Programming Language

Names, Bindings, and Scopes

Contents

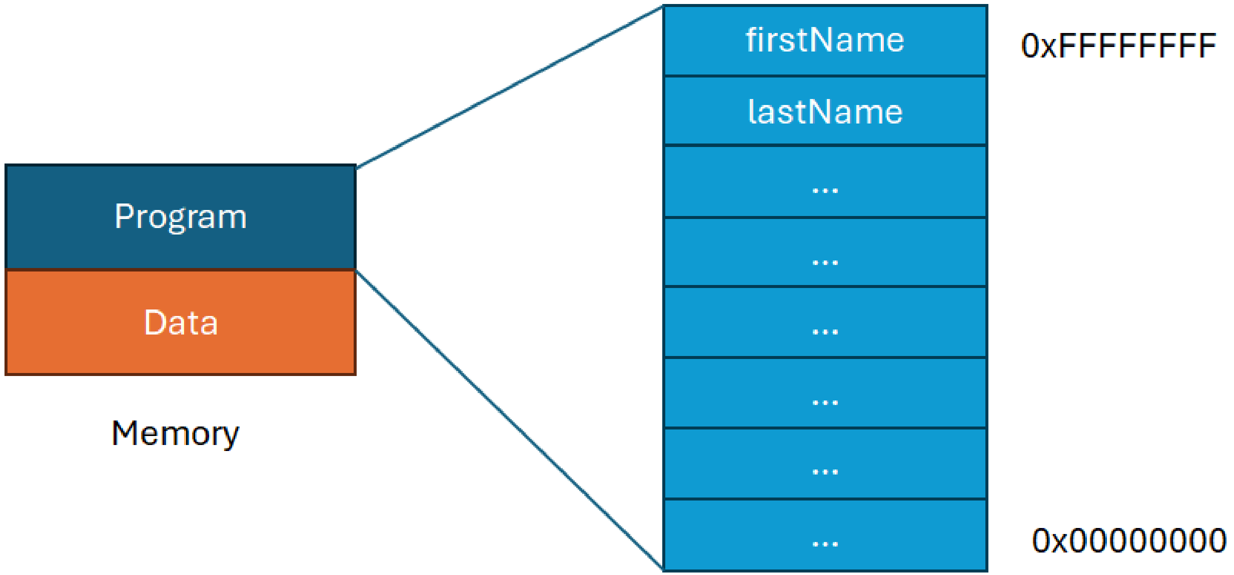
Introduction (1.)

- Programming Languages
  - Imperative programming languages
    - C/C++, Java, C#, etc.
  - Imperative/multi-paradigm programming languages
    - C/C++, PHP, C#, Java, Kotlin, Python
  - Pure functional programming languages
    - Haskell, Lisp, Scheme, Clojure, Erlang, F#, OCaml, Scala

Reference:
https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Imperative_languages

How programming languages related to von Neumann computer architecture

- Variables are stored in memory cells



Design Issues (2.)

- Name Forms
  - A **name** is a string of characters used to identify some entity in a program.
  - The naming rule is a letter followed by a string consisting of letters, digits, and underscore characters (_).
    - Excepts:
      - PHP uses $ at the beginning of variable name
      - Perl uses $, @, % at the beginning of variable name
      - Ruby uses @ or @@ at the beginning of variable name
  - The name length can be varied on languages.
  - Case Sensitive – rose, ROSE are hard to read - **Readability**
    - Function naming can lead to **writability problem**. For example, In Java, **parseInt** is allowed but ParseInt and parseint is not allowed.

