Pointers

Dr Bhanu

Introduction

- Pointers are variables whose values are memory addresses.
- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an **address** of a variable that contains a specific value.
- In this sense, a variable name directly references a value, and a pointer indirectly references a value (Fig. 7.1).
- Referencing a value through a pointer is called indirection.

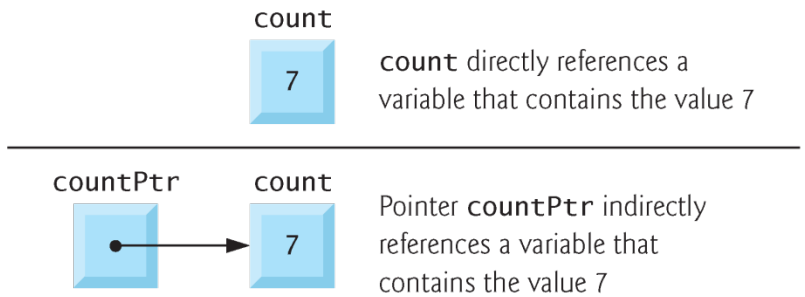


Fig. 7.1 | Directly and indirectly referencing a variable.

Pointer Variable Definitions and Initialization

- Pointers, like all variables, must be defined before they can be used.
- The definition
  - `int *countPtr, count;`

  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer). Also, the variable `count` is defined to be an `int`, not a pointer to an `int`.
- The * only applies to `countPtr` in the definition.
- When * is used in this manner in a definition, it indicates that the variable being defined is a pointer.
- Pointers can be defined to point to objects of any type.
- Pointers should be initialized either when they're defined or in an assignment statement.

Common Programming Error 7.1

*The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare* `xPtr` *and* `yPtr` *as* `int` *pointers, use* `int *xPtr, *yPtr;`.

Common Programming Error 7.2

Include the letters `ptr` *in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.*

Error-Prevention Tip 7.1

Initialize pointers to prevent unexpected results.

Pointer Operators

- The &, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement
  - `yPtr = &y;`

  assigns the address of the variable `y` to pointer variable `yPtr`.
- Variable `yPtr` is then said to "point to" `y`.
- Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.
    ```

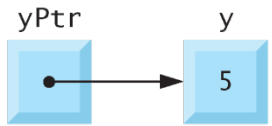


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Pointer Variable Definitions and Initialization

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points.
- For example, the statement
  - `printf( "%d", *yPtr );`

  prints the value of variable `y`, namely 5.
- Using * in this manner is called dereferencing a pointer.
- The `printf` conversion specifier `%p` outputs the memory location as a hexadecimal integer on most platforms.
- The & and * operators are complements of one another

```
#include<stdio.h>

int main()
{
int y = 5;
int *yptr;

 yptr = &y;

printf("yptr  = %p\n", yptr);

printf("y = %d\n",y);
printf("y through pointer= %d\n", *yptr);

return 0;
}
```

Common Programming Error 7.3

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

Operators	Associativity	Type
`()` `[]`	left to right	highest
`+` `-` `++` `--` `!` `*` `&` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.5 | Operator precedence and associativity.

Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—call-by-value and call-by-reference.
- All arguments in C are passed by value.
- Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the overhead of making a copy of the object.
- In C, you use pointers and the indirection operator to simulate call-by-reference.

Passing Arguments to Functions by Reference

- When calling a function with arguments that should be modified, the addresses of the arguments are passed.
- This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- Arrays are not passed using operator & because C automatically passes the starting location in memory of the array (the name of an array).
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

Passing Arguments to Functions by Reference

- The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—`cubeBV` and `cubeBR`.
- Figure 7.6 passes the variable `number` to function `cubeBV` using call-by-value
- The `cubeBV` function cubes its argument and passes the new value back to `main` using a `return` statement.
- The new value is assigned to `number` in `main`

Fig. 7.6

```
#include<stdio.h>

int cubeBV( int n);

int main()
{
   int number = 5, nq;

   nq = cubeBV( number);

   printf("number cubed  = %d\n", nq);

   return 0;
}

int cubeBV( int n)
{
   return n * n * n;
}
```

Fig. 7.7

```
#include<stdio.h>

void cubeBR( int *nptr);

int main()
{
   int number = 5;

 cubeBR( &number);

   printf("number cubed  = %d\n", number);

   return 0;
}

void cubeBR(  int *nptr)
{
   *nptr = *nptr * *nptr * *nptr;
   return ;
}
```

