CS118 – Programming Fundamentals

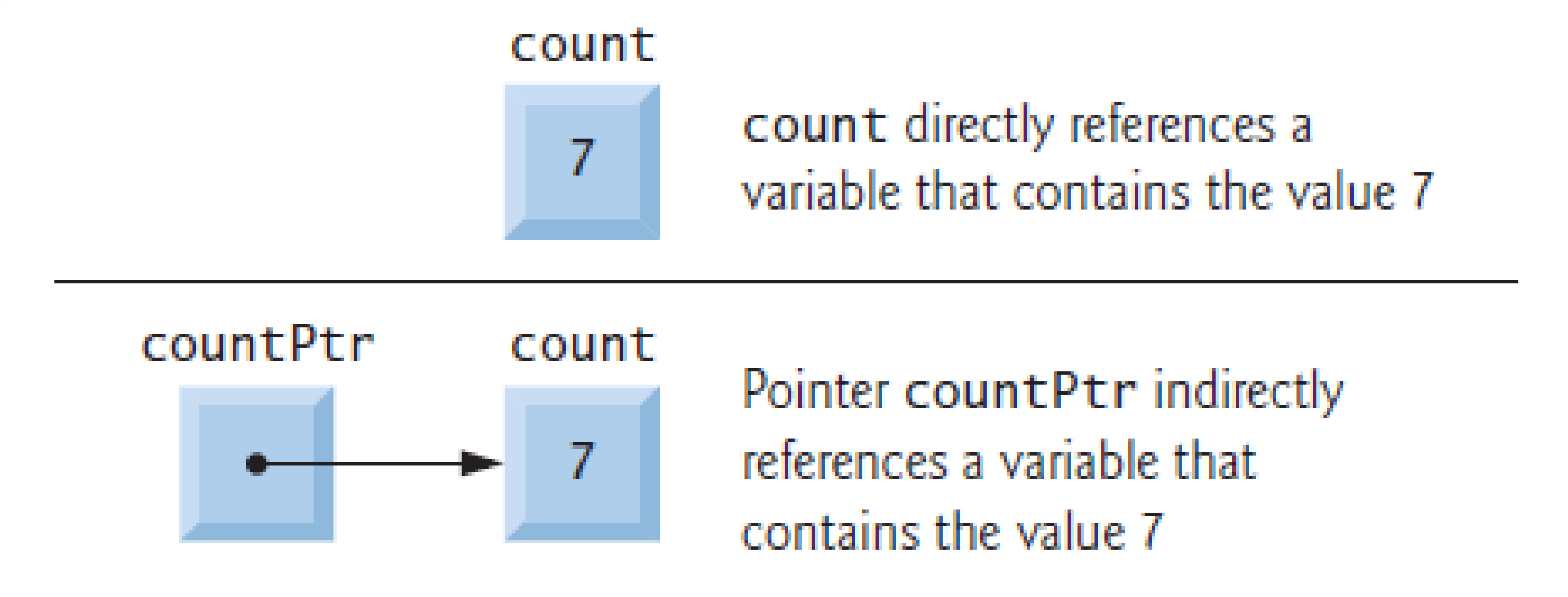
Lecture # 27
Tuesday, December 03, 2019
FALL 2019
FAST – NUCES, Faisalabad Campus

Zain Iqbal

What is a Pointer variable?

- Pointer variables contain ***memory addresses*** as their values
- Normally, a variable directly contains a specific value
- A pointer variable contains the address of the location that contains the specific value
- A variable is a **direct reference** to a value
- A pointer is an **indirect reference** to a value
  - Referencing a value through a pointer is known as **indirection**

Pointer Elaborated



count directly references a variable that contains the value 7

Pointer countPtr indirectly references a variable that contains the value 7

Fig. 8.1. | Directly and Indirectly referencing a variable

Declaring a pointer

- Pointer **must** be declared before they can be used
- Syntax to declare a pointer is

  data_type *identifier;
- E.g.

```
int *iPtr;
double *dPtr;
```

- iPtr and dPtr are pointers to integer and double respectively.
  - We read from right to left i.e. iPtr is a pointer to integer.
- What if we declare

```
int *iPtr, count;
```