- Special Words – reserve word
  - To make program more readable.
  - Cannot be used as a name of variable.
    - In COBOL, there are 300 reserved words, and most of them are common for programmers like LENGTH, BOTTOM, DESTINATION and COUNT.
  - Cannot redeclare variable that import from libraries that are defined in other programs units like Java import C and C++ libraries because reserved words might conflict with variable name.

https://onlinegdb.com/GmRNoCEUq

```
using System;

public class HelloWorld
{
    public static void Main(string[] args)
    {
        var _x = 100;
        int var = 5;
        //float int =9;
        Console.WriteLine ("_x = {0}  ,  var = {1}" , _x,var);
```

- From previous example, C# has Contextual Keywords
- Reserved Keywords (Cannot be used as identifiers)
  - Keywords like int, class, public, if, while are reserved and cannot be used as variable names unless you use the @ prefix:

https://onlinegdb.com/Y_mhDCVRcQ

```
using System;

public class HelloWorld
{
    public static void Main(string[] args)
    {
        var var = 10;
        int @int = 5;      // ✓ Works with @ prefix
        float @float = 2.5f;      // ✓ Works with @ prefix
        Console.WriteLine ("var = {0}, @int = {1}, @float = {2}" ,var, @int, @float); // Contextual Keyword
    }
}
```

Variables (3.)

A program variable is an abstraction of a computer memory cell or collection of cells.

- **Name** – Variable names
- **Address** - The address of a variable is the machine memory address with which it is associated.
- **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.
- **Value** - The value of a variable is the contents of the memory cell or cells associated with the variable.

The Concept of Binding(4.)

1. Binding of Attributes to Variables
2. Type Bindings

   2.1 Static Type Bindings

   2.2 Dynamic Type Bindings
3. Storage Bindings and Lifetime

   3.1 Static Variables

   3.2 Stack-Dynamic Variables

   3.3 Explicit Heap-Dynamic Variables

   3.4 Implicit Heap-Dynamic Variables

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**. Binding and binding times are prominent concepts in the semantics of programming languages.

- Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time.

- Entity = สิ่งที่มีตัวตนในโปรแกรม เช่น:
  - Variables (ตัวแปร)
  - Functions (ฟังก์ชัน)
  - Types (ชนิดข้อมูล)
  - Objects (วัตถุ)
  - Constants (ค่าคงที่)
  - Labels (ป้ายชื่อ)

```
int x = 5;     // global name x
void foo() {
    int x = 10;  // local name x
    scanf ( “%d”, &x  );
}
```

- x → variable name (global & local variable)
- foo & scanf → function name

Example

```
int x = 10;
```

Type of binding:

1. NAME BINDING (Identifier Binding)
   - The identifier "x" is bound to a variable
   - Binding Time: Compile time
2. TYPE BINDING
   - x is bound to type "int"
   - Binding Time: Compile time
3. STORAGE BINDING (Address Binding)
   - x is bound to memory address 0x0318
   - Binding Time: Load/Run time
4. VALUE BINDING
   - x is bound to value 10
   - Binding Time: Run time

Binding of Attributes to Variables (4.1)

- A binding is **static** if it first occurs before run time begins (compile time) and remains unchanged throughout program execution. (Static Binding)
- If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic**. (Dynamic Binding)
- The physical binding of a variable to a storage cell in a virtual memory environment is complex – page or segment of the address space is changed frequently (managed by computer hardware) .

https://onlinegdb.com/K59yXtg8ZD

```
using System;

class GFG {
    static int f2() {
        return 10;
    }

    static string f1() {
        return "string f1";
    }
    // Main Method
    static public void Main()
    {
        dynamic val1 = f1(); //dynamic binding
        Console.WriteLine("type of val1: {0} ,val1 = {1}",
         val1.GetType().ToString(),val1);
        val1 = f2();
        Console.WriteLine("type of val1: {0},val1 = {1}",
                          val1.GetType().ToString(),val1);
        var val3 = f2();    //static binding
        Console.WriteLine("type of val3: {0},val3 = {1}",
                          val3.GetType().ToString(),val3);
         //val3 =  "new value";
```

```csharp
using Newtonsoft.Json;

string json = @"{
    'name': 'John',
    'age': 30,
    'address': {
        'city': 'Bangkok',
        'country': 'Thailand'
    },
    'hobbies': ['reading', 'coding']
}";

// Parse to dynamic
dynamic data = JsonConvert.DeserializeObject(json);

// Access properties
Console.WriteLine(data.name);              // John
Console.WriteLine(data.age);               // 30
Console.WriteLine(data.address.city);      // Bangkok
Console.WriteLine(data.hobbies[0]);        // reading
```

```csharp
using Newtonsoft.Json;
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Address Address { get; set; }
    public List<string> Hobbies { get; set; }
}
public class Address
{
    public string City { get; set; }
    public string Country { get; set; }
}
string json = @"{
    'name': 'John',
    'age': 30,
    'address': {
        'city': 'Bangkok',
        'country': 'Thailand'
    },
    'hobbies': ['reading', 'coding']
}";
Person person = JsonConvert.DeserializeObject<Person>(json);
Console.WriteLine(person.Name);             // John
Console.WriteLine(person.Age);              // 30
Console.WriteLine(person.Address.City);     // Bangkok
Console.WriteLine(person.Hobbies[0]);       // reading
```

