Programming Language Concepts

Dr. Imran

Lecture 2

Programming Language Concepts
Variables, Names and Bindings

VARIABLES

- Variables are identifiers which represent some unknown, or variable-value.
- A variable is named storage (some memory address’s contents)

```
x = a + b;
Speed_Limit = 90;
```

TYPE <Variable Name> ;

Examples:

int marks; **double Pi**; **char grade;**

Programming Language Concepts
Variables, Names and Bindings

VARIABLES

- Variable names are case sensitive in C++.

<u>Variable valid names (C/C++)</u>

- Start with a letter
- Contains letters
- Contains digits
- Contains undersocre

Programming Language Concepts
Variables, Names and Bindings

VARIABLES

- Variable names are case sensitive in C++.

Variable valid names (JavaScript)

- Start with a **letter**, an **underscore** or a **dollar** sign
- Cannot contain **spaces**.
- Cannot be the same as **reserve keywords**.
- By convention, JavaScript variable names are written in **camelCase**.
- Names should be **descriptive** that indicate their content and usage e.g. **sellingPrice** and **costPrice** instead of **x** and **y**.
- JavaScript variables do not have **set types**.

Programming Language Concepts
Variables, Names and Bindings

VARIABLE NAMES

- Variable Names or Identifiers are essential in programming languages - we use them to identify the virtual entities that we manipulate in programs.

Choose meaningful names
- Don’t use abbreviations and acronyms: mtbf, TLA, myw, nbv

Don't use overly long names
- **Ok:**

  partial_sum
  element_count
  staple_partition

- **Too long (valid but not good practice):**

  remaining_free_slots_in_the_symbol_table

Programming Language Concepts
Variables, Names and Bindings

VARIABLE NAMES

- When we declare a variable, what happens ?
  - Memory allocation
    - How much memory? (*data type*)

- Memory associated with a name (variable name)
- The allocated space has a unique **address**

int Marks; →

Marks

`%$^%$%$*^%`

FE07

Programming Language Concepts
Variables, Names and Bindings

VARIABLES

- Variables are reference to a memory location whose contents may change. names are case sensitive in C++.
- Now generalized as "a placeholder for a value of some possibly complex type"
- e.g. in functional languages variables can store closures of arbitrary higher-order types like ((int → int) → int). This is a semantic concept not only memory location.
- Some languages allow you to access the memory location information like in C/C++ .

Programming Language Concepts
Variables, Names and Bindings

VARIABLES/MEMORY ADDRESSES

When a variable is declared, a specific memory is assigned to that variable based on its type and the variable's data is stored there

```
int a = 100;
```

In example above a variable a is declared as a integer and a memory slot of 4 bytes is assigned to the variable a.

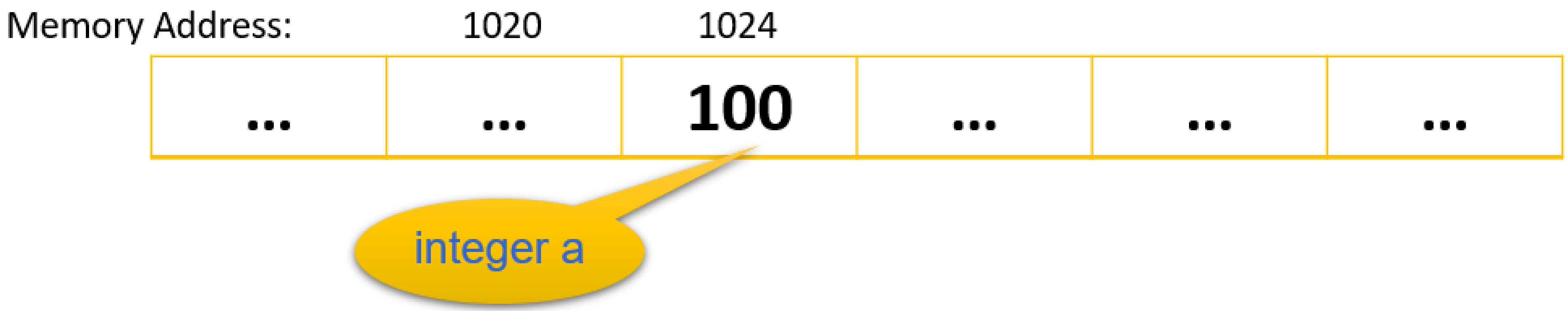
Programming Language Concepts
Variables, Names and Bindings

VARIABLES/MEMORY ADDRESSES

Each variable is assigned a memory slot (the size depends on the data type) and the variable's data is stored there

int a = 100;

Variable a's value 100 is stored at location 1024 as shown below:



Programming Language Concepts
Variables, Names and Bindings

- **Variables/Memory Addresses**

A variable holds a value and that value is stored at a specific location in memory.

