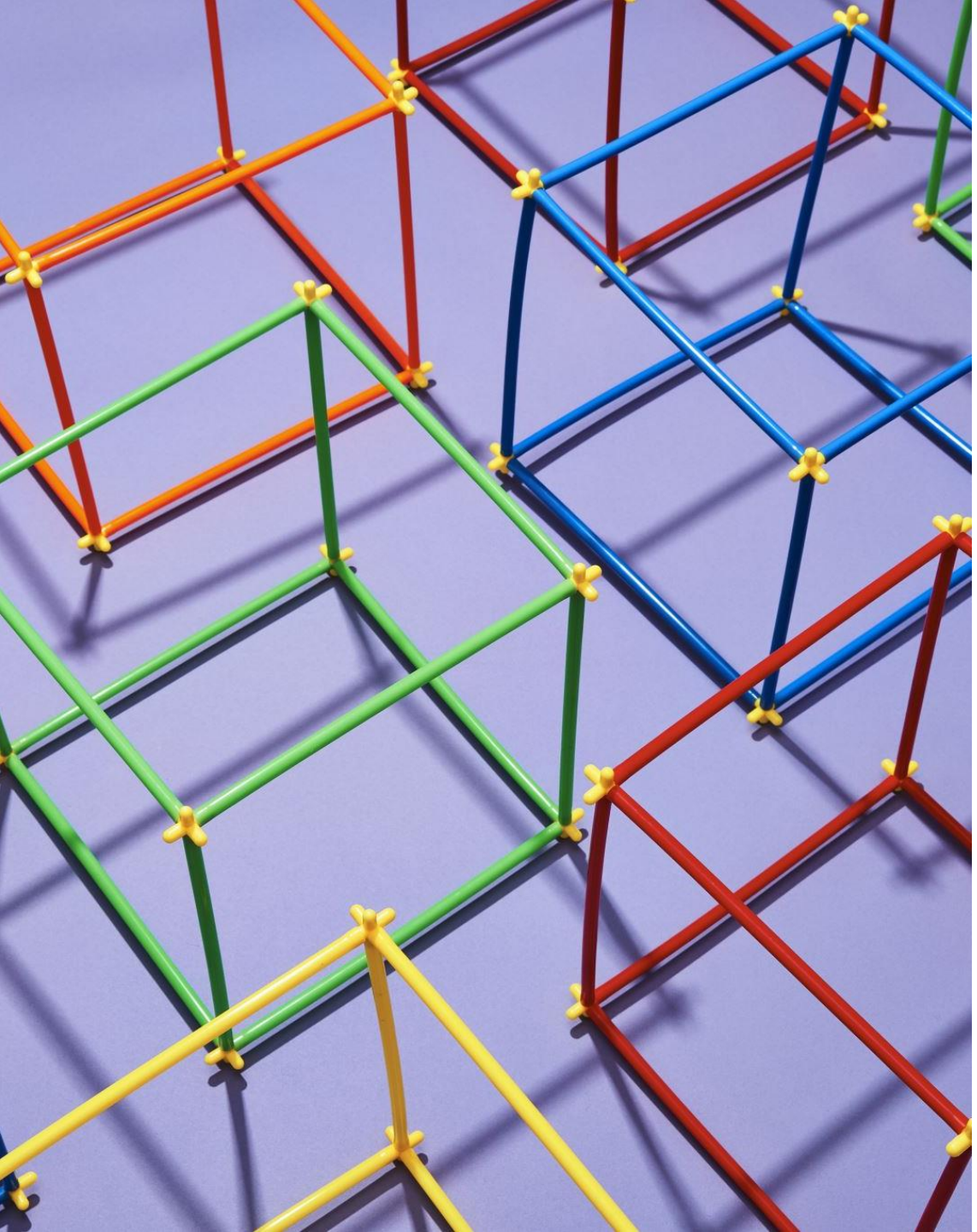
Asst. Prof. Dr. Emre ŞATIR

Lecture 9
Names, Bindings, and Scopes

Lecture 9 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Named Constants

Introduction



Introduction

- Imperative programming languages are, to varying degrees, abstractions of the underlying von Neumann computer architecture.
- The architecture's two primary components are its memory, which stores both instructions and data, and its processor, which provides operations for modifying the contents of the memory.
- The abstractions in a language for the memory cells of the machine are **variables**.

Introduction

- A **variable** can be characterized by a collection of properties, or attributes, the most important of which is **type**, a fundamental concept in programming languages (Data types are discussed later).
- Among the most important of these issues are the “lifetime” and “scope” of variables.
- Before beginning our discussion of variables, the design of one of the fundamental attributes of variables, **names**, must be covered.
- Names are also associated with subprograms, formal parameters, and other program constructs. The term “identifier” is often used interchangeably with name.

Names

Names

- A **name** is a string of characters used to identify some entity in a program.

Name Forms

- Some languages have limits for name lengths, but most of them not.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters (_).
- Some languages may impose restrictions such as “names cannot start with the following character(s)” or “names cannot include the following character(s) ”).

history note

The earliest programming languages used single-character names. This notation was natural because early programming was primarily mathematical, and mathematicians have long used single-character names for unknowns in their formal notations.

Fortran I broke with the tradition of the single-character name, allowing up to six characters in its names.

Python Variable Names: Do's and Don'ts

Do…	Don't…
Use meaningful names • first_name is much preferred over x	• Use single letter names • Use python reserved words (and, if, integer, float)
Use underscores to represent spaces: • total_score • overtime_hours	Use spaces: • The variable name "total score" will generate an error
Use lowercase letters for variable names • weekly_pay	Use capital letters….ever • The variable WeeklyPay is not "Pythonic"
Only use letters for variable names • full_name	Use special characters or numbers in variable names: • first&lastname • hours_over_40

Name Forms

- Although the use of underscore characters to form names was widely used in the 1970s and 1980s, that practice is now far less popular.
- In the C-based languages, it has to a large extent been replaced by the so-called “camel notation”, in which all of the words of a multiple-word name except the first are capitalized, as in `myStack` (It is called “camel” because words written in it often have embedded uppercase letters, which look like a camel’s humps).