- Compiler stores variables in Symbol Table

Name	Type	Address	Value
x	int	0x0318	3
y	char	0x031c	'A'
sum	float	0x0320	5.5

Note
Address in the table is not the actual address, it is symbolic address

Type Bindings (4.2)

- 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 the type is specified and when the binding takes place.

Static Type Bindings (Compile-Time) (4.2.1)

- **Explicit** declaration – use keyword like let, var, const to introduce variable.
  - int a = 0; (C/Java)
  - let a = 0; (JavaScript)
  - var a = 0; (JavaScript)
  - const a = 0; (JavaScript)
- **Implicit** declaration
  - a = 0;
  - Type inference – another kind of implicit type declarations uses **context**.
    - var a = 0; - infer as int
    - var b = “Smith” – infer as string

Dynamic Type Bindings (Runtime) (4.2.2)

- Variables are bound when it is assigned a value in an assignment statement.
- It is good for assigned value from external source that is unknown type.
- However, it is less reliable because dynamic type binding allows any variable to be assigned.
- To check type must be **check at Runtime** by implementing dynamic attribute binding which is the disadvantage.

https://onlinegdb.com/a23fbB1bOo

```python
def func(max):
    imaxx = range(max)
    print("Name 2")
    for i in imaxx:
        x = 1
    print(x)

func(2) # func(0) //what happend?
```

