CC-112 Programming Fundamentals

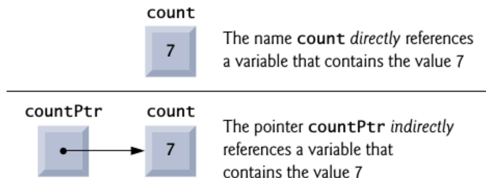
Pointers

Nazar Khan
Department of Computer Science
University of the Punjab

Pointer Variable Definitions and Initialization

- A pointer contains an *address* of another variable that contains a value.
- A variable name *directly references* a value, and a pointer *indirectly references* a value.

```c
int count=7;
int *countPtr;
```



- Referencing a value through a pointer is called *indirection*.

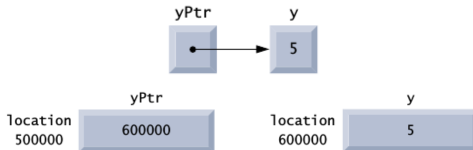
Pointer Variable Definitions and Initialization

- 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.
- A pointer may be initialized to NULL, 0 or an address.
- A pointer with the value NULL points to nothing.
- Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred for clarity. The value 0 is the only integer value that can be assigned directly to a pointer variable.
- NULL is a symbolic constant defined in the <stddef.h> header (and several other headers).

Pointer Operators

- The &, or *address operator*, is a unary operator that returns the address of its operand.

```
int y = 5;
int *yPtr;
yPtr = &y;
printf("%d", *yPtr); //prints value of y which is 5
printf("%p", yPtr);  //prints address of y as a hexadecimal integer
```



- The operand of the address operator must be a variable.
- The *indirection operator* * returns the value of the object to which its operand points.

Pointer Operators

```
1   // Using the & and * pointer operators.
2   #include <stdio.h>
3
4   int main(void)
5   {
6      int a = 7;
7      int *aPtr = &a; // set aPtr to the address of a
8
9      printf("The address of a is %p"
10            "\nThe value of aPtr is %p", &a, aPtr);
11
12     printf("\n\nThe value of a is %d"
13            "\nThe value of *aPtr is %d", a, *aPtr);
14
15     printf("\n\nShowing that * and & are complements of "
16            "each other\n&*aPtr = %p"
17            "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
18  }
```

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

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

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Passing Arguments to Functions by Reference

Passing by value

```
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void)
{
   int number = 5; // initialize number

   printf("Original value is %d", number);

   // pass number by value to cubeByValue
   number = cubeByValue(number);

   printf("\nNew value is %d\n", number);
}

// return cube of integer argument
int cubeByValue(int n)
{
   return n * n * n;
}
```

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

Passing by reference

```
// Cube a variable via pass-by-reference
// with pointer argument.
#include <stdio.h>

void cubeByReference(int *nPtr);

int main(void)
{
   int number = 5; // initialize number

   printf("Original value is %d", number);

   // pass address of number
   cubeByReference(&number);

   printf("\nNew value is %d\n", number);
}

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

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

Step 1: Before **main** calls **cubeByValue**:

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

   number = cubeByValue(number);
}
```

number: 5

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

n: undefined

Step 2: After **cubeByValue** receives the call:

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

   number = cubeByValue(number);
}
```

number: 5

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

n: 5

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

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

   number = cubeByValue(number);
}
```

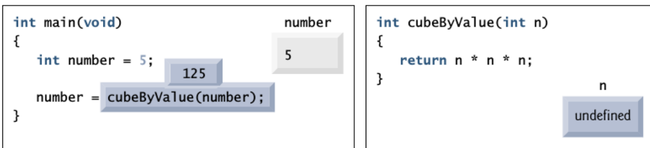
number: 5

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

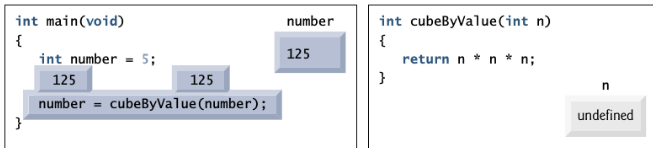
125 (value of n * n * n)