The memory assigned to a variable has a memory address. That memory address can be accessed using **ampersand &** operator.

```
#include <iostream>
using namespace std;
int main()
{ int var1 = 3;
int var2 = 15;
int var3 = 29;
cout<<&var1<<endl;
cout<<&var2<<endl;
cout<<&var3<<endl;
return 0; }
```

Output

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

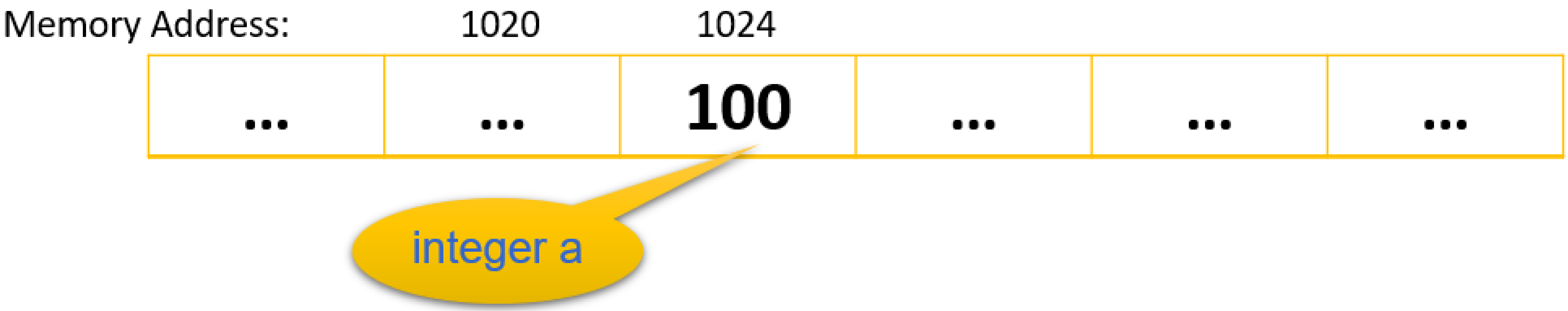
Programming Language Concepts
Variables, Names and Bindings

VARIABLES/MEMORY ADDRESSES

Address Operator

- The ‘*address of*’ operator (&) gives the memory address of the variable.

```
int a = 100;          //get the value
cout << a;            //prints the value 100
cout << &a;           //get the memory address and prints 1024
```

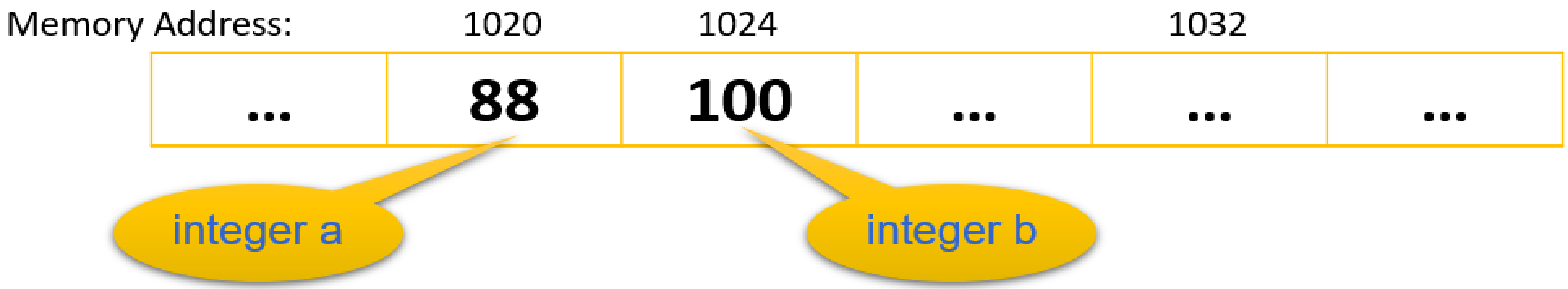


Programming Language Concepts
Variables, Names and Bindings

VARIABLES/MEMORY ADDRESSES

Address Operator

```
int a, b;
a = 88;                        //get the value of a
b = 100;                       //get the value of b
cout << “The address of a is = ”<< &a << endl;       //prints the address 1020
cout << “The address of b is = ”<< &b << endl;       //prints the address 1024
```



Programming Language Concepts
Variables, Names and Bindings

VARIABLES

A variable typically has six attributes associated with it.

- A **name**
- An **address** (aka an L-Value, i.e. the left-hand side of an assignment)
- A **value** (aka an R-Value, i.e. the right-hand side of an assignment)
- A **type**
- An **extent**
- A **scope**

Programming Language Concepts
Variables, Names and Bindings

VARIABLE TYPES

There are typically four types of variables.

Static Variables or Global Variables

- A variable bound to a memory location at initialization time.
- e.g. Static class variables in Java are static variables

Stack Dynamic Variables

- Memory is allocated from a runtime stack and bound when a declaration is executed and deallocated when the procedure block it is contained in returns.
- e.g. Local variables in a method declaration

Programming Language Concepts
Variables, Names and Bindings

VARIABLE TYPES

There are typically four types of variables.