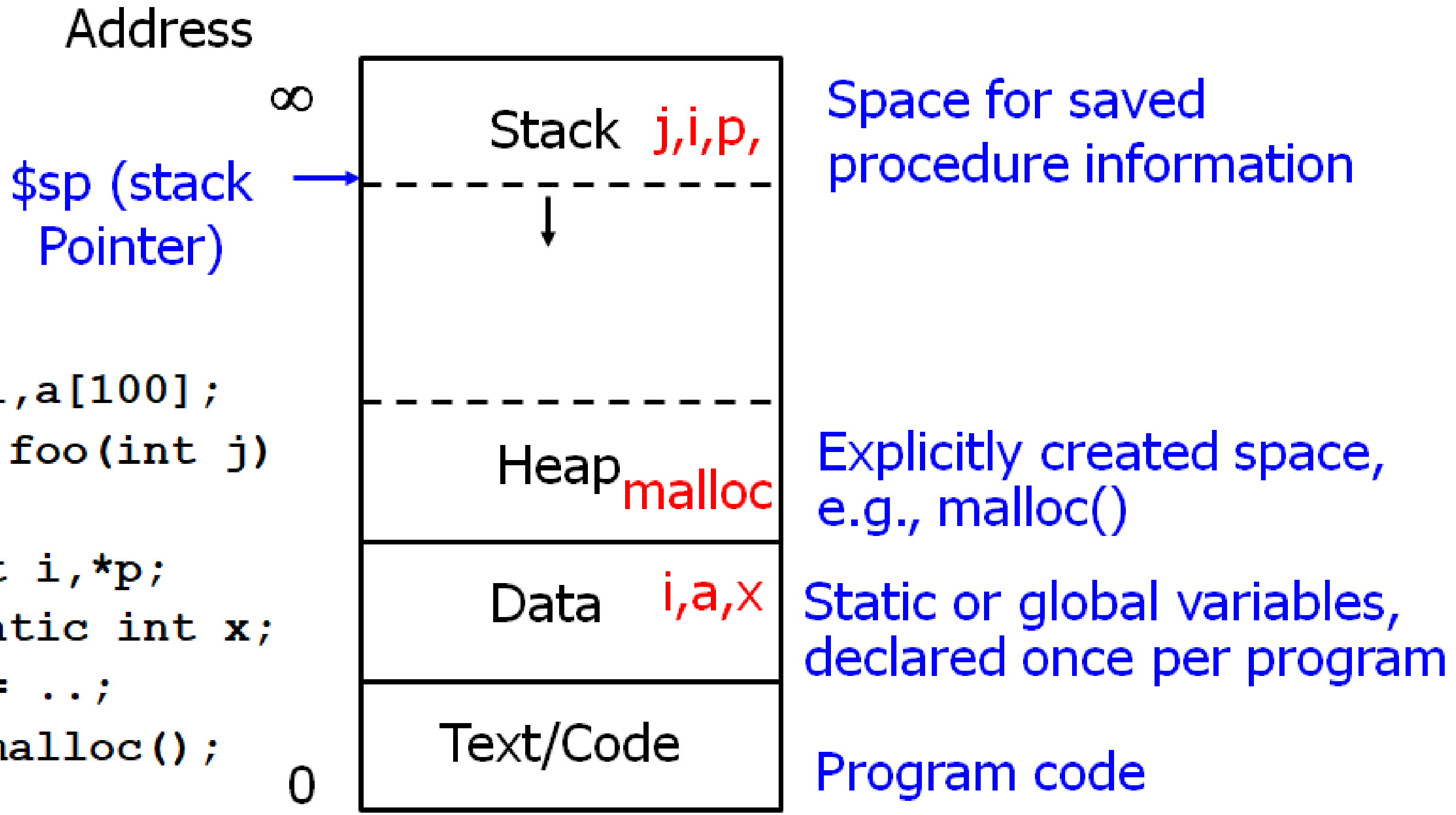
Note

- Python check type at runtime

Storage Bindings and Lifetime (4.3)

- The **storage bindings** for its variables is the process of allocation or deallocation memory cell from a pool of available memory.
  - **Allocation** - The variable is bound to a memory cell.
  - **Deallocation** – The variable is unbound from memory cell then return memory cell to the pool of available memory.
- The **lifetime** of a variables is the time during which the variable is bound to a specific memory cell.

```
int i,a[100];
void foo(int j)
{
   int i,*p;
   static int x;
   i = ..;
   p=malloc();
}
```


Address
∞
$sp (stack Pointer)
Stack j,i,p,
Heap malloc
Data i,a,x
Text/Code
0
Space for saved procedure information
Explicitly created space, e.g., malloc()
Static or global variables, declared once per program
Program code

Static Variables (4.3.1)

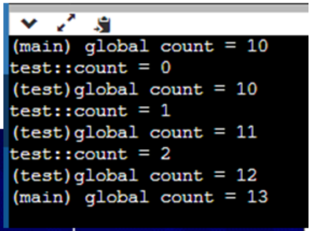
- A static variable is one that is bound to a memory cell before program execution begins and remains bound to that same memory cell until program execution terminates.
  - For example, Global Variables or Class Variables.
  - The advantage of global variables is efficient because of accessing memory directly.