n: 5

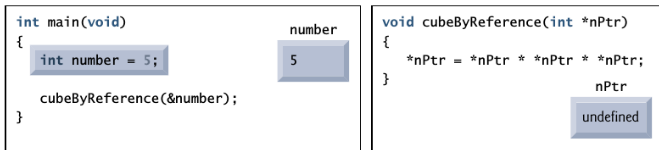
Step 4: After **cubeByValue** returns to **main** and before assigning the result to **number**:



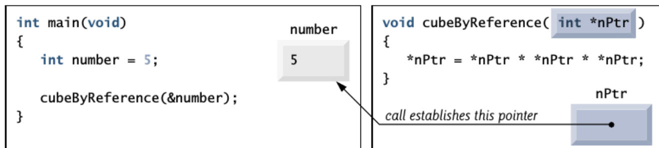
Step 5: After **main** completes the assignment to **number**:



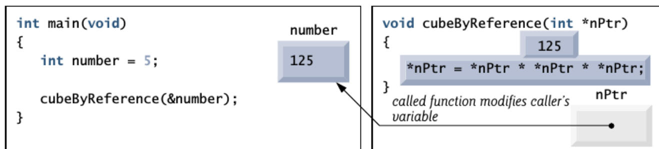
Step 1: Before main calls cubeByReference:



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



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



Passing Arguments to Functions by Reference

- All arguments in C are passed by value. C programs accomplish pass-by-reference by using pointers and the indirection operator.
- To pass by reference, apply the address operator (&).
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to read and/or modify the value at that location in the *caller's memory*.
- A function receiving an address as an argument must define a pointer parameter to receive the address.

Passing Arguments to Functions by Reference

- The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array.
- A function must “know” when it’s receiving an array vs. a single variable passed by reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.

Using the `const` Qualifier with Pointers

- The `const` qualifier indicates that the value of a particular variable should not be modified.
- If an attempt is made to modify a value that's declared `const`, the compiler catches it and issues either a warning or an error, depending on the particular compiler.
- There are four possible combinations of data and pointers. Assume

```
int x=10;
const int y=5;
```

  1. `int *myPtr = &x;` gives a non-constant pointer to non-constant data
  2. `int *const myPtr = &x;` gives a constant pointer to non-constant data
  3. `const int *myPtr = &y;` gives a non-constant pointer to constant data
  4. `const int *const myPtr = &y;` gives a non-constant pointer to constant data
- Similarly, there are four ways to pass a pointer to a function:

Case 1: Non-constant pointer to non-constant data

- Data can be modified through the dereferenced pointer
- Pointer can be modified to point to other data items.

```
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <stdio.h>
#include <ctype.h>

void convertToUppercase(char *sPtr); // prototype

int main(void)
{
   char string[] = "cHaRaCters and $32.98"; // initialize char array

   printf("The string before conversion is: %s", string);
   convertToUppercase(string);
   printf("\nThe string after conversion is: %s\n", string);
}

// convert string to uppercase letters
void convertToUppercase(char *sPtr)
{
   while (*sPtr != '\0') { // current character is not '\0'
      *sPtr = toupper(*sPtr); // convert to uppercase
      ++sPtr; // make sPtr point to the next character
   }
}
```

Case 1: Non-constant pointer to non-constant data

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Case 2: Non-constant pointer to constant data

- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type.
- But the data to which it points cannot be modified.

```
// Printing a string one character at a time using
// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters(const char *sPtr);

int main(void)
{
   // initialize char array
   char string[] = "print characters of a string";

   puts("The string is:");
   printCharacters(string);
   puts("");
}

// sPtr cannot be used to modify the character to which it points,
// i.e., sPtr is a "read-only" pointer
void printCharacters(const char *sPtr)
{
```

