Lecture 12

M M Imran

Type and Storage Class

- Every variable in C++ has two features: type and storage class.
- Type specifies the type of data that can be stored in a variable.
  - For example: int, float, char etc.
- Storage class controls two different properties of a variable:
  - Lifetime (determines how long a variable can exist)
  - Scope (determines which part of the program can access it).

Types

Depending upon the storage class of a variable, it can be divided into several types:

- Local variable
- Global variable
- Static local variable
- Register Variable
- Thread Local Storage

Local Variable

- A variable defined inside a function (defined inside function body between braces) is called a local variable or automatic variable.

- Its scope is only limited to the function where it is defined. In simple terms, local variable exists and can be accessed only inside a function.

- The life of a local variable ends (It is destroyed) when the function exits.

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

void test();

int main()
{
    // local variable to main()
    int var = 5;

    test();

    // illegal: var1 not declared inside main()
    var1 = 9;
}

void test()
{
    // local variable to test()
    int var1;
    var1 = 6;

    // illegal: var not declared inside test()
    cout << var;
}
```

The variable **var** cannot be used inside **test()** and **var1** cannot be used inside **main()** function.

Global Variable

- If a variable is defined outside all functions, then it is called a global variable.
- The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration.
- Likewise, its life ends only when the program ends.

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

// Global variable declaration
int c = 12;

void test();

int main()
{
    ++c;

    // Outputs 13
    cout << c <<endl;
    test();

    return 0;
}

void test()
{
    ++c;

    // Outputs 14
    cout << c;
}
```

Output

```
13
14
```

In the above program, **c** is a global variable.

This variable is visible to both functions **main()** and **test()** in the above program.

Static Local variable

- Keyword static is used for specifying a static variable. For example:

```
int main()
{
  static float a;
  ... .. ...
}
```

- A static local variable exists only inside a function where it is declared (similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.
- The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

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

void test()
{
    // var is a static variable
    static int var = 0;
    ++var;

    cout << var << endl;
}

int main()
{

    test();
    test();