Static Variables or Global Variables

```
class Student {
public:
static int noOfStudents;
};
int Student::noOfStudents;
int main() {
Student aStudent;
aStudent.noOfStudents = 1;
Student::noOfStudents = 1;
}
```

Programming Language Concepts
Variables, Names and Bindings

VARIABLE TYPES

There are typically four types of variables.

Accessing Static Variables or Global Variables

To access a static data member there are two ways

Access like a normal data member

Access using a scope resolution operator "::"

```
Student aStudent;
aStudent.noOfStudents = 1;
Student::noOfStudents = 1;
```

Programming Language Concepts
Variables, Names and Bindings

VARIABLE TYPES

There are typically four types of variables.

Explicit Heap-Dynamic Variables

- Nameless abstract memory locations that are allocated/deallocated by explicit runtime commands by the programmer.
- e.g. malloc/free in C, new/delete in C++, all objects in Java using new()

Implicit Heap-Dynamic Variables

- Memory in heap is allocated when variables are assigned to values. It is deallocated and reallocated with re-assignment. Error prone and inefficient.
- Used in ALGOL 68, LISP, C and JavaScript (for arrays).

Programming Language Concepts
Variables, Names and Bindings

BINDING TYPES

There are two types of bindings.

Static Type Binding

Two approaches to static type binding are:

- Type Declaration
- Type Inference

Type Declaration

- Most commonly used approach (used in Algol, Pascal, Ada, Cobol, C/C++, Java)
- A variable is introduced with an explicit type and possibly an initial value.

Programming Language Concepts
Variables, Names and Bindings

BINDING TYPES

There are two types of bindings.

Static Type Binding

Types determined at compile time
And doesn't change during runtime
(e.g. Java)

- Two approaches to static type binding are:
- Type Declaration
- Type Inference

Type Inference

- No types in variable declarations; the type is inferred from the usage of the variable or by following a fixed naming scheme.
- Primitive type inference (arguably another form of explicit declaration) - e.g. in Fortran I, J, K, L, M and N are Integer types, otherwise Real assumed. In Perl $p is a number or a string, @p an array, %p a hash.

Programming Language Concepts
Variables, Names and Bindings

BINDING TYPES

There are two types of bindings.

Dynamic Type Binding

Dynamic binding typically occurs as a variable is assigned a value at runtime.

- A variables type binding can change during execution simply by assigning to it a value of a different type.
- Commonly used in scripting languages such as JavaScript, Lua, Perl, PHP, Python, Ruby
- Efficiency implications (both time and space) due to runtime type checking
- Arguably advantages in readability and coding convenience.

Programming Language Concepts

Dr. Imran

Lecture 3

Programming Language Concepts
Variables, Names and Bindings

EXTENT

- The **extent (or lifetime)** of a variable refers to the periods of execution time for which it is bound to a particular location storing a meaningful value

- Extent is a semantic concept and depends on the execution model.

Programming Language Concepts
Variables, Names and Bindings

EXTENT

Different kinds of variables have different extents:

Static variables

- Have an extent of whole program execution
- They are created even when there is no object of a class
- They remain in memory even when all objects of a class are destroyed

Programming Language Concepts
Variables, Names and Bindings

EXTENT

Different kinds of variables have different extents:

- **Stack-dynamic variables** have an extent of a particular stack frame or procedure call
- **Explicit heap-dynamic variables** have an extent from explicit allocation to explicit deallocation (cf. garbage collection and memory leak)
- **Implicit heap-dynamic variables** have an extent from implicit allocation to implicit deallocation (values may persist in memory but addresses are freed)

Programming Language Concepts
Variables, Names and Bindings

SCOPE

- The scope of a variable is the part of the code in which it can be referenced.
- Alternatively, it is the part of a program where a variable’s name is meaningful.
- A variable’s scope affects its extent. A no-longer referenceable value may be considered as a meaningless value. Garbage collectors are based on this principle.
- Local variables are declared within a program block; the block is the scope of the variable.
- Static variables have whole program scope, except where they are temporarily hidden by a locally scoped variable with the same name.

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in ECMAScript

```
// Global scope
var printOne = 1;
function one(){
alert(a);
}
one()
outputs ‘1’
```

Programming Language Concepts

Variables, Names and Bindings

SCOPE

Scope in ECMAScript

```
// Local scope
function two(a){
alert(a);
}
two(2)
outputs '2'
```

```
// Local scope
function three(){
var a = 3;
alert(a);
}
three()
outputs '3'
```

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in ECMAScript

```
// No block scope
function four(){
    if true {
        var a = 4;
    }
    alert(a);
}
four()
outputs ‘4’
```

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in C++

Global Variable

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

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Local variable

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in C++

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

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
}

int main()
{
    cout<<"Age is: "<<age;

    return 0;
}
```

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in C++

```
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}

int main()
{
    cout<<"Age is: ";
    func();

    return 0;
}
```

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in C++

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

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
}

// main function
int main()
{
    display();

    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

Programming Language Concepts
Variables, Names and Bindings

SCOPE

Scope in C++

```
using namespace std;

// Global x
int x = 0;

int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```