*** Note that the use of underscores and mixed case in names is a programming style issue, not a “language design issue”.

Name Forms

- Is it ok to use a non-English, foreign language variable name (for example Turkish)?

  Example: `int sayaç;`

- It completely depends on what language you are programming in.
- For example, Java support it (mostly, as long as it's Unicode), however, C will not like it at all as it only supports ASCII.

Name Forms

- All variable names in PHP must begin with a dollar sign.
- In Perl, the special character at the beginning of a variable's name, $, @, or %, specifies its type.
- In Ruby, special characters at the beginning of a variable's name, @ or @@, indicate that the variable is an instance or a class variable, respectively.

Name Forms

- The following are “the primary design issues” for names:
  - Are names case sensitive?
  - Are the special words of the language reserved words or keywords?

Case Sensitivity

- In many languages, notably the C-based languages, uppercase and lowercase letters in names are distinct; that is, names in these languages are **case sensitive**.
- For example, the following three names are distinct in these languages: `rose`, `ROSE`, and `Rose`.
- To some people, this is a serious detriment to readability (a disadvantage), because names that look very similar in fact denote different entities.
- Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Special Words

- **Special words** in programming languages are used to make programs more readable by naming actions to be performed. (An aid to readability; used to delimit or separate statement clauses)
- In most languages, special words are classified as **reserved words**, which means they cannot be redefined by programmers. (A reserved word is a special word of a programming language that cannot be used as a name).
- But in some, such as Fortran, they are only **keywords**, which means they can be redefined. Of course, this is a factor that reduces readability.

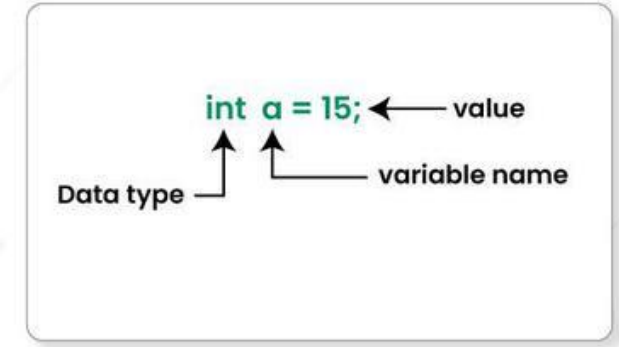
Special Words

- There is one potential problem with reserved words: If the language includes a large number of reserved words, the user may have difficulty making up names that are not reserved.
- The best example of this is COBOL, which has 300 reserved words. Unfortunately, some of the most commonly chosen names by programmers are in the list of reserved words—for example, `LENGTH`, `BOTTOM`, `DESTINATION`, and `COUNT`.

Variables

Variables

- A program **variable** is an abstraction of a computer memory cell or collection of cells.
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Type
  - Value
  - Lifetime
  - Scope



Variables Attributes - Name/Address

- Variable **names** are the most common names in programs. They were discussed before in the general context of entity names in programs.
- Most variables have names. The ones that do not (Heap-Dynamic Variables) are discussed later.
- The **address** of a variable is the machine memory address with which it is associated.
  - If two variable names can be used to access the same memory location, they are called ***aliases***.
  - Aliases are created via pointers and reference variables.
  - Aliases are harmful to “reliability” (must be careful when making changes) and “readability” (program readers must remember all of them).

Variables Attributes - Type

- The **type** of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type.
- For example, the `int` type in Java specifies a value range of −2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division, and modulus.
- In the case of floating point, type also determines the precision.

Variables Attributes - Value

- The **value** of a variable is the contents of the memory cell or cells associated with the variable.
  - The **l-value** of a variable is its address.
  - The **r-value** of a variable is its value.
- A variable's value is sometimes called its *r-value* because it is what is required when the name of the variable appears in the right side of an assignment statement.
- To access the *r-value*, the *l-value* must be determined first.

The Concept of Binding

The Concept of Binding

- 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**.
- Bindings can take place at;
  - language design time,
  - compile time,
  - or execution time (run time).