Case 2: Non-constant pointer to constant data

```
   // loop through entire string
   for (; *sPtr != '\0'; ++sPtr) { // no initialization
      printf("%c", *sPtr);
   }
}
```

```
The string is:
print characters of a string
```

Non-constant pointer to constant data

```c
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <stdio.h>
void f(const int *xPtr); // prototype

int main(void)
{
   int y; // define y

   f(&y); // f attempts illegal modification
}

// xPtr cannot be used to modify the
// value of the variable to which it points
void f(const int *xPtr)
{
   *xPtr = 100; // error: cannot modify a const object
}
```

```
error: l-value specifies const object
```

Case 3: Constant pointer to non-constant data

- A constant pointer to non-constant data always points to the same memory location.
- The data at that location can be modified through the pointer. This is the default for an array name.

```c
// Attempting to modify a constant pointer to non-constant data.
#include <stdio.h>

int main(void)
{
   int x; // define x
   int y; // define y

   // ptr is a constant pointer to an integer that can be modified
   // through ptr, but ptr always points to the same memory location
   int * const ptr = &x;

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

```
line 15 error: l-value specifies const object
```

Case 4: Constant pointer to constant data

- A constant pointer to constant data always points to the same memory location.
- The data at that memory location cannot be modified.

```
// Attempting to modify a constant pointer to constant data.
#include <stdio.h>

int main(void)
{
   int x = 5; // initialize x
   int y; // define y

   // ptr is a constant pointer to a constant integer. ptr always
   // points to the same location; the integer at that location
   // cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

```
line 16 error: l-value specifies const object
line 17 error: l-value specifies const object
```

The `sizeof` Operator

- Unary operator `sizeof` determine the size in bytes of a variable or type at compilation time.
- When applied to the name of an array, `sizeof` returns the total number of bytes in the array.
- Operator `sizeof` can be applied to any variable name, type or value.
- The parentheses used with `sizeof` are required if a type name is supplied as its operand.

The sizeof Operator

```c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr); // prototype

int main(void)
{
   float array[SIZE]; // create array

   printf("The number of bytes in the array is %u"
          "\nThe number of bytes returned by getSize is %u\n",
          sizeof(array), getSize(array));
}

// return size of ptr
size_t getSize(float *ptr)
{
   return sizeof(ptr);
}
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

The sizeof Operator

```
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

int main(void)
{
   char c;
   short s;
   int i;
   long l;
   long long ll;
   float f;
   double d;
   long double ld;
   int array[20]; // create array of 20 int elements
   int *ptr = array; // create pointer to array

   printf("     sizeof c = %u\tsizeof(char)  = %u"
          "\n     sizeof s = %u\tsizeof(short) = %u"
          "\n     sizeof i = %u\tsizeof(int) = %u"
          "\n     sizeof l = %u\tsizeof(long) = %u"
          "\n    sizeof ll = %u\tsizeof(long long) = %u"
          "\n     sizeof f = %u\tsizeof(float) = %u"
          "\n     sizeof d = %u\tsizeof(double) = %u"
          "\n    sizeof ld = %u\tsizeof(long double) = %u"
          "\n sizeof array = %u"
          "\n   sizeof ptr = %u\n",
          sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
          sizeof(int), sizeof l, sizeof(long), sizeof ll,
          sizeof(long long), sizeof f, sizeof(float), sizeof d,
```

The sizeof Operator

```
        sizeof(double), sizeof ld, sizeof(long double),
        sizeof array, sizeof ptr);
}
```

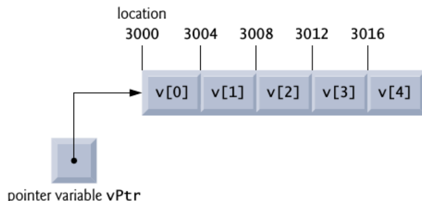
```
sizeof c = 1        sizeof(char) = 1
sizeof s = 2        sizeof(short) = 2
sizeof i = 4        sizeof(int) = 4
sizeof l = 4        sizeof(long) = 4
sizeof ll = 8       sizeof(long long) = 8
sizeof f = 4        sizeof(float) = 4
sizeof d = 8        sizeof(double) = 8
sizeof ld = 8       sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Pointer Expressions and Pointer Arithmetic