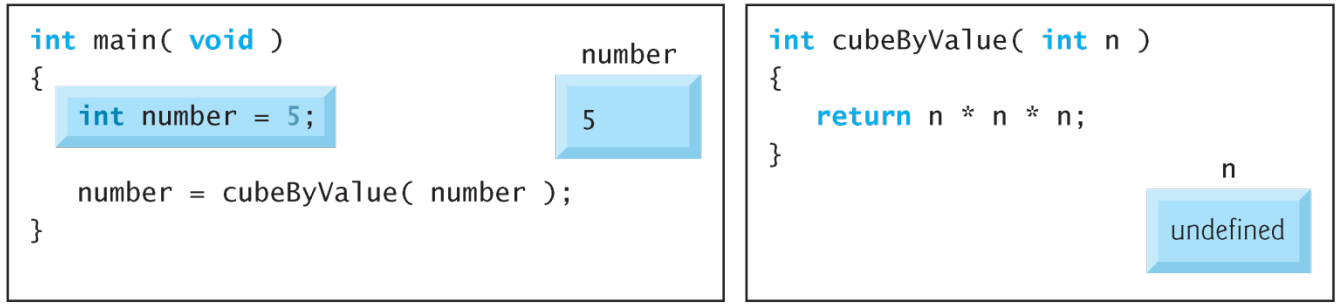
Passing Arguments to Functions by Reference

- Figure 7.7 passes the variable `number` using call-by-reference - the address of `number` is passed—to function `cubeBR`.
- Function `cubeBR` takes as a parameter a pointer to an `int` called `nPtr`
- The function dereferences the pointer and cubes the value to which `nPtr` points, then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`.
- Figure 7.8 and Fig. 7.9 analyze graphically the programs in Fig. 7.6 and Fig. 7.7, respectively.

Passing Arguments to Functions by Reference

- A function receiving an address as an argument must define a pointer parameter to receive the address.
- For example, in Fig. 7.7 the header for function `cubeBR` is:
  - `void cubeBR( int *nPtr )`
- The header specifies that `cubeBR` receives the address of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.
- The function prototype for `cubeBR` contains `int *` in parentheses.

Step 1: Before `main` calls `cubeByValue`:



Step 2: After `cubeByValue` receives the call:

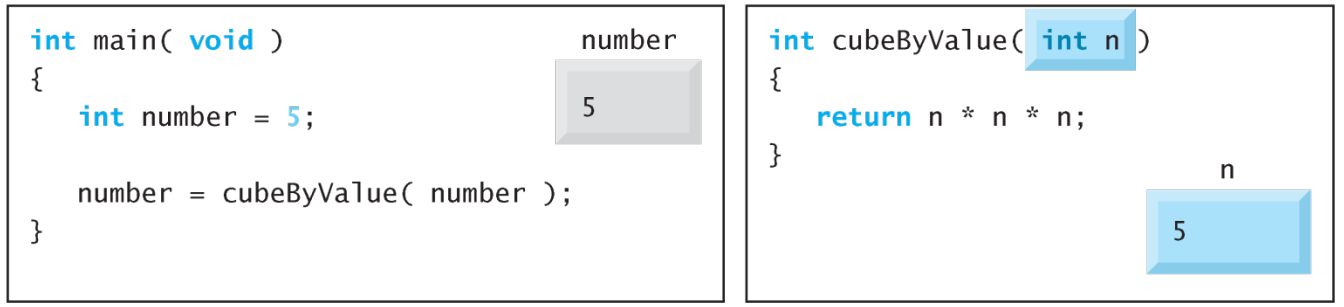
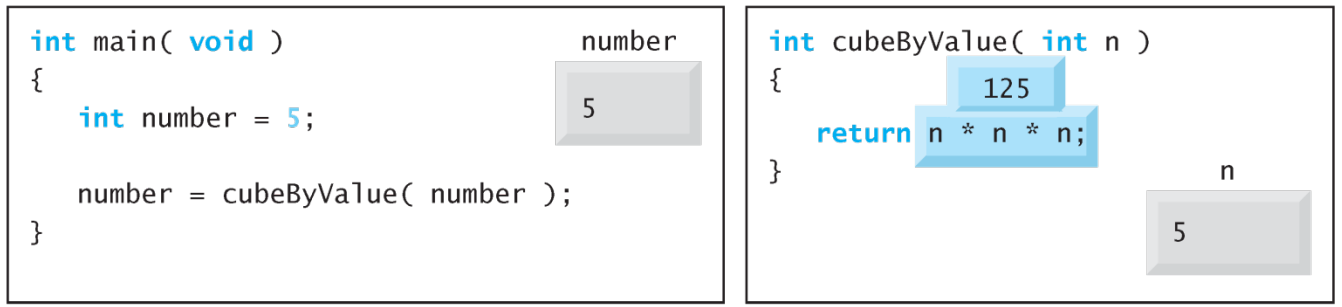


Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

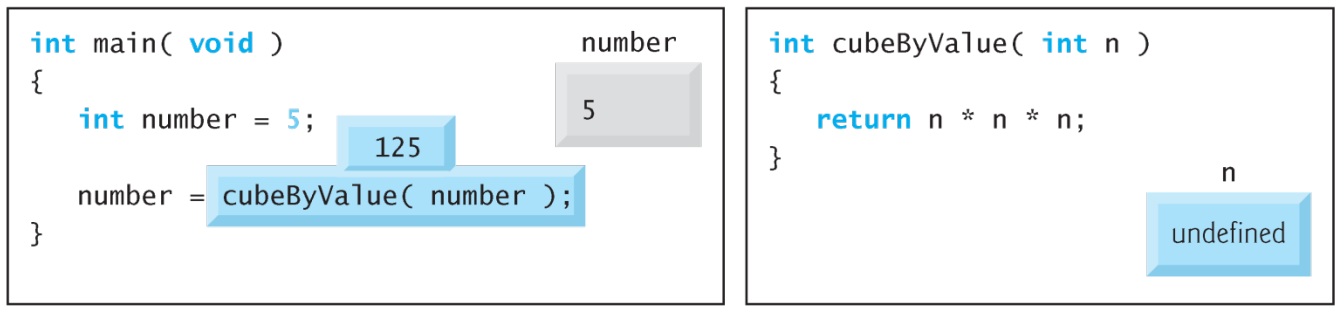


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Step 5: After `main` completes the assignment to `number`:

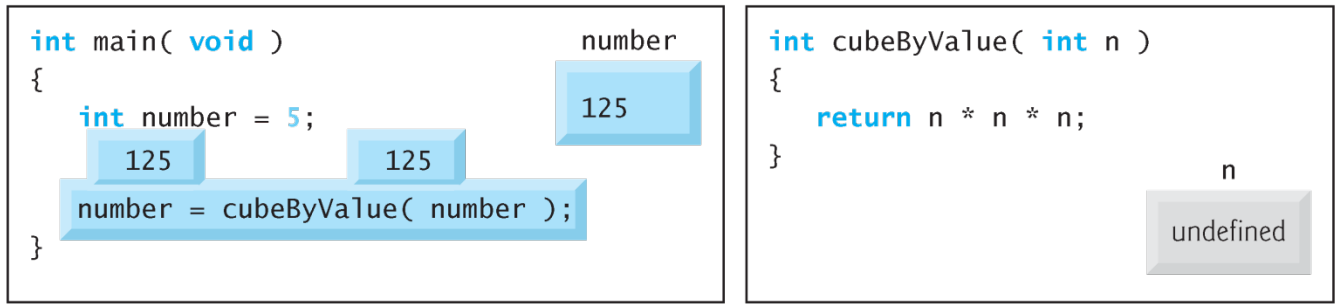


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Step 1: Before `main` calls `cubeByReference`:

```
int main( void )
{
   int number = 5;

   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( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

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

call establishes this pointer

nPtr

Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

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

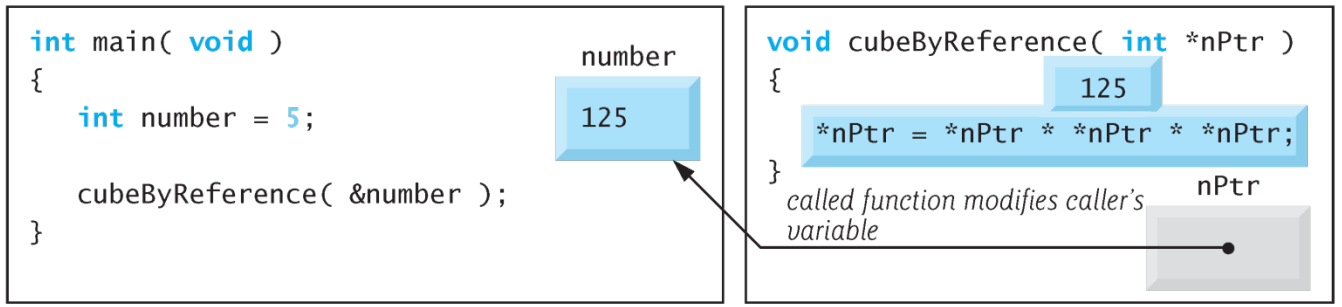


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Parameters	Call by value	Call by reference
Definition	While calling a function, when you pass values by copying variables, it is known as "Call By Value."	While calling a function, when the address of the variables is passed, it is known as "Call By Reference.
Arguments	A copy of the variable is passed.	Address of variable is passed.
Effect	Changes made in a copy of variable never modify the value of variable outside the function.	Change in the variable also affects the value of the variable outside the function.
Alteration of value	Does not allow you to make any changes in the actual variables.	Allows you to make changes in the values of variables by using function calls.
Passing of variable	Values of variables are passed using a straightforward method.	Pointer variables are required to store the address of variables.
Value modification	Original value not modified.	The original value is modified.
Memory Location	Actual and formal arguments will be created in different memory locations	Actual and formal arguments will be created in the same memory location
Safety	Actual arguments remain safe as they cannot be modified accidentally.	Actual arguments are not Safe. They can be modified, so you need to handle arguments operations carefully.

Passing Arguments to Functions by Reference – Swapping two numbers using Pointers