The Concept of Binding

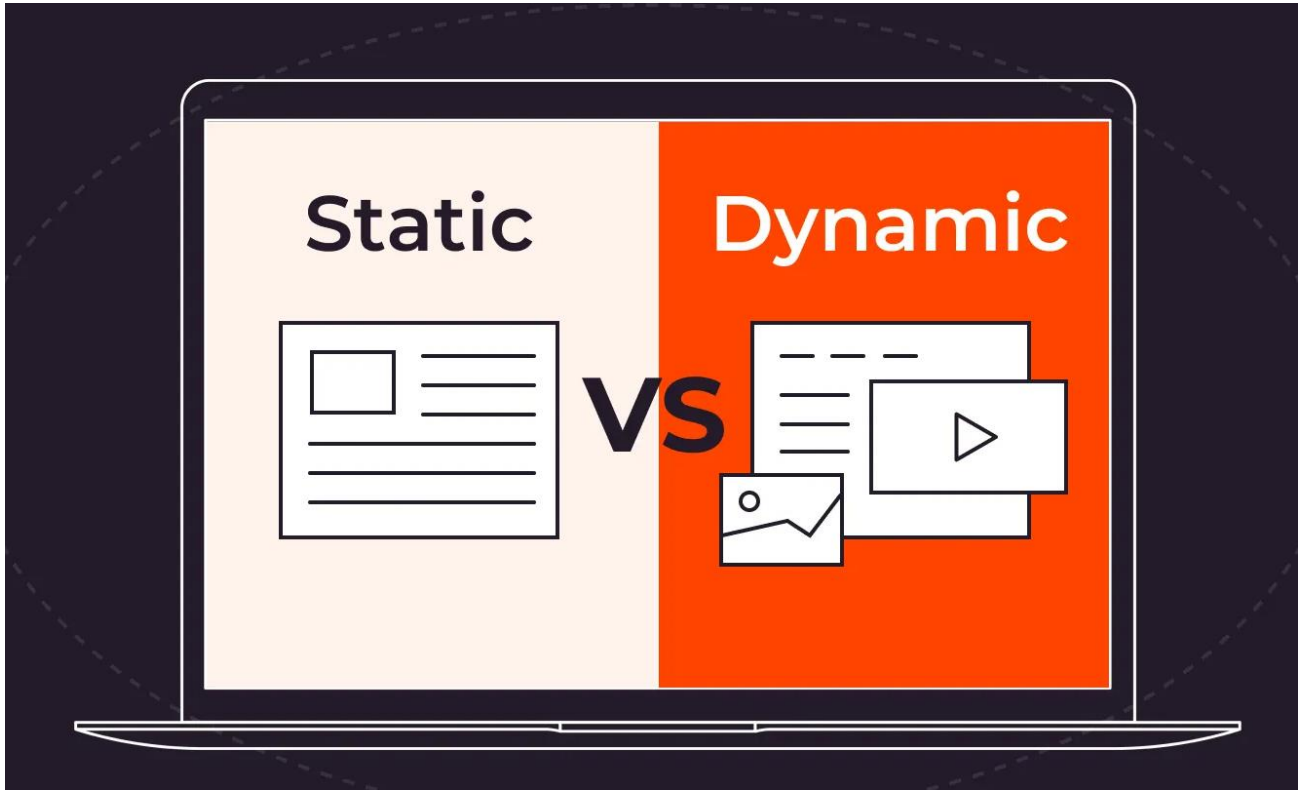
- Consider the following C statements:

```
int count;
scanf("%d", count);
count = count * 5;
```

- The asterisk symbol (*) is bound to the multiplication operation at language design time.
- The type of count is bound at compile time.
- The value of count is bound at execution time (run time).

The Concept of Binding

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



The Concept of Binding

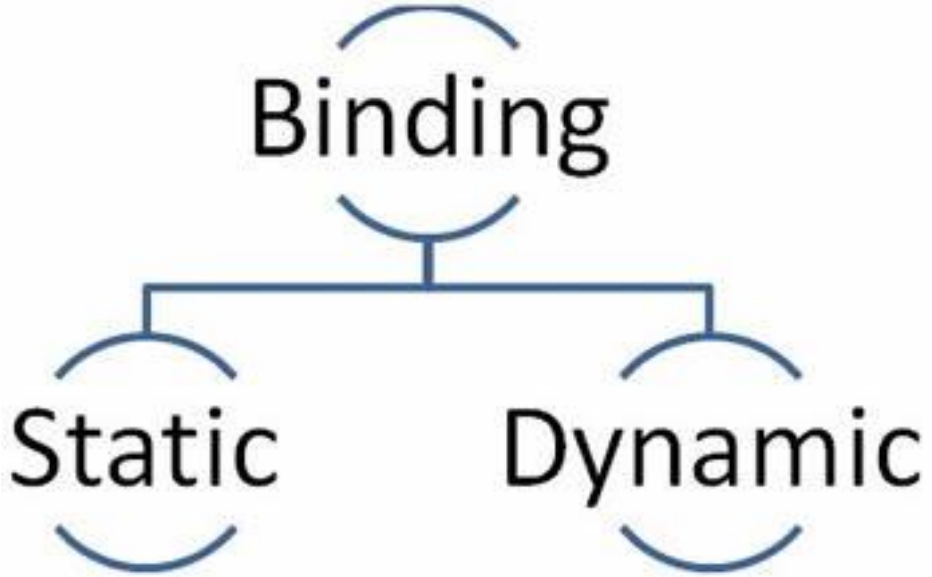
- Type Bindings
- Storage Bindings and Lifetime

Type Bindings

- Before a variable can be referenced in a program, it must be bound to a data type.
- The two important aspects of this binding are
  - How is a type specified?
  - When does the binding take place?

Type Bindings

- Static Type Binding
- Dynamic Type Binding



Static Type Binding

- If type binding is **static**, the type may be specified by either an “explicit” or an “implicit” declaration (i.e., both explicit and implicit declarations create static bindings to types).
- An **explicit** declaration is a statement in a program that lists variable names and specifies that they are a particular type.
- Most widely used programming languages that use static type binding exclusively and were designed since the mid-1960s require explicit declarations of all variables.

```
int count;
```

Static Type Binding

- An **implicit** declaration is a means of associating variables with types through default conventions, rather than declaration statements.
- In this case, the first appearance of a variable name in a program constitutes its implicit declaration.
- There are several different bases for implicit variable type bindings.
- The simplest of these is 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.
- For example, in Perl any name that begins with $ is a scalar. If a name begins with @, it is an array; if it begins with a %, it is a hash structure.

Static Type Binding

- Some languages use **type inferencing** to determine types of variables. This is another kind of implicit type declarations that uses context.
- For example, in C# a variable can be declared with "`var`" and an initial value. The initial value sets the type:

```
var sum = 0;          → int
var total = 0.0;      → float
var name = "Fred";    → string
```

*** Keep in mind that these are statically typed variables—their types are fixed for the lifetime of the unit in which they are declared.

Dynamic Type Binding

- With **dynamic type binding**, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name.
- Instead, the variable is bound to a type when it is assigned a value in an assignment statement.
- Furthermore, a variable's type can change any number of times during program execution.
- In Python, Ruby, JavaScript, and PHP, type binding is dynamic.
- Also, C# supports dynamic type binding (includes the "**`dynamic`**" reserved word).