Note: * applies only to **iPtr**. Count is not a pointer. It is a simple integer

- So two pointers must be declared as

```
int *iPtr, *countPtr;
```

Pointer initialization

- Pointers must be initialized to 0, NULL or an address of corresponding type
  - Either in declaration or in assignment
- A pointer with value 0 or NULL "**points to nothing**" and is known as **null pointer**
- In the new standard, you should use the constant **nullptr** to initialize a pointer instead of 0 or NULL
  - int *iPtr = 0;
  - int *iPtr = NULL;
  - int iPtr = nullptr;

Pointer to a variable

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

- The statement

```
yPtr = &y; // assign address of y to yPtr
```

- Assigns the value of variable y to yPtr
  - **yPtr** is said to point to **y**
  - Now **yPtr** is *indirectly* references **y's** value

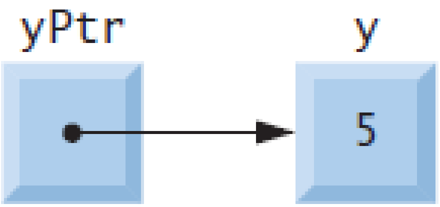


Fig. 8.2 | Graphical representation of a pointer pointing to a variable

Pointer in Memory

- Figure 8.3 shows another pointer representation in memory with integer variable y stored at memory location 600000 and pointer variable yPtr stored at memory location 500000. The operand of the address operator must be an lvalue; the address operator cannot be applied to constants or to expressions that do not result in references.

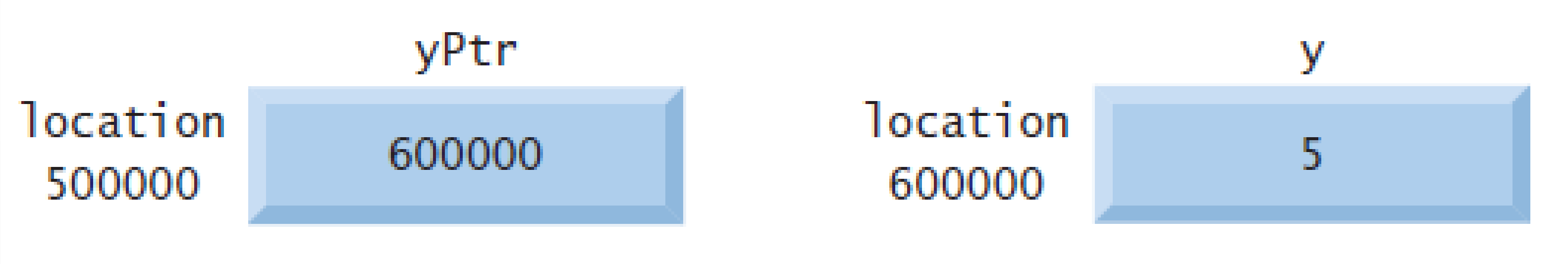


Fig. 8.3 | Representation of y and yPtr in memory.

Dereferencing a Pointer

- * **operator**, commonly referred to as the *indirection operator* or *dereferencing operator*, returns a synonym (i.e., an alias or a nickname) for the object to which its pointer operand points. For example (referring again to Fig. 8.2), the statement
- cout << *yPtr << endl;
- prints the value of variable y, namely, 5, just as the statement
- cout << y << endl;
- would. Using * in this manner is called dereferencing a pointer