https://onlinegdb.com/DAKoNYz_k

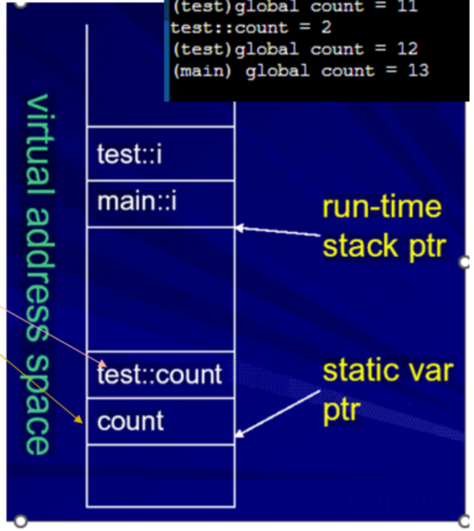
Static vs global Variables Scope

A global variable can be accessed from anywhere inside the program while a static variable only has a block scope.

```
#include <iostream>
using namespace std;
int count=10;
void  test () {
   int i;
   static int count=0;
   cout << "test::count = " << count++  << "\n";
   cout << "(test)global count = " <<  ::count++
   << "\n";
}

int main()
{
    int i;
    cout << "(main) global count = " << count  << "\n";
    test ();
    test ();
    test ();
     cout << "(main) global count = " << count  << "\n";
    return 0;
}
```



```
(main) global count = 10
test::count = 0
(test)global count = 10
test::count = 1
(test)global count = 11
test::count = 2
(test)global count = 12
(main) global count = 13
```



https://onlinegdb.com/ELsXA6jjK

```cpp
#include <iostream>
using namespace std;
int factorial(int n) {
    static int result = 1;  // Static variable - มี 1 ที่เดียว!

    if (n <= 1) return result;

    result = n * factorial(n - 1);  // ทุก call ใช้ result ตัวเดียวกัน!
    return result;
}

int main(){
    cout << "First call:  " << factorial(3) << "\n";   // ✅ ได้ 6
    cout << "Second call: " << factorial(3) << "\n";   // ❌ ได้ 36 (ผิด!)
    cout << "Third call:  " << factorial(4) << "\n";   // ❌ เพี้ยนไปเลย!
}
```

- The disadvantages are following:

1. **Reduced Flexibility** (Cannot Support Recursion)
   - **The Problem:** **Static variables** are bound to memory at compile time and keep the **same memory location** for the entire program execution.
   - **The Consequence:** A function that uses local static variables cannot call itself recursively.
   - **Why?** Recursion requires that every time a function calls itself, it gets a fresh, unique copy of its local variables and parameters. Since a static variable has only one memory location, every recursive call would try to write to the same spot, corrupting the data from the previous calls and making recursion impossible.

Stack-Dynamic Variables (4.3.2)

- **Stack-dynamic variables** are those whose storage bindings are created when their declaration statements are elaborated , but whose types are statically bound. It is created from the run-time stack.

https://onlinegdb.com/JKUvXRUygU

```cpp
#include <iostream>
using namespace std;
void function(){
    int x = 10;
}
int main(){
    function();
    return 0;
}
```

Note
elaborated = when the declaration is executed at runtime

- **The disadvantage** is the run-time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required.

https://onlinegdb.com/fn1b39X6_

```python
def hello(x):
    if x==1:
        return "op"
    else:
        u=1
        e=12
        s=hello(x-1)
        e+=1
        print(s)
        print(x)
        u+=1
    return e

hello(3)
```

- What happend?

https://onlinegdb.com/xEg4b8Rqq

```
#include <iostream>
using namespace std;
int dof1(int run)
{
    char x[1024*1024];
    printf("\nrun=%d, &run=%p, &x[0]=%p",run,&run,&x[0]);
    if(run > 0) {
        dof1(run-1);
        return 0;
    }
    return 1;
}
int main() {
    dof1( 10);
    return 0;
}
```

- What happend?

https://onlinegdb.com/xEg4b8Rqq

```
#include <iostream>
using namespace std;
int dof1(int run)
{
    char x[1024*1024];
    printf("\nrun=%d, &run=%p, &x[0]=%p",run,&run,&x[0]);
    if(run > 0) {
        dof1(run-1);
        return 0;
    }
    return 1;
}
int main() {
    dof1( 10);
    return 0;
}
```

- Buffer overflow!

Explicit Heap-Dynamic Variables (4.3.3)

- Explicit heap-dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer.

- The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation.

```c
People *p = (struct People *)malloc(sizeof(People));
```

Implicit Heap-Dynamic Variables (4.3.4)

- Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned.

```
heights = [74, 84, 86, 90, 71]
```

Note
The entire mechanism is implicit—the programmer never wrote malloc() or free(). The system manages the Heap storage binding and updates all attributes (type, size, location) dynamically at runtime, every time the variable is assigned or modified.