Dynamic Type Binding

- For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

- Regardless of the previous type of the variable named `list`, this assignment causes it to become the name of a single-dimensioned array of length 2. If the statement

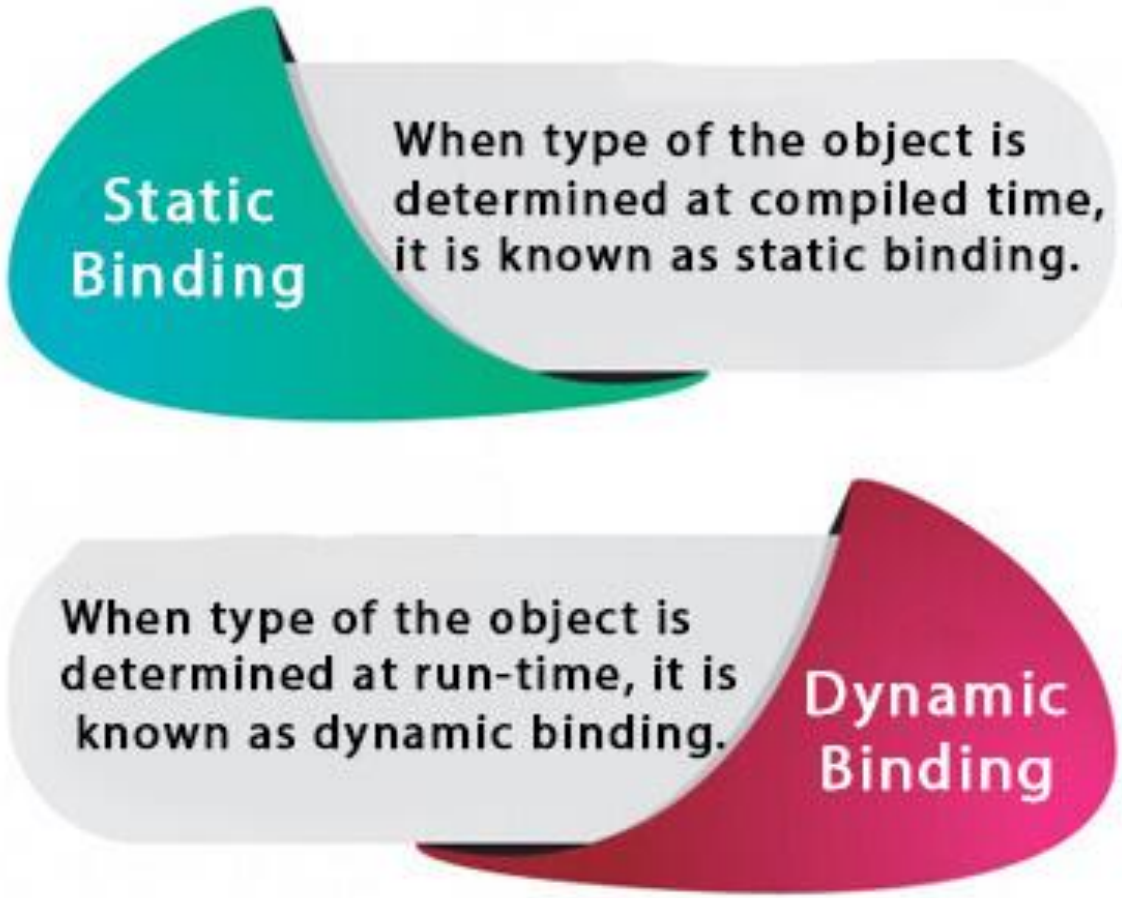
```
list = 47;
```

- followed the previous example assignment, list would become the name of a scalar variable.

Dynamic Type Binding

- Languages that have dynamic type binding for variables are usually implemented using pure interpreters rather than compilers.
- Computers do not have instructions whose operand types are not known at compile time.
- Therefore, a compiler cannot build machine instructions for the expression `A + B` if the types of `A` and `B` are not known at compile time.

Static vs Dynamic Binding



Static vs. Dynamic Binding

- **Static Binding:** binding based on static type.
  - More efficient
  - Less flexible
  - Static type checking leads to safer programs
- **Dynamic Binding:** binding based on dynamic type.
  - Less efficient
  - More flexible
  - May need dynamic type checking!

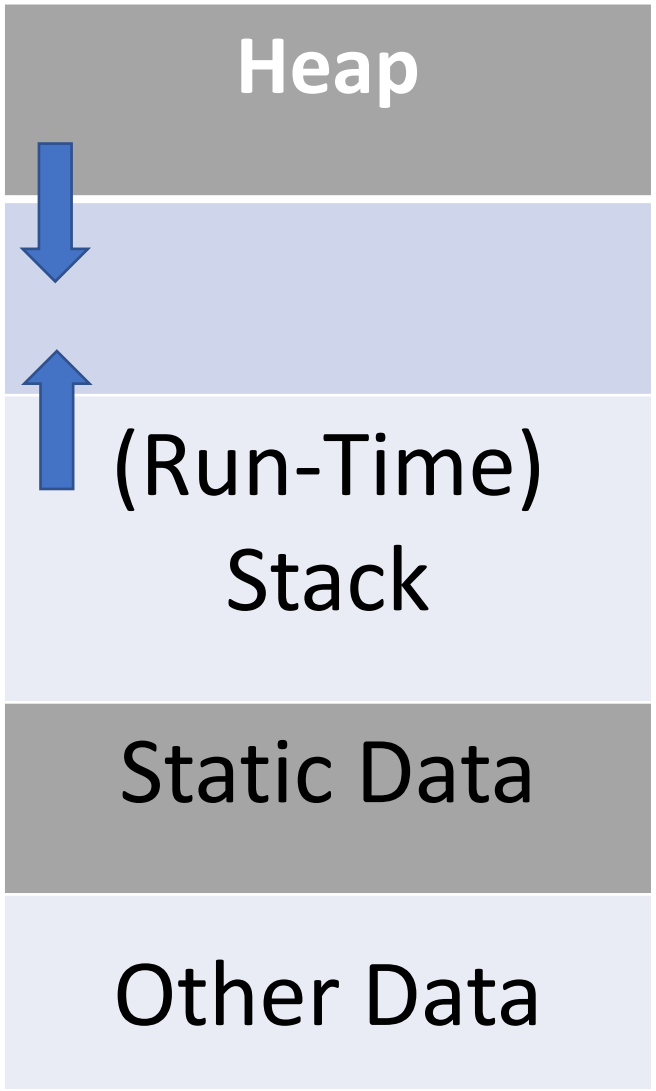
Storage Bindings and Lifetime

- The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called allocation.
- Deallocation is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.
- The lifetime of a variable is the time during which the variable is bound to a specific memory location.

