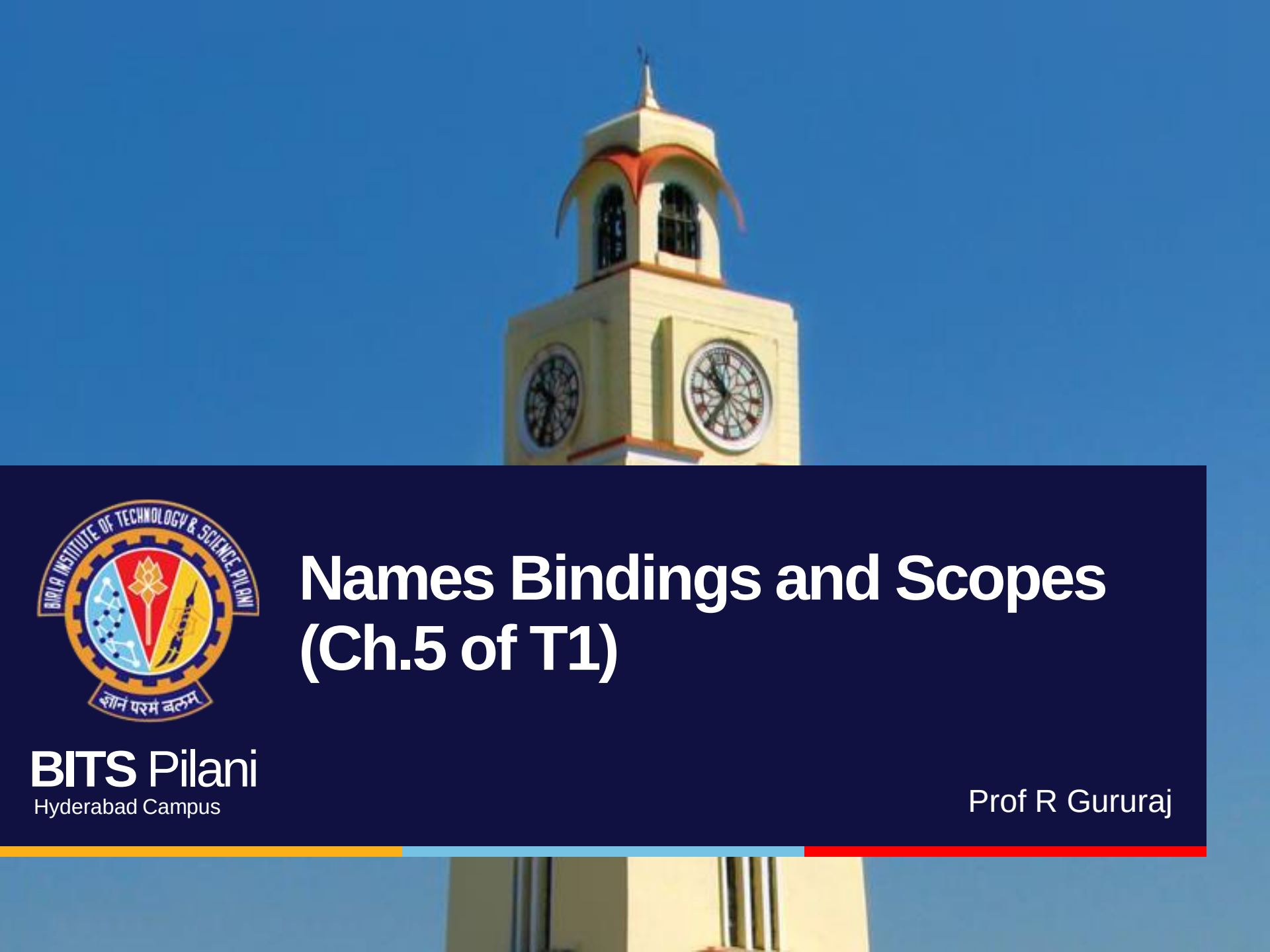
Principles of Programming Languages(CS F301)

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.



Names Bindings and Scopes (Ch.5 of T1)

BITS Pilani
Hyderabad Campus

Prof R Gururaj

Introduction

In Von-Neumann architecture, the important components are-

Memory

CPU

The abstractions in a language, for the memory cells of the machine are *variables*.

A variable can be characterized by collection of properties or attributes most important of which is – *type* , a fundamental concept in a programming language.