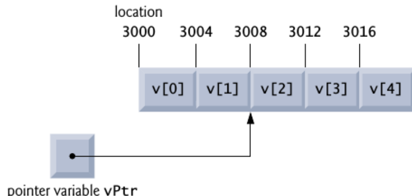
- A limited set of arithmetic operations may be performed on pointers
  - a pointer may be incremented (++) or decremented (−−),
  - an integer may be added to a pointer (+ or + =),
  - an integer may be subtracted from a pointer (− or − =), and
  - one pointer may be subtracted from another (useful only when both pointers point to elements of the same array).
- When an integer is added to or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.

Pointer Expressions and Pointer Arithmetic

- Assume `v` is an array of 5 integers and `int *vPtr=v;`



- Then `vPtr+=2;` corresponds to



- A pointer can be assigned to another pointer if both have the same type.

Pointer Expressions and Pointer Arithmetic

- An exception is the pointer of type `void *` which can represent any pointer type.
- All pointer types can be assigned a `void *` pointer, and a `void *` pointer can be assigned a pointer of any type.
- A `void *` pointer cannot be dereferenced.
- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is `NULL`.

Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An array name can be thought of as a *constant pointer*.
- Pointers can be used to do any operation involving array indexing.
- When a pointer points to the beginning of an array, adding an offset to the pointer indicates which element of the array should be referenced, and the offset value is identical to the array index. This is referred to as pointer/offset notation.
- An array name can be treated as a pointer and used in pointer arithmetic expressions that do not attempt to modify the address of the pointer.
- Pointers can be indexed exactly as arrays can. This is referred to as pointer/index notation.
- A parameter of type `const char *` typically represents a constant string.

Relationship between Pointers and Arrays

```
// Using indexing and pointer notations with arrays.
#include <stdio.h>
#define ARRAY_SIZE 4

int main(void)
{
   int b[] = {10, 20, 30, 40}; // create and initialize array b
   int *bPtr = b; // create bPtr and point it to array b

   // output array b using array index notation
   puts("Array b printed with:\nArray index notation");

   // loop through array b
   for (size_t i = 0; i < ARRAY_SIZE; ++i) {
      printf("b[%u] = %d\n", i, b[i]);
   }

   // output array b using array name and pointer/offset notation
   puts("\nPointer/offset notation where\n"
        "the pointer is the array name");

   // loop through array b
   for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
      printf("*(b + %u) = %d\n", offset, *(b + offset));
   }

   // output array b using bPtr and array index notation
   puts("\nPointer index notation");
```

Relationship between Pointers and Arrays

```
   // loop through array b
   for (size_t i = 0; i < ARRAY_SIZE; ++i) {
      printf("bPtr[%u] = %d\n", i, bPtr[i]);
   }

   // output array b using bPtr and pointer/offset notation
   puts("\nPointer/offset notation");

   // loop through array b
   for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
      printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
   }
}
```

Relationship between Pointers and Arrays

```
Array b printed with:
Array index notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer index notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Relationship between Pointers and Arrays

```
// Copying a string using array notation and pointer notation.
#include <stdio.h>
#define SIZE 10

void copy1(char * const s1, const char * const s2); // prototype
void copy2(char *s1, const char *s2); // prototype

int main(void)
{
   char string1[SIZE]; // create array string1
   char *string2 = "Hello"; // create a pointer to a string

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   char string3[SIZE]; // create array string3
   char string4[] = "Good Bye"; // create an array containing a string