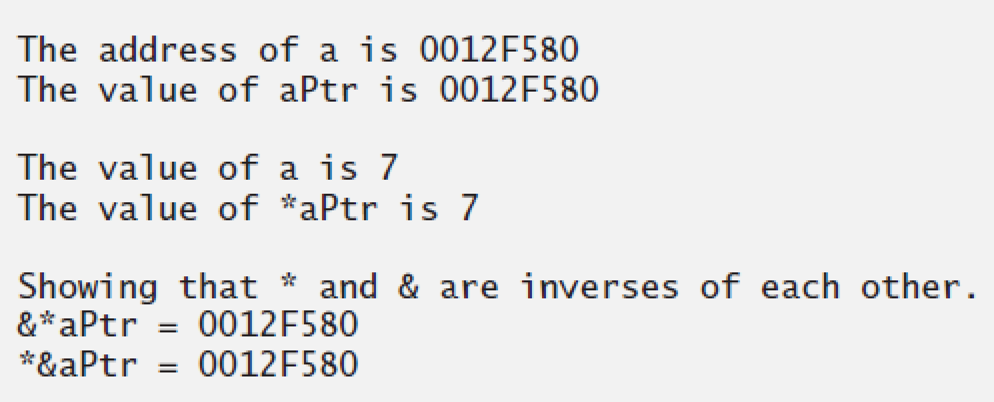
Dereferencing a Pointer

- A dereferenced pointer may also be used on the *left* side of an assignment statement, as in
- *yPtr = 9;
- which would assign 9 to y in Fig. 8.3. The dereferenced pointer may also be used to receive an input value as in
- cin >> *yPtr;
- which places the input value in y. The dereferenced pointer is an *lvalue*

- **Note:** The & and * operators are inverses of one another

Sample Program – I

```cpp
// Fig. 8.4: fig08_04.cpp
// Pointer operators & and *.
#include <iostream>
using namespace std;

int main()
{
   int a; // a is an integer
   int *aPtr; // aPtr is an int * which is a pointer to an integer

   a = 7; // assigned 7 to a
   aPtr = &a; // assign the address of a to aPtr

   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr ;
   cout << "\n\nThe value of a is " << a
        << "\nThe value of *aPtr is " << *aPtr;
   cout << "\n\nShowing that * and & are inverses of "
        << "each other.\n&*aPtr = " << &*aPtr
        << "\n*&aPtr = " << *&aPtr << endl;
} // end main
```