Scope (5.)

- 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.
- A variable is **local** in a program unit or block if it is declared there.
- 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.

Why variable scope?

- Given multiple bindings of a single name, how do we know which binding does an occurrence of this name refer to?
  - Two bindings for "x"
    - one of type *int*
    - another *float*

https://onlinegdb.com/NwCuuYk2d

```cpp
#include <iostream>
using namespace std;
int x;
void foo(int y)
{
    float x;
    x = 10;
}
int main(){
    foo(x);
    return 0;
}
```

Static Scope (5.1)

- Scope of a variable can be statically determined
  - Based on program text, a spatial concept
- To connect a name reference to a variable, you (or the compiler) must find the declaration
  - First search locally, then in increasingly larger enclosing scopes, until one is found for the given name, or an undeclared variable error
- Variables can be hidden from a unit by having a "closer" variable with the same name
  - C++ and Ada allow access to "hidden" variables:
    - unit.name (in Ada)
    - class_name::name (in C++)

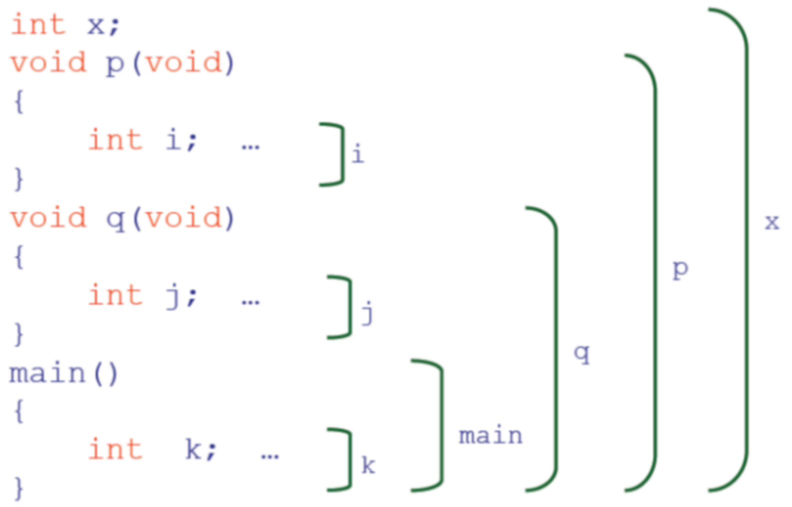
https://onlinegdb.com/uF1vmbknX

```cpp
#include<iostream>
using namespace std;

class MyClass {
public:
    static int count;              // Static variable (declaration)
    static void displayCount() {   // Static method
        cout << "Count: " << count << endl;
    }
};

// Initialize static variable outside class
int MyClass::count = 0;

int main() {
    // Access static members WITHOUT creating an object
    MyClass::count = 5;
    MyClass::displayCount();       // Output: Count: 5

    // Can also access via object (but not recommended)
    MyClass obj;
    obj.count = 10;                // Works but confusing
}
```

- **Block**: a method of creating new static scopes inside program units (from ALGOL 60)
    - e.g.: C and C++ in any compound statement

```
for (...) {
    int index;
    ...
}
```

An Example of Block Scope in C variable & function call



```
int x;
void p(void)
{
    int i;  …
}
void q(void)
{
    int j;  …
}
main()
{
    int  k;  …
}
```

Dynamic Scope (5.2)

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
  - Can only be determined at run time
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

- What scope is it?

https://onlinegdb.com/n9E-USPy3

```
#!/usr/bin/perl
# A perl code to demonstrate dynamic scoping

$y = 1;
$z = 2;
$a = 3;
$b = 4;

sub f {
    print "a = ".$a."\n";
    print "b = ".$b."\n";
    print "x = ".$x."\n";
    print "y = ".$y."\n";

    return $x;
}
```

```
sub g {
    # Since local is used, x uses
    # dynamic scoping.
    local $x = 20;
    print "\ng = ".f()."\nend\n\n";
}
sub g2 {
    # Since local is used, y uses
    # dynamic scoping.
    local $y = 200;
    print "\ng = ".f()."\nend\n\n";
}

$x = 10;
print "\n 1. f = ".f()."\n";
g();
g2();
#print "\n 2. f = ".f()."\n";
```

- What scope is it?