Designing types includes *scope* and *lifetime*.

Pure function oriented programming languages do not support names like imperative languages.

C-based languages- C, Objective C, C++, Java and C# support.

Names

Names associated with variables.

Names are also associated with – subprograms, parameters, other program constructs.

The term *identifier* is often used interchangeably with name.

Design issues:

1. Are names case sensitive.
2. Are the special words are reserved words or keywords.

Name forms:

Fortran95+	31 characters
C-99	no limit but first 63 significant
Java, C++, C#, Ada	no limit all are significant

Names mostly consist of letter/digit/underscore

All C-based use Camel notation.

Use of underscore and mixed-case is a programming style, not a programming language design issue.

In PHP all names/variables start with $.

In Perl, $, @, % at the beginning specify its type.

In Ruby @ and @@ indicate instance and class name.

C-based languages are case sensitive.

SQL is not case sensitive.

Special Words

Used to make programs more readable by naming actions to be performed.

They are also used to separate the syntactic parts of the statements and programs.

In most of the languages special words are classified as-

Reserved words (can not be used as names)- can't be redefined. If you have large number of reserved words then user will have problem making names. COBOL has 300 reserved words (BOTTOM, LENGTH, DESTINATION, COUNT etc).

Keywords (ex. Fortran)

same as reserved words, but these can be redefined

Keyword is special only in certain contexts.

Fortran is the language remaining still maintaining keywords.

Real Apple

Real=3.4

Integer Real

Real Integer

Are valid in Fortran. Distinguished based on the context

Variables

- A *variable* in a program is an abstraction of a computer memory cell or collection of cells.
- Programmers often think of variables as names given to memory cells.

In machine language memory locations have binary addresses.

In Assembly language we can give names (more readable)

A variable is characterized by following attributes (is a six-tuple):

- Name
- Address (l-value)
- Content (r-value)
- Type
- Lifetime
- Scope

Other concepts related to variables are:

- Aliasing
- Binding
- Binding times
- Declarations
- Scoping rules
- Referencing environment

Name

- Names are associated with variables.

Address

- Address of a variable is the machine memory address with which it is associated.
- Address of variable is also known as its *l – value*, because the address is what is required when the name of variable appears on the LHS of assignment.

Ex: sum= total;

- It is possible to have multiple variables have same address. In such case the variables are called as *aliases*.
- It can create problems. It sometimes reduces the readability.
- Ex: Unions, Pointers, parameters, object references etc.

Type

- The range of values a variable can take, and set of operations defined on the values.

Values

- It is the content stored in the memory location corresponding to the variable.
- Also known as *r-value* of the variable, because it is what is required when the name of the variable appears in the RHS of assignment.

  Ex: sum= total;

The Concept of Binding (5.4)

- It is an association between an attribute and an entity
  Ex: a variable and type/value
  a symbol and operation and
- The time at which binding takes place is called as *binding time*.

The binding and the binding time are the prominent features in the semantics of a programming language.

A binding can take place at-

- Language design time (* → multiplication)
- Language implementation time (int → range)
- Compile time (binding of variables to particular data types.)
- Load time (a variable binding to memory location)
- Link time (call library subprogram – binding to subprogram code happens at link time)
- Run time (binding variables to memory)

Example:

count=count + 5;

Type of the variable *count* - compile time.

Possible values for *count* – language design time.

Meaning of + is decided at compile time when the operands are known.

Internal representation of literal 5 is done at design time.

Value of count- is bound at execution time.

Static and Dynamic Binding

Binding is static if it occurs before runtime and remains unchanged throughout the program execution.

If it changes it is dynamic binding.

Static Binding

Binding is static if it occurs before runtime and remains unchanged throughout the program execution.

Type Binding

Before a variable is used in a program, it must be bound to some data type.

Static type binding:

Explicit declaration: int a;

Implicit declaration: (like in Fortran) if an identifier begins with I,J,K,L,M,N it is an integer type otherwise it is real type.

Type inference (implicit using the context):

Ex: C# var sum=0;

var sum=0.0;

var name="Tom";

Dynamic Binding

The type of the variable is not specified by a declaration statement, rather a variable is bound to a type when it is assigned a value in assignment statement.

More flexible.

Can't be determined by name spelling etc.

Advantage is more generic programs can be written. Ex: we can write programs to handle any numeric data.