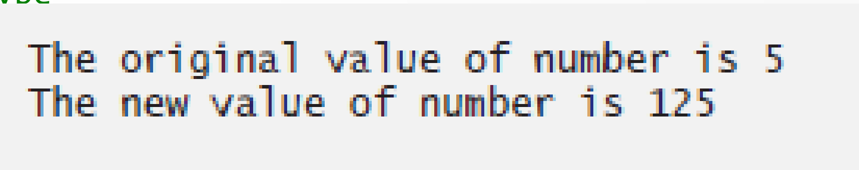
```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Pass by reference with pointers

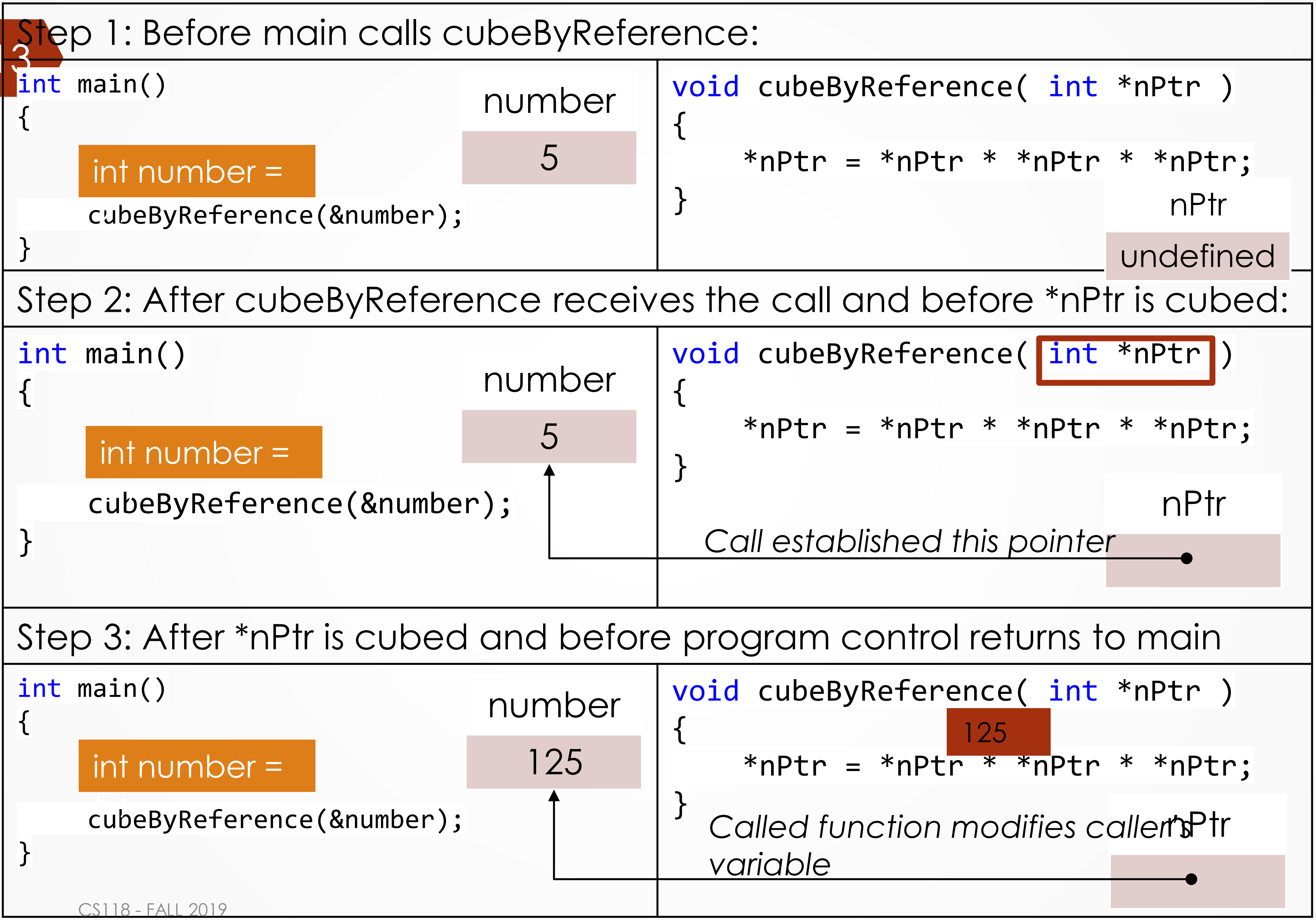
- There are three ways to pass an argument to a function
  - Pass-by-Value
  - Pass-by-reference with reference argument
  - Pass-by-reference with pointer argument
- Pointers, like references,
  - can be used to modify one or more variables in the caller
  - to pass pointers to large data objects to avoid the overhead of passing the objects by value

Pass-by-Reference with pointers

```cpp
// Fig. 8.7: fig08_07.cpp
// Pass-by-Reference used to cube a variable’s value.
#include <iostream>
using namespace std;

int cubeByReference ( int *); // prototype

int main()
{
  int number = 5;

  cout << "The original value of number is " << number;

  cubeByReference( &number ); // pass number address to cubeByReference
  cout << "\nThe new value of number is " << number << endl;
} // end main

// calculate the cube of *nPtr; modifies the variable in main
int cubeByReference( int* nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```



```
The original value of number is 5
The new value of number is 125
```



Step 1: Before main calls cubeByReference:

```
int main()
{
    int number =
    cubeByReference(&number);
}
```

number

5

```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

undefined

Step 2: After cubeByReference receives the call and before *nPtr is cubed:

```
int main()
{
    int number =
    cubeByReference(&number);
}
```

number

5

```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

Call established this pointer

Step 3: After *nPtr is cubed and before program control returns to main

```
int main()
{
    int number =
    cubeByReference(&number);
}
```

number

125

```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

125

nPtr

Called function modifies caller's variable

Questions