- Dynamic and static

https://onlinegdb.com/n9E-USPy3

```perl
#!/usr/bin/perl
# A perl code to demonstrate dynamic scoping

$y = 1;
$z = 2;
$a = 3;
$b = 4;

sub f {
    print "a = ".$a."\n";
    print "b = ".$b."\n";
    print "x = ".$x."\n";
    print "y = ".$y."\n";

    return $x;
}
```

```perl
sub g {
    # Since local is used, x uses
    # dynamic scoping.
    local $x = 20;
    print "\ng = ".f()."\nend\n\n";
}
sub g2 {
    # Since local is used, y uses
    # dynamic scoping.
    local $y = 200;
    print "\ng = ".f()."\nend\n\n";
}

$x = 10;
print "\n 1. f = ".f()."\n";
g();
g2();
#print "\n 2. f = ".f()."\n";
```

- What scope is it?

https://onlinegdb.com/_FslPoinF

```
#include <iostream>
using namespace std;
int x=10;
int y=1;
int z=2;
int a=3;
int b=4;
int f(int x,int y)
{
   cout <<"\n" << a <<"\n"<< b<<"\n" << x <<"\n"<<y <<"\n";
   return x;
}
void  g ()
{
   int x=20;
   cout << "\ng.f " << f (x,y);
}
```

```
void  g2 ()
{
    int y=200;
    cout << "\ng2.f " << f (x,y);
}
int main()
{
    cout << "\nmain.f " << f (x,y);
    g();
    g2();
    //cout << "\n main.f " << f (x,y,z,a,b);
    return 0;
}
```

- What scope is it?

- Static

https://onlinegdb.com/_FsIPoinF

```
#include <iostream>
using namespace std;
int x=10;
int y=1;
int z=2;
int a=3;
int b=4;
int f(int x,int y)
{
   cout <<"\n" << a <<"\n"<< b<<"\n" << x <<"\n"<<y <<"\n";
   return x;
}
void  g ()
{
   int x=20;
   cout << "\ng.f " << f (x,y);
}
```

```
void  g2 ()
{
    int y=200;
    cout << "\ng2.f " << f (x,y);
}
int main()
{
    cout << "\nmain.f " << f (x,y);
    g();
    g2();
    //cout << "\n main.f " << f (x,y,z,a,b);
    return 0;
}
```

- What scope is it?

https://onlinegdb.com/-XkqNIZN2p

```
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
   print "   x = ".$x."\n";
   return $x;
}
```

```
sub g
{
   # Since local is used, x uses
   # dynamic scoping.
   print  "2. g  \n";
   local $x = 20;
   f();
}


print " 1. m \n";
f();
g();
```

- What scope is it?

- Dynamic

https://onlinegdb.com/-XkqNIZN2p

```
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
   print "   x = ".$x."\n";
   return $x;
}
```

```
sub g
{
   # Since local is used, x uses
   # dynamic scoping.
   print  "2. g  \n";
   local $x = 20;
   f();
}


print " 1. m \n";
f();
g();
```

Static vs. Dynamic Scoping

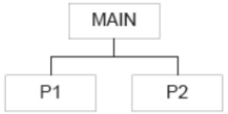
```
program MAIN;
  var a : integer;

  procedure P1;
  begin
    print a;
  end; {of P1}

  procedure P2;
  var a : integer;
  begin
      a := 0;
      P1;
  end; {of P2}

  begin
      a := 7;
      P2;
  end. {of MAIN}
```

static (lexical)

non-local variables are
bound based on program
structure

if not local, go "out" a level

➔ example prints 7

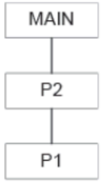


dynamic

non-local variables are
bound based on calling
sequence

if not local, go to calling point

➔ example prints 0



- Static scoping (my)

https://onlinegdb.com/UDBpkpqsA

```
#!/usr/bin/perl
# Static (Lexical) Scoping

my $a;  # MAIN's a

sub P1 {
    print "P1 prints a = $a\n";
}

sub P2 {
    my $a = 0;   # my = lexical scope (local to P2 only)
    P1();        # P1 sees MAIN's $a = 7
}

# MAIN
$a = 7;
P2();

# Output: P1 prints a = 7
```