Storage Bindings and Lifetime

- To investigate storage bindings of variables, it is convenient to separate variables into three categories, according to their lifetimes.
  - Static Variables
  - Stack-Dynamic Variables
  - Heap-Dynamic Variables
    - Explicit Heap-Dynamic Variables
    - Implicit Heap-Dynamic Variables

The Memory



Static Variables

- Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.
- Static variables are stored in the "static data" area of memory.
- Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution.

Static Variables

- Sometimes it is convenient to have subprograms that are history sensitive.
- Such a subprogram must have "**local static variables**".
- C and C++ allow programmers to include the `static` specifier on a variable definition in a function, making the variables it defines static.

C Language Local Static Variable Example

```c
int main()
{
    fun();
    fun();
    fun();
    fun();
    fun();

    return 0;
}
```

```c
void fun()
{
    int count = 0;
    count = count + 1;
    printf("The value of count: %d\n", count);
}
```

```
The value of count: 1
The value of count: 1
The value of count: 1
The value of count: 1
The value of count: 1
```

C Language Local Static Variable Example

```c
int main()
{
    fun();
    fun();
    fun();
    fun();
    fun();

    return 0;
}
```

```c
void fun()
{
    static int count = 0;
    count = count + 1;
    printf("The value of count: %d\n", count);
}
```

```
The value of count: 1
The value of count: 2
The value of count: 3
The value of count: 4
The value of count: 5
```

Static Variables

- When the `static` modifier appears in the declaration of a variable in a class definition in C++, Java, and C#, it also implies that the variable is a “class variable”, rather than an instance variable.
- Class variables are created statically some time before the class is first instantiated.

Stack-Dynamic Variables

- Stack-dynamic variables are those whose storage bindings are created when their declaration statements are elaborated.
- Elaboration of such a declaration takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time.
- For example, the variable declarations that appear at the beginning of a Java method (or a C function) are elaborated when the method is called, and the variables defined by those declarations are deallocated when the method completes its execution.
- These types of variables are often called "**local variables**".
- As their name indicates, stack-dynamic variables are allocated from the “run-time stack”.

Explicit Heap-Dynamic Variables

- Heap-dynamic variables are nameless (abstract) memory cells.
- Explicit heap-dynamic variables that are allocated by explicit run-time instructions written by the programmer.
- These variables, which are allocated from and deallocated to the "heap", can only be referenced through pointer or reference variables.
- In Java, all data except the primitive scalars are objects. Java objects are explicitly heap dynamic (created by the `new` keyword) and are accessed through reference variables.

Explicit Heap-Dynamic Variables

- As an example of explicit heap-dynamic variables, consider the following C++ code segment:

```
int *intnode; // Create a pointer
intnode = new int;  // Create the heap-dynamic variable
. . .
delete intnode; // Deallocate the heap-dynamic variable
                // to which intnode points
```

*** Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used.

Implicit Heap-Dynamic Variables

- Implicit heap-dynamic variables are bound to heap storage only when they are assigned values.
- For example, consider the following JavaScript assignment statement:

```
highs = [74, 84, 86, 90, 71];
```

- Regardless of whether the variable named `highs` was previously used in the program or what it was used for, it is now an array of five numeric values.
- In fact, all their attributes are bound every time they are assigned (dynamic).

Implicit Heap-Dynamic Variables

- The advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written.
- One disadvantage of implicit heap-dynamic variables is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others.
- Another disadvantage is the loss of some error detection by the compiler, as discussed before (only run-time type checking can be done).

Scope

Scope

- The **scope** of a variable is the range of statements in which the variable is visible.
- A variable is **visible** in a statement if it can be referenced or assigned in that statement.

Some Important Definitions

- A variable is **local** in a program unit or block if it is declared there (For example in a function in C or in a method in Java).
- The **nonlocal variables** of a program unit or block are those that are visible within the program unit or block but are not declared there.
- **Global variables** are a special category of “nonlocal variables”, which are discussed later.

Static & Dynamic Scoping

Scope

- Static Scope
- Dynamic Scope

Static Scope

- **Static scoping** (sometimes called **lexical scoping**), is so named because the scope of a variable can be statically determined—that is, prior to execution.
- The scope is determined when the code is compiled.
- It's based on program text. This permits a human program reader (and a compiler) to determine the scope of every variable in the program simply by examining its source code.