No strict checking of type.

Up till mid 90s many PLs were static except LISP.

Later Python, Ruby, JavaScript, PHP came up which are dynamic.

In Java script-

list = [10.4, 6.75] is an array

list=45; int type

In C#(2010) a variable can be declared as dynamic Ex: dynamic sum;

Issues with dynamic binding:

i=x; x is an array

i=y; y is an int

Later if we have some computation involving *i* and expecting an array, it may give wrong results.

But in static binding it is not allowed. *i=y;* will not be executed.

Storage Binding & Lifetime

The memory cell to which a variable is bound, must be taken from the pool of available memory.

When you acquire space → *allocation*

release space it is → *deallocation*

The lifetime of a variable is the time during which the variable is bound to a specific memory location.

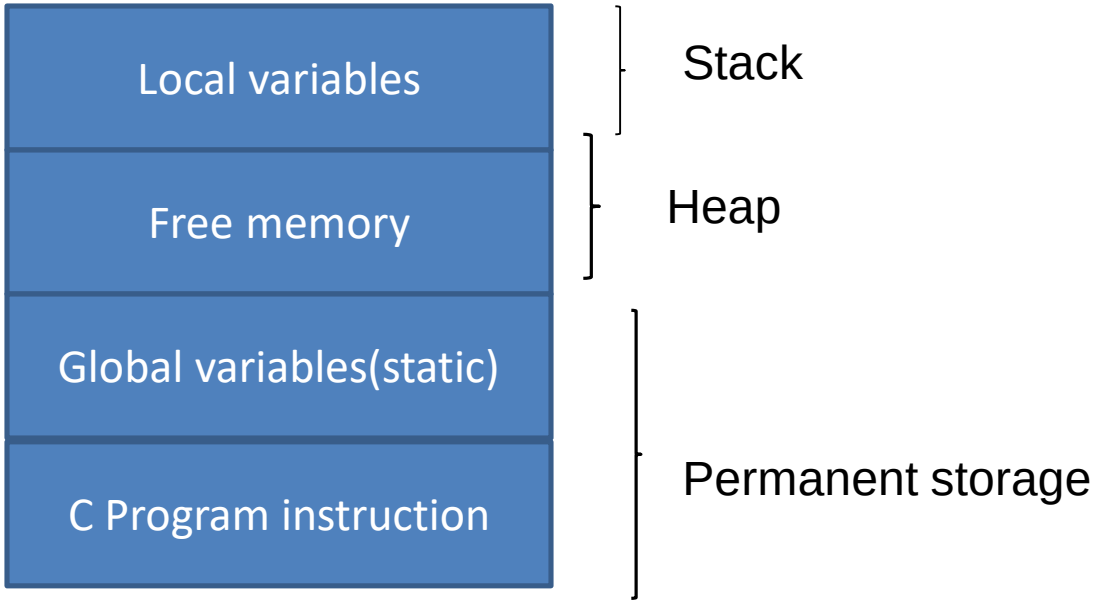
lifetime begins-on allocation.

ends -on deallocation.

Dynamic memory allocation *malloc(100 * sizeof(int)).*

Categories of variables:

- Static
- Stack dynamic
- Explicit Heap-dynamic
- Implicit heap-dynamic


Local variables
Free memory
Global variables(static)
C Program instruction
Stack
Heap
Permanent storage

Static variables

Binding done before execution starts.

Usually Global.

Advantage: Direct addressing is used, hence efficient.

One disadvantage is - Reduced flexibility: space can't used by two sub programs.

Recursion not possible.

Ex. Static variables of class.

Stack dynamic variables:

Binding is done when their declaration statements are elaborated.

But types are statically bound.

Space allocated when method is called.

And deallocated when method is exited.

Advantage: (i) recursion is possible. (ii) Efficient.

Disadvantage:

Overhead due to allocation and deallocation.

They are implicitly deallocated when the function is exited.

Heap-dynamic variables:

Explicit heap-dynamic variables:

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

Referenced through pointers or references only.

Created explicitly either by operator (new) or call to system subprograms.

Ex: *int *intnode;*

intnode= new int;

Explicit deallocation is done in some PLs like C++.

delete intnode;

Explicit deallocation is done in some PLs like C++.

Heap is collection of storage cells. Highly disorganized.