- Dynamic scoping (local)

https://onlinegdb.com/YzAwoAViw

```
#!/usr/bin/perl
# Dynamic Scoping

$a = undef;  # MAIN's a (global)

sub P1 {
    print "P1 prints a = $a\n";
}

sub P2 {
    local $a = 0;  # local = dynamic scope
    P1();          # P1 sees P2's $a = 0
}

# MAIN
$a = 7;
P2();

# Output: P1 prints a = 0
```

- Perl supports both scopes

- Static Scoping

https://onlinegdb.com/FM1p6DtZ5

```
program MAIN;
var a : integer;

procedure P1;
begin
    writeln('P1 prints a = ', a);
end;

procedure P2;
var a : integer;
begin
    a := 0;
    P1;  { P1 sees MAIN's a = 7 (static) }
end;

begin
    a := 7;
    P2;
end.

{ Output: P1 prints a = 7 }
```

- Dynamic Scoping

https://onlinegdb.com/M5vbgOKhQ

```
#!/bin/bash
a=7
P1() {
    echo "P1 prints a = $a"
}
P2() {
    local a=0
    P1
}

echo "=== Dynamic Scoping in Bash ==="
echo "MAIN sets a = 7"
P2
echo "Result: P1 prints 0 (dynamic scope)"
```

https://onlinegdb.com/l1xTIllxJ

- Static Scoping

```cpp
#include <iostream>
using namespace std;

int a;  // MAIN's a

void P1() {
    cout << "Static: P1 prints a = " << a << "\n";
}

void P2() {
    int a = 0;  // local a in P2
    P1();  // P1 sees global a = 7 (static scoping)
}

int main() {
    a = 7;
    P2();
    return 0;
}
// Output: Static: P1 prints a = 7
```

https://onlinegdb.com/HVYi71luP

- Dynamic Scoping

```cpp
#include <iostream>
using namespace std;

int a;  // global a

void P1() {
    cout << "Dynamic: P1 prints a = " << a << "\n";
}

void P2() {
    int saved_a = a;  // save old value
    a = 0;            // modify global (simulate dynamic)
    P1();             // P1 sees modified a = 0
    a = saved_a;      // restore
}

int main() {
    a = 7;
    P2();
    return 0;
}
// Output: Dynamic: P1 prints a = 0
```

Scope and Lifetime (6.)

Scope and lifetime are different concepts:

- Scope: Where in the code a variable is accessible (textual/spatial)
- Lifetime: When during execution a variable exists in memory (temporal)

Example: C++ static local variable

```cpp
#include <iostream>
using namespace std;

void printheader() {
    cout << "\n=== Computing Sum ===\n";
}

void compute() {
    static int sum;  // static local variable
    printheader();   // calls another function

    sum += 10;
    cout << "Sum = " << sum << "\n";
}
```

```cpp
int main() {
    cout << "Call 1:";
    compute();  // sum = 10

    cout << "\nCall 2:";
    compute();  // sum = 20 (keeps value!)

    cout << "\nCall 3:";
    compute();  // sum = 30 (keeps value!)

    return 0;
}
```

- Scope of sum: Only inside compute() function
- Lifetime of sum: Entire program execution
- Result: When printheader() executes, sum exists in memory but printheader() cannot access it (scope restriction)
- Key insight: A variable can be alive but invisible!

Summary

- Variables are abstractions for memory cells of the computer and are characterized by name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities: type and storage binding
- Scope determines the visibility of variables in a statement

- Exercise

จงอธิบายข้อดีของ static local ของภาษา C/C++ เมื่อเปรียบเทียบกับตัวแปร global ?