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
}

// copy s2 to s1 using array notation
void copy1(char * const s1, const char * const s2)
{
   // loop through strings
   for (size_t i = 0; (s1[i] = s2[i]) != '\0'; ++i) {
      ; // do nothing in body
   }
```

Relationship between Pointers and Arrays

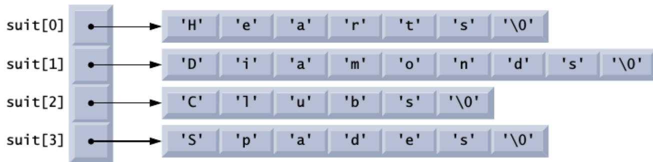
```
}

// copy s2 to s1 using pointer notation
void copy2(char *s1, const char *s2)
{
   // loop through strings
   for (; (*s1 = *s2) != '\0'; ++s1, ++s2) {
      ; // do nothing in body
   }
}
```

```
string1 = Hello
string3 = Good Bye
```

Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings.
- Each entry in the array is a string, but in C a string is essentially a pointer to its first character.
- So, each entry in an array of strings is actually a pointer to the first character of a string.

```
const char *suit[4] = "Hearts", "Diamonds", "Clubs", "Spades";
```



Pointers to Functions

- A *function pointer* contains the *address* of the function in memory.
- A function name is really the starting address in memory of the code that performs the function's task.
- Pointers to functions can be *passed* to functions, *returned* from functions, *stored* in arrays and *assigned* to other function pointers.
- A pointer to a function is dereferenced to call the function.
- A function pointer can also be used directly as the function name when calling the function.

Pointers to Functions

```
// Multipurpose sorting program using function pointers.
#include <stdio.h>
#define SIZE 10

// prototypes
void bubble(int work[], size_t size, int (*compare)(int a, int b));
int ascending(int a, int b);
int descending(int a, int b);

int main(void)
{
   // initialize unordered array a
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   printf("%s", "Enter 1 to sort in ascending order,\n"
          "Enter 2 to sort in descending order: ");
   int order; // 1 for ascending order or 2 for descending order
   scanf("%d", &order);

   puts("\nData items in original order");

   // output original array
   for (size_t counter = 0; counter < SIZE; ++counter) {
      printf("%5d", a[counter]);
   }

   // sort array in ascending order; pass function ascending as an
   // argument to specify ascending sorting order
   if (order == 1) {
```

Pointers to Functions

```c
      bubble(a, SIZE, ascending);
      puts("\nData items in ascending order");
   }
   else { // pass function descending
      bubble(a, SIZE, descending);
      puts("\nData items in descending order");
   }

   // output sorted array
   for (size_t counter = 0; counter < SIZE; ++counter) {
      printf("%5d", a[counter]);
   }

   puts("\n");
}

// multipurpose bubble sort; parameter compare is a pointer to
// the comparison function that determines sorting order
void bubble(int work[], size_t size, int (*compare)(int a, int b))
{
   void swap(int *element1Ptr, int *element2ptr); // prototype

   // loop to control passes
   for (unsigned int pass = 1; pass < size; ++pass) {

      // loop to control number of comparisons per pass
      for (size_t count = 0; count < size - 1; ++count) {

         // if adjacent elements are out of order, swap them
```

Pointers to Functions

```
         if ((*compare)(work[count], work[count + 1])) {
            swap(&work[count], &work[count + 1]);
         }
      }
   }
}

// swap values at memory locations to which element1Ptr and
// element2Ptr point
void swap(int *element1Ptr, int *element2Ptr)
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}

// determine whether elements are out of order for an ascending
// order sort
int ascending(int a, int b)
{
   return b < a; // should swap if b is less than a
}

// determine whether elements are out of order for a descending
// order sort
int descending(int a, int b)
{
   return b > a; // should swap if b is greater than a
}
```

Pointers to Functions

First run

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8    10    12    89    68    45    37
Data items in ascending order
    2    4    6    8    10    12    37    45    68    89
```

Second run

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8    10    12    89    68    45    37
Data items in ascending order
    89   68   45   37   12    10     8     6     4     2
```