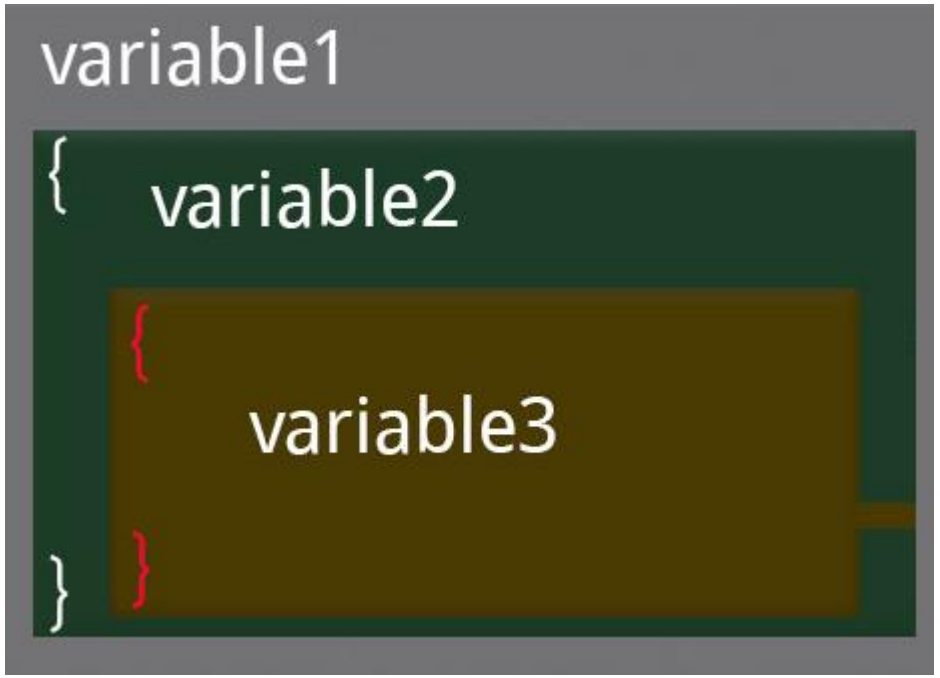
Static Scope

- There are two categories of static-scoped languages:
  - those in which subprograms can be nested, which creates "**nested static scopes**",
  - and those in which subprograms cannot be nested.
- Pascal, Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python allow nested subprograms, but the C-based languages do not.

*** Nested functions have access to variables declared in their outher scope.

Blocks

- Many languages allow new static scopes to be defined in the midst of executable code.
- This powerful concept, introduced in ALGOL 60, allows a section of code to have its own local variables whose scope is minimized.
- Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited.
- Such a section of code is called a **block**. Blocks provide the origin of the phrase **block-structured language (Structured Programming)**.
- To sum up we can say that using blocks is a method of creating static scopes inside program units.



Consider the following C code:

```
int main()
{
    int count=1;
    printf("The value of count: %d\n", count);

    int i=0;
    while(i<10)
    {
        int count=10;
        count++;

        printf("The value of count: %d\n", count);

        i++;
    }

    printf("The value of count: %d\n", count);

    return 0;
}
```

Output of the Code

The value of count: 1
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 11
The value of count: 1

Blocks

- The reference to `count` in the `while` loop is to that loop's local `count`.
- In this case, the value of `count` is hidden from the code inside the while loop.
- In general, a declaration for a variable effectively hides any declaration of a variable with the same name in a larger enclosing scope.
- Note that this code is legal in C and C++ but illegal in Java and C#. The designers of Java and C# believed that the reuse of names in nested blocks was too error prone to be allowed.

When trying to compile the Java code below, an "variable count is already defined in method main(String[])" error will be received.

```
public static void main(String[] args)
{
    int count=1;
    System.out.println("The value of count: " + count);

    int i=0;
    while(i<10)
    {
        int count=10;
        count++;

        System.out.println("The value of count: " + count);

        i++;
     }

     System.out.println("The value of count: " + count);
}
```

Declaration Order

- In some languages, all data declarations in a function must appear at the beginning of the function.
- However, some languages—for example, C, C++, Java, and C#—allow variable declarations to appear anywhere a statement can appear in a program unit.
- In C, C++ and Java, the scope of all local variables is from the declaration to the end of the block.

C and Java Declaration Order Examples

```c
int main()
{
    int a=1;
    int b=2;

    printf("The value of a: %d\n", a);
    printf("The value of b: %d\n", b);
    //System.out.println("The value of c: " + c);

    int c=3;

    printf("The value of a: %d\n", a);
    printf("The value of b: %d\n", b);
    printf("The value of c: %d\n", c);

    return 0;
}
```

```java
public static void main(String[] args)
{
    int a=1;
    int b=2;

    System.out.println("The value of a: " + a);
    System.out.println("The value of b: " + b);
    //System.out.println("The value of c: " + c);

    int c=3;

    System.out.println("The value of a: " + a);
    System.out.println("The value of b: " + b);
    System.out.println("The value of c: " + c);
}
```

Declaration Order

- In C# however, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block.
- Recall that C# (like Java) does not allow the declaration of a variable in a nested block to have the same name as a variable in a nesting scope.
- For example, consider the following C# code:

```
{ int x;
  ...
  { int x; // illegal
  ...
  }
  ...
}
```

Declaration Order

- Because the scope of a declaration is the whole block, the following nested declaration of x is also illegal in C# (which is legal in C, C++, and Java):

```
{
        { int x; // illegal
        ...
        }
        int x;
}
```

*** Note that C# still requires that all be declared before they are used. Therefore, although the scope of a variable extends from the declaration to the top of the block in which that declaration appears, the variable still cannot be used above its declaration.

This Java code is legal:

```
public static void main(String[] args)
{
    int i=0;
    while(i<10)
    {
        int count=10;
        count++;

        System.out.println("The value of count: " + count);

        i++;
    }

    int count=1;
    System.out.println("The value of count: " + count);
}
```

Declaration Order

- In C++, Java, and C# (not in C), variables can be declared in `for` statements. The scope of such variables is restricted to the `for` construct.
- Consider the following skeletal method:

```
void fun()
{
    . . .
    for (int count = 0; count < 10; count++)
    {
        . . .
    }
    . . .
}
```