```
void swap (int *aptr, int *bptr);
int main()
{
  int m = 25;
  int n = 100;
  printf("*** Original values ***\n");
  printf("m is %d, n is %d\n", m, n);
  swap(&m, &n);
    printf("*** Modified values ***\n");
  printf("m is %d, n is %d\n", m, n);
  return 0;
}
void swap (int *aptr, int *bptr)
{
  int temp;
  temp = *aptr;
  *aptr = *bptr;
  *bptr = temp;
    return;
}
```

Passing arrays to Functions

- The first way includes the most widely used technique that is declaring blank subscript notation [].

return_type function(type arrayname[])

- The second way is also a most widely used technique used optionally to the first way.
- It involves defining the size in subscript notation [ ].

return_type function(type arrayname[SIZE])

- The third way is basically a general method that includes the use of the concept of a pointer.

return_type function(type *arrayname)

```c
// Passing array as a Parameter to the function
 #include <stdio.h>

float arraySum(float age[]);

int main()
{
   float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
   result = arraySum(age);
   printf("Result = %.2f", result);
   return 0;
}

float arraySum(float age[])
{
   float sum = 0.0;
   for (int i = 0; i < 6; ++i)
    sum += age[i];

   return sum;
}
```

```
// Passing the base address of the array to the function

#include <stdio.h>

float arraySum(float *ageptr);

int main()
{
   float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
   result = arraySum(age);
   printf("Result = %.2f", result);
   return 0;
}

float arraySum(float *ageptr)
{
   float sum = 0.0;
    for (int i = 0; i < 6; ++i) {
       sum += *ageptr;
       ageptr++;
    }
 return sum;
}
```

```
#include<stdio.h>

int minArr(int arr[],int size)
{
   int min=arr[0];

   for(int i=1;i<size;i++)
   {
   if(arr[i] < min)
   min=arr[i];

   }
   return min;
}

int main()
{
   int i=0,min=0;
   int numbers[]={5,4,2,10,1,6};
   min=minArr(numbers,6);
   printf("Min no = %d \n",min);
   return 0;
}
```

Use of SIZEOF operator

```
int m, n, SIZE;
int x[] = {1, 2, 3};

m = sizeof(x);

printf("m = %d\n", m);
n = sizeof(x[1]);

printf("n = %d\n", n);

SIZE = m / n;
printf("SIZE = %d\n", SIZE);
```

Pointer arithmetic

```
#include <stdio.h>

int main()
{

   int a[] = {34, 67, 89, 12};
   int *aptr;

   aptr = &a;
   printf("aptr = %p\n", aptr);
   aptr++;
   printf("aptr = %p\n", aptr);

   char c[] = {'d', 'f', 't', 's'};
   char  *cptr;
   cptr = &c;
    printf("cptr = %p\n", cptr);
   cptr++;
   printf("cptr = %p", cptr);

   return 0;
}
```

Passing Multi-dimensional arrays to Functions

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
   int num[2][2];
   printf("Enter 4 numbers:\n");
   for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j)
         scanf("%d", &num[i][j]);

   displayNumbers(num);
   return 0;
}
```

```
void displayNumbers(int num[2][2])
{
   printf("Displaying:\n");
   for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 2; ++j) {
         printf("%d\n", num[i][j]);
      }
   }
}
```