    return 0;
}
```

Output

```
1
2
```

- In the above program, test() function is invoked 2 times.
- During the first call, variable var is declared as static variable and initialized to 0. Then 1 is added to var which is displayed in the screen.
- When the function test() returns, variable var still exists because it is a static variable.
- During second function call, no new variable var is created. The same var is increased by 1 and then displayed to the screen.

Register Variable (Deprecated in C++11)

- Keyword register is used for specifying register variables.
- Register variables are similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables.
- If a program encounters a register variable, it stores the variable in processor's register rather than memory if available. This makes it faster than the local variables.
- However, this keyword was deprecated in C++11 and should not be used.

Thread Local Storage

- Thread-local storage is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread.

- Keyword thread_local is used for this purpose.

Variable Scope Overview

- Scope indicates the extent that a variable may be referenced.

- There are at least two levels of scope:
  - Local – these include any parameters and variables declared within a function.
  - Global – these include any variables declared outside any function.

- Global variable declarations should be placed near the top of a program, above or below any function prototypes.

Variable Scope Overview

- A global variable:
  - Has an advantage in that its value does not to be passed into functions since it is already available to them.
  - Has a disadvantage in that any part of an application can change its value. This makes an application harder to read and debug.

- The use of global variables should be limited.
- The use of global constants, however, is fine.

Variable Scope Overview

Variable type	Scope	Lifetime
Automatic local	Within function declared.	• Created when function starts. • Destroyed when function ends.
Static local	Within function declared.	• Created when function starts first time. • Destroyed when application ends.
Global	Within application declared.	• Created when application starts. • Destroyed when application ends.

Scope Resolution Operator

- Although it may be done, it is not a good programming practice to use the same names for a global variable and a local variable.

- By default, a local variable with the same name as a global variable overrides or hides the global variable.

- A global variable with the same name as a local variable may be accessed in the local variable's scope by preceding the global variable with **scope resolution operator ::**

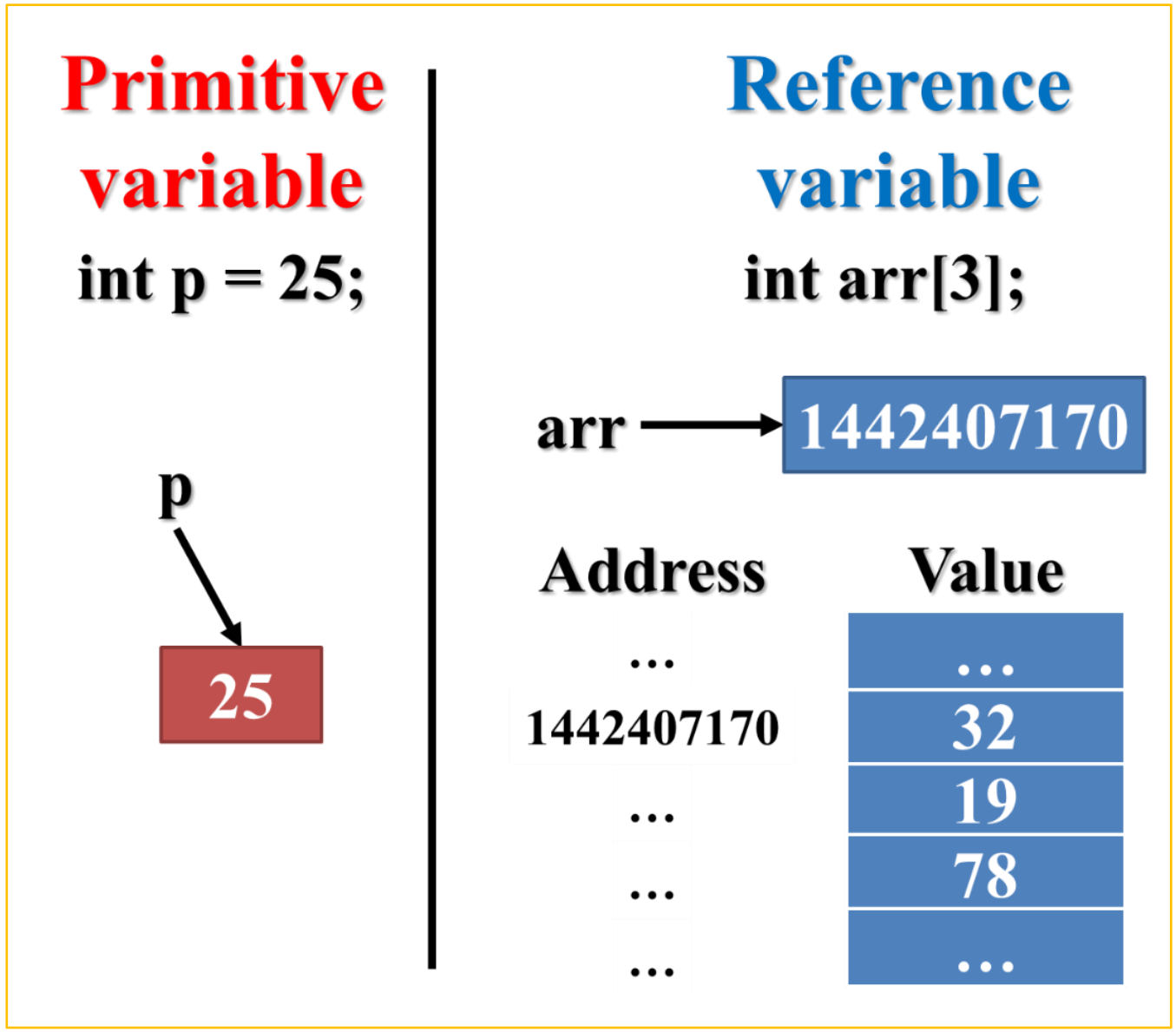
Local and Global variable example:

```
…
int year;  // Declare global variable
…
int main()
{
      …
        int year;  // Declare local variable
      …
        year = 2020;  // Reference local variable
      …
        ::year = 2020;  // Reference global variable
      …
}
```

Value Type and Reference Type

- When an argument is matched with a parameter, a value is copied from the argument to the parameter.

- There are two kinds of variables:
  - A primitive variable contains a value.
  - A reference variable also contains a value but the value is a memory address that points to a location in memory that contains the variable's value(s).



p is a primitive variable that points directly to a value.

Primitive Variable Chart

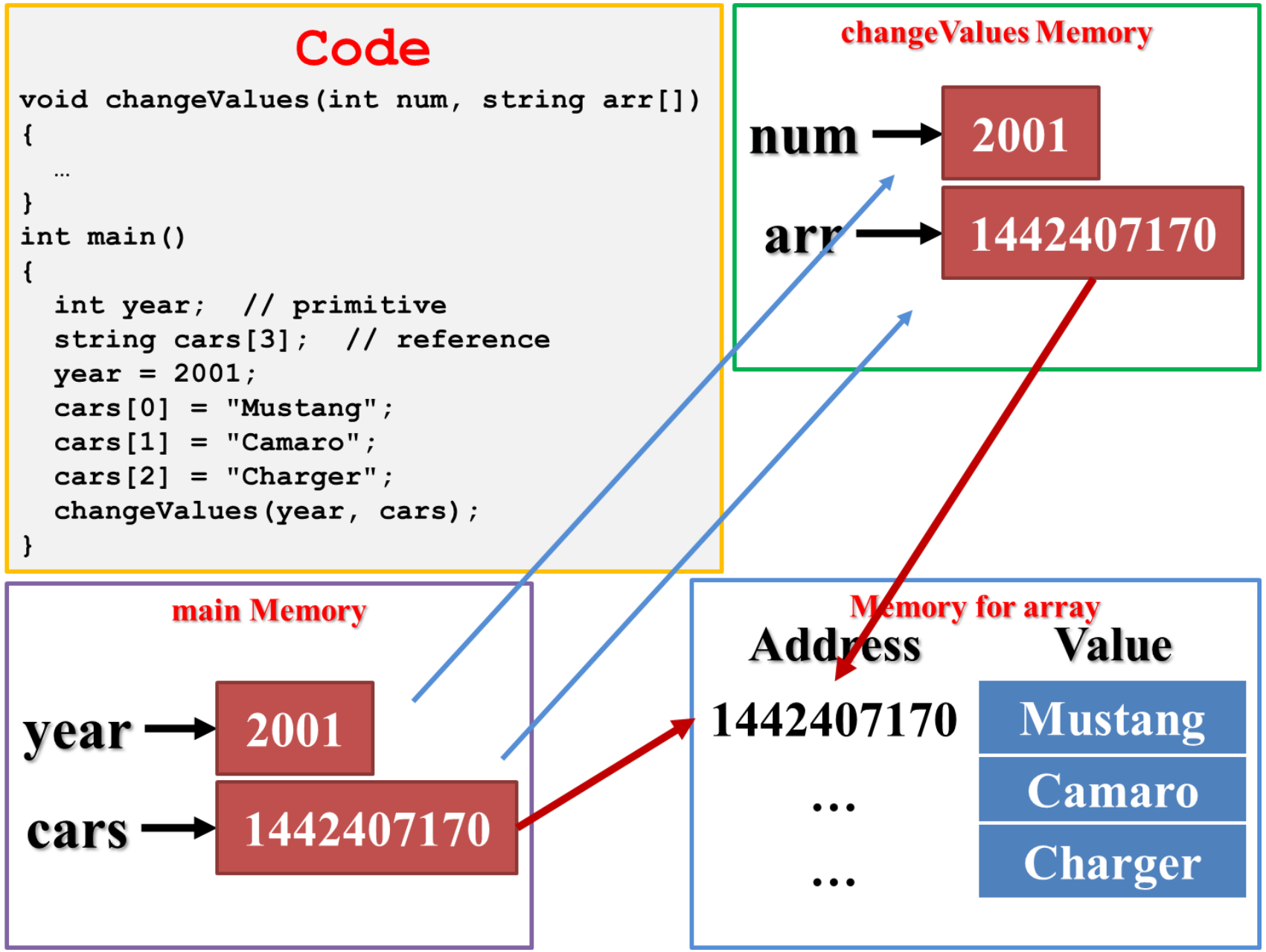
Data type	Bytes	Range
Integer types		
char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	0 to 18,446,744,073,709,551,615
Real-number types		
float	4	-3.4^{38} to -1.2^{-38}, 0, 1.2^{-38} to 3.4^{38}
double	8	-1.8^{308} to -2.2^{-308}, 0, 2.2^{-308} to 1.8^{308}
long double	8	-1.8^{308} to -2.2^{-308}, 0, 2.2^{-308} to 1.8^{308}
Other types		
bool	1	true or false
char	1	A single character, enclosed by single quotes
string	28	A sequence of characters, enclosed by double quotes

Reference Variable

In the previous example, **arr** is a reference variable (an array in this example) that points to a memory address containing data of one of the above data types or a programmer-defined data type (discussed later).

Pass by value and reference

- Sometimes an argument is a variable.
- If the argument variable is:
  - A primitive variable, its value gets copied to the corresponding parameter. This is called **pass by value**. In the function, any value changes made to the parameter are not carried outside to the argument variable.
  - A reference variable, its memory address gets copied to the corresponding parameter. This is called **pass by reference**. In the function, any value changes made to the parameter are carried outside to the argument variable.


Code
void changeValues(int num, string arr[])
{
…
}
int main()
{
int year; // primitive
string cars[3]; // reference
year = 2001;
cars[0] = "Mustang";
cars[1] = "Camaro";
cars[2] = "Charger";
changeValues(year, cars);
}
changeValues Memory
num
2001
arr
1442407170
main Memory
year
2001
cars
1442407170
Memory for array
Address
Value
1442407170
Mustang
Camaro
Charger
…
…

Pass by value

```cpp
#include <iostream>

using namespace std;

void change(int data);

int main() {
    int data = 3;
    change(data);
    cout << "Value of the data is: " << data << endl;
    return 0;
}

void change(int data) {
    data = 5;
}
```

Output:

```
Value of the data is: 3
```

Pass by reference

```
#include<iostream>

using namespace std;

void swap(int * x, int * y) {
    int swap;
    swap = * x;
    * x = * y;
    * y = swap;
}

int main() {
    int x = 500, y = 100;
    swap( & x, & y); // passing value to function
    cout << "Value of x is: " << x << endl;
    cout << "Value of y is: " << y << endl;
    return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between pass by value and reference

Pass by value	Pass by reference
A copy of value is passed to the function	An address of value is passed to the function
Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location