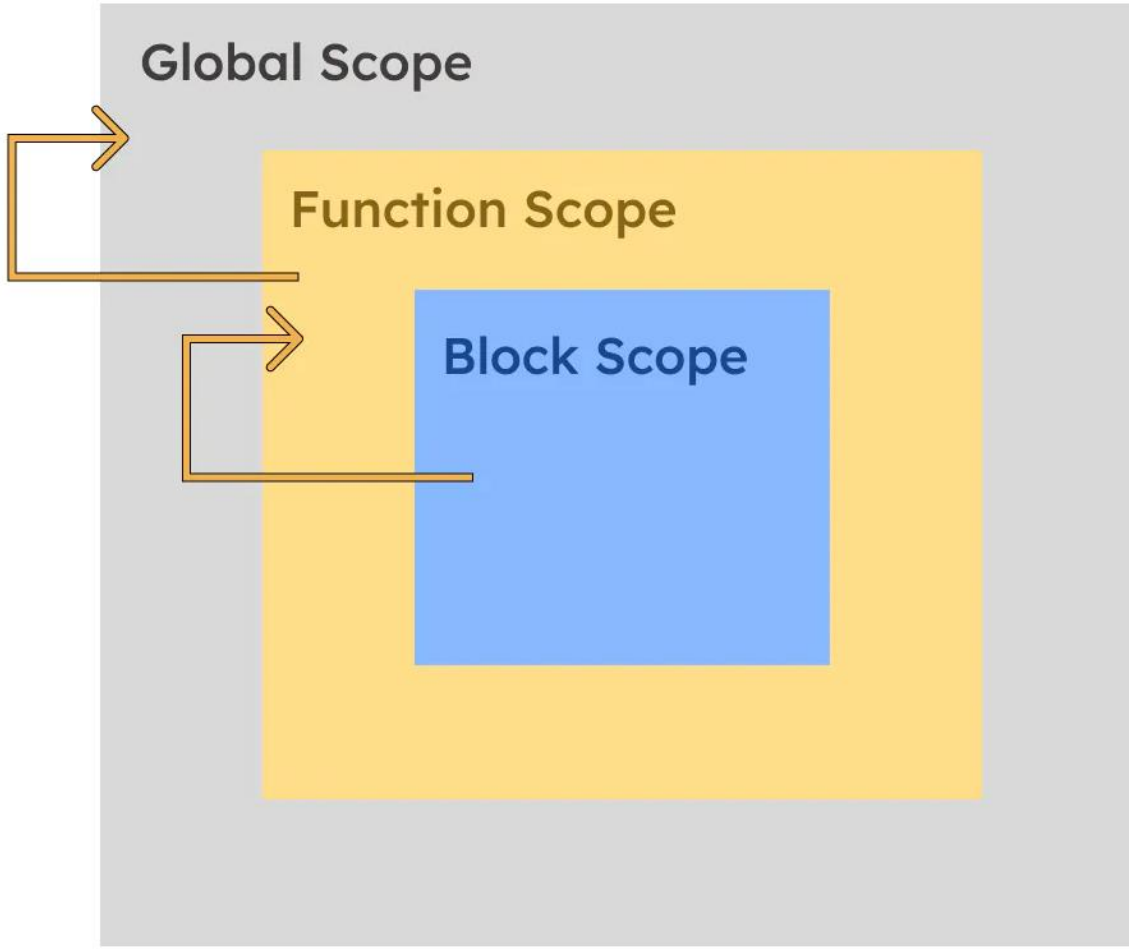
* The scope of `count` is from the for statement to the end of its body.

When trying to compile the Java code below, an “cannot find symbol" error will be received.

```
public static void main(String[] args)
{
    for(int count=0; count<10; count++)
    {
        System.out.println("The value of count: " + count);
    }

    System.out.println("The value of count: " + count);
}
```

Global Scope

- Some languages, including C, C++, PHP, JavaScript, and Python, allow a program structure that is a sequence of function definitions, in which variable definitions can appear outside the functions.
- Definitions outside functions in a file create **global variables**, which potentially can be visible to those functions.



Global Scope – C Example

- A global variable in C is implicitly visible in all subsequent functions in the file, except those that include a declaration of a local variable with the same name.

```c
int sum = 0;

int main()
{
    printf("The value of sum: %d\n", sum);

    fun1();

    printf("The value of sum: %d\n", sum);

    fun2();

    printf("The value of sum: %d\n", sum);

    return 0;
}
```

```c
void fun1()
{
    sum = 5;
}
void fun2()
{
    int sum = 0;
    sum = sum + 1;
}
```

The value of sum: 0
The value of sum: 5
The value of sum: 5

Global Scope – C Example

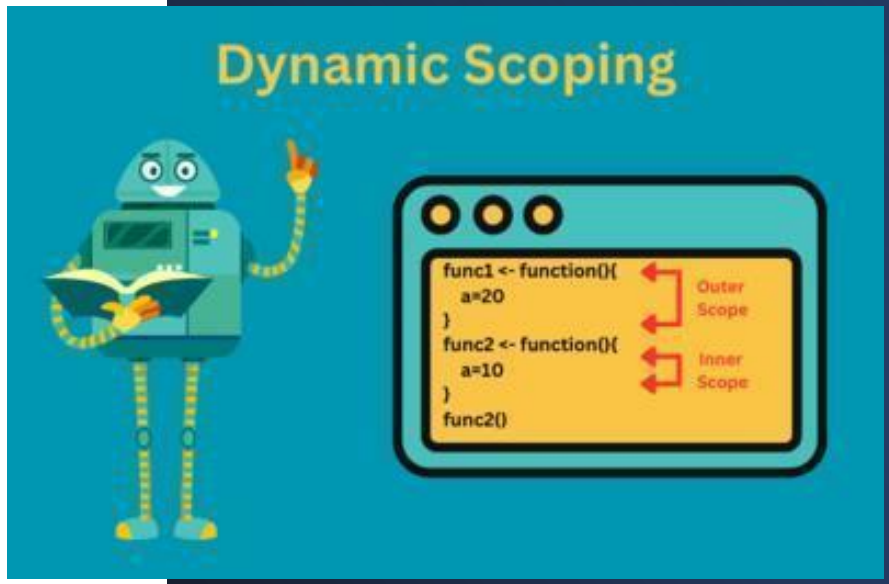
```c
int sum = 0;

int main()
{
    printf("The value of sum: %d\n", sum);

    fun1();

    printf("The value of sum: %d\n", sum);

    fun2();

    printf("The value of sum: %d\n", sum);

    return 0;
}
```

```c
void fun1()
{
    sum = 5;
}
void fun2()
{
    sum = sum + 1;
}
```

The value of sum: 0
The value of sum: 5
The value of sum: 6

Dynamic Scope

- The scope of variables in APL, SNOBOL, and the early versions of Lisp is **dynamic**.
- Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other (not their textual layout).
- Thus, the scope can be determined only at run time.