Often used to build dynamic structures like stack, tree etc which can be built conveniently using pointers/ref.

Disadvantages:

1, difficulty in using pointers.

2. Cost induced by references.

3. Complexity in storage management implementation.

Heap-dynamic variables:

Implicit heap-dynamic variables:

Variables are bound to heap-storage only when they are assigned values.

marks[12.5, 59.5, 13.5]

-Highly flexible.

-Generic code can be written.

Scope

The scope of the variable is the range of statements in which the variable is visible.

A variable is visible in a statement if it can be referenced in the statement.

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

Static Scope: determined prior to execution.

Static scope: Scope is determined prior to execution.

Introduced in ALGOL60.

Nested subprograms:

Ada, Java script, Python etc. support nested sub programs.

Many programming languages allow new static scopes by putting the statements in curly braces.

Static scoping in case of Nested subprograms:

Ada, Java script, Python etc. support nested sub programs.

Java Script Example

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

The function big()
is the parent of sub1() and sub2()

We have *parent* and *ancestral* relationships among programs.
Here big() is the static parent of sub1 and sub2

Dynamic Scoping

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.
Ex: APL, SNOBOL4 , early versions of LISP.

Less Reliable
Hard to read
Accessing non local variables takes longer

```
function big() {
   function sub1() {
      if (cond)
      Sub2C) ;
      var x = 7;
   }
   function sub2() {
      var y = x;
      var z = 3;
   }
   if (cond) Sub2C);
   var x = 3;
}
```

JavaScript

Consider the two different call sequences for sub2 in the earlier example. First, big calls sub1, which calls sub2. In this case, the search proceeds from the local procedure, sub2, to its caller, sub1, where a declaration for x is found. So, the reference to x in sub2 in this case is to the x declared in sub1. Next, sub2 is called directly from big. In this case, the dynamic parent of sub2 is big, and the reference is to the x declared in big.

Note that if

```
void sub() {
  int count;
  ...
  while (...) {
    int count;
    count++;
    ...
  }
  ...
}
```

In general, a declaration for a variable effectively hides any declaration of a variable with same name in a larger enclosing scope.

This is valid in C and C++ , but illegal in Java and C# as they believe that this is error prone.

Global scope: variables declared outside the function are in global scope.

Global Scope

Global scope:

C, C++, Java script, and python allow program structures which are sequence of function definitions.

Variables defined outside are global and visible to all functions.

PHP code for global variables.

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day <br />";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month <br />";
}

calendar();
```

Output:

```
local day is Tuesday
global day is Monday
global month is January
```

(1) If the function has a local variable as the global variable, then the

global can be accessed through the $GLOBALS array, using the name of the global as a string literal subscript, and (2) if there is no local variable in the function with the same name as the global, the global can be made visible by including it in a `global` declaration statement.

Scope and Lifetime

The scope and lifetime of a variable appear to be related.

Referencing environments

The referencing environment of a statement is the collection of all variables that are visible to the statement.

Referencing environments

The referencing environment of a statement is the collection of all variables that are visible to the statement.

Python code

```
g = 3;  # A global

def sub1():
    a = 5;  # Creates a local
    b = 7;  # Creates another local
    ...  <-------------------------------------- 1
  def sub2():
    global g;  # Global g is now assignable here

    c = 9;  # Creates a new local
    ...  <-------------------------------------- 2
    def sub3():
      nonlocal c:  # Makes nonlocal c visible here
      g = 11;  # Creates a new local
      ...  <-------------------------------------- 3
```

Point	*Referencing Environment*
1	local `a` and `b` (of `sub1`), global `g` for reference, but not for assignment
2	local `c` (of `sub2`), global `g` for both reference and for assignment
3	nonlocal `c` (of `sub2`), local `g` (of `sub3`)

Named Constant

innovate achieve lead

A named constant is a variable that is bound to a value once.

The increase readability and reliability of programs

Ex. Use pi in place of 3.1415965

In Java we have -

final int len=20;

In C++ we have:

*const int area= width*bredth ;*

Summary of Ch.5

innovate achieve lead

1. What is a variable and name.
2. Reserved word and keyword.
3. Variables and their attributes.
4. The concept of binding.
5. Static and dynamic binding.
6. Storage binding and lifetime
7. Categories of variable.
8. Scope.
9. Lifetime.
10. Referencing environment of a variable.
11. Named constants.