Scope

- **Lexical Scope** vs **Dynamic Scope**
  - **Lexical Scope**, or **Static Scope**: The scope of a variable is defined by its **location within the source code** and nested functions have access to variables declared in their outer scope.
  - **Dynamic Scope**: The scope of a variable depends on where the functions and scopes are called from
- **Lexical Scope** is write-time, whereas **Dynamic Scope** is runtime
- **Javascript has Lexical Scope!**

Dynamic Scope

- It is not difficult to understand why dynamic scoping is not as widely used as static scoping.
- Programs in static-scoped languages are easier to read (dynamic scoping makes programs much more difficult to read), are more reliable, and execute faster than equivalent programs in dynamic-scoped languages.
- It was precisely for these reasons that dynamic scoping was replaced by static scoping in most current dialects of Lisp.

Dynamic Scope

- On the other hand, dynamic scoping is not without merit.
- In many cases, the parameters passed from one subprogram to another are variables that are defined in the caller.
- None of these needs to be passed in a dynamically scoped language, because they are implicitly visible in the called subprogram.

Dynamic Scope

- The **referencing environment** of a statement is the collection of all variables that are visible in the statement.
- A subprogram is **active** if its execution has begun but has not yet terminated.
- The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active.
- Once again, some variables in active subprograms can be hidden from the referencing environment (with the same names in previous subprogram).

Dynamic Scope

Consider the following example program. Assume that the only function calls are the following: `main` calls `sub2`, which calls `sub1`.

```
void sub1() {
  int a, b;
  . . . <------------ 1
}  /* end of sub1 */
void sub2() {
  int b, c;
  . . . . <------------ 2
  sub1();
}  /* end of sub2 */
void main() {
  int c, d;
  . . . <------------ 3
  sub2();
}  /* end of main */
```

The referencing environments of the indicated program points are as follows:

Point	*Referencing Environment*
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main`, (`c` of `main` and `b` of `sub2` are hidden)
2	`b` and `c` of `sub2`, `d` of `main`, (`c` of `main` is hidden)
3	`c` and `d` of `main`

Scope and Lifetime

Scope and Lifetime

- “Scope” and “lifetime” are sometimes closely related but are different concepts.
- Sometimes the scope and lifetime of a variable appear to be related. For example, consider a variable that is declared in the beginning of a Java method that contains no method calls.
- The scope of such a variable is from its declaration to the end of the method. The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.
- Although the scope and lifetime of the variable are clearly not the same, because static scope is a textual, or spatial concept whereas lifetime is a temporal concept, they at least appear to be related in this case.

SCOPE
&
LIFETIME
OF
VARIABLES

Scope and Lifetime

- This apparent relationship between scope and lifetime does not hold in some other situations.
- For example, scope and lifetime are also unrelated when subprogram calls are involved. Consider the following Java methods:

```
void compute()
{
   int sum;
   . . .
   printheader();
   . . .
}

void printheader()
{
   . . .
}
```

Scope and Lifetime

- The scope of the variable `sum` is completely contained within the `compute` function.
- It does not extend to the body of the function `printheader`, although `printheader` executes in the midst of the execution of `compute`.
- However, the lifetime of `sum` extends over the time during which `printheader` executes.
- Whatever storage location `sum` is bound to before the call to `printheader`, that binding will continue during and after the execution of `printheader`.

Scope and Lifetime

- As an another example, recall that in C and C++, a variable that is declared in a function using the specifier `static` is statically bound to the scope of that function and is also statically bound to storage (local static variables).
- So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

Recap

- Consider the C code below:

```
void my_function()
{
    int my_variable;
    …
    …
}
```

- The type of `my_variable` (int) is bound as “static”. (In C, all types are statically bound.)
- `my_variable` is a “stack-dynamic variable”. (its store binding is dynamic).
- So, the lifetime of `my_variable` extends over the time during which `my_function` executes.
- The scope of the variable `my_variable` is “static” and contained within the `my_function` function. (C is a static-scoping language)

Named Constants

Named Constants

- A named constant is a variable that is bound to a value only once (it is bound to a value only when it is bound to storage).
- Named constants are useful as aids to reliability, readability and modifiability of a program.
- *** Reliability → accidentally trying to change a value that should not change.
- Readability can be improved, for example, by using the name `pi` instead of the constant `3.14159265`.

Named Constants

- Another important use of named constants is to parameterize a program.
- For example, consider a program that processes a fixed number of data values, say 100.
- Such a program usually uses the constant 100 in a number of locations for declaring array subscript ranges and for loop control limits.
- Consider the following skeletal Java program segment:

```
void example() {
  int[] intList = new int[100];
  String[] strList =  new  String[100];
  . . .
  for  (index = 0; index < 100; index++) {
    . . .
  }
  . . .
  for  (index = 0; index < 100; index++) {
    . . .
  }
  . . .
  average = sum / 100;
  . . .
}
```

Named Constants

- When this program must be modified to deal with a different number of data values, all occurrences of 100 must be found and changed.
- On a large program, this can be tedious and error prone.
- An easier and more reliable method is to use a named constant as a program parameter, as follows: (This illustrates how named constants can aid modifiability).

```
void example() {
  final int  len = 100;
  int[] intList =  new int[len];
  String[] strList =  new  String[len];
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  average = sum / len;
  . . .
}
```