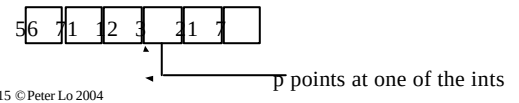
Pointers

Pointers

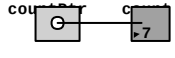
- Pointers are one of the most difficult topics to understand in C.
- Pointers "point at" areas of your computer's memory.
  - int *p; /* says p is a pointer to an int */
- Imagine your computer's memory as a series of boxes which all hold ints



Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)

  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



Pointer Variable Declarations and Initialization

- Pointer declarations
  - **`*`** used with pointer variables

```
int *myPtr;
```

  - Declares a pointer to an **`int`** (pointer of type **`int *`**)
  - Multiple pointers require using a **`*`** before each variable declaration

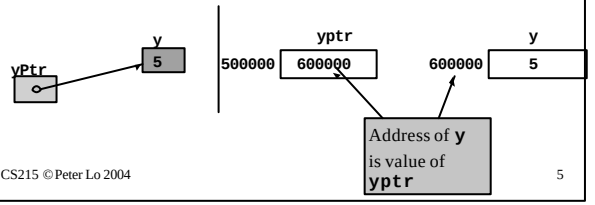
```
int *myPtr1, *myPtr2;
```

  - Can declare pointers to any data type
  - Initialize pointers to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** – points to nothing (**`NULL`** preferred)

Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
yPtr "points to" y
```



Pointer Operators

- ***** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - ***yptr** returns **y** (because **yptr** points to **y**)
  - ***** can be used for assignment
    - Returns alias to an object

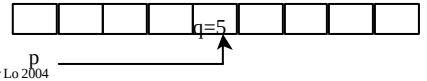
```
*yptr = 7;  // changes y to 7
```

  - Dereferenced pointer (operand of *****) must be an lvalue (no constants)

Address "&" vs. Value "*"

- & means "address of" or "point at me"
- * means "value" or "what am I pointing at?"
  - int *p;
  - int q = 5; /* q is an int */
  - p = &q; /* p now points at q */
  - printf ("p is %d\n",*p);
  - *p = 5; /* p has the same value of 5 */

We use *p to mean "the value p is pointing at"

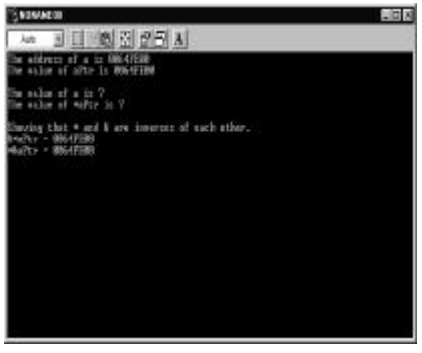


Example 6A

```
/* 6A
Using the & and * operators */
#include <stdio.h>
#include <conio.h>
int main()
{
  int a;      /* a is an integer */
  int *aPtr;  /* aPtr is a pointer to an integer */

  a = 7;
  aPtr = &a;  /* aPtr set to address of a */
  printf( "The address of a is %p"
          "\nThe value of aPtr is %p", &a, aPtr );
  printf( "\n\nThe value of a is %d"
          "\nThe value of *aPtr is %d", a, *aPtr );
  printf( "\n\nShowing that * and & are inverses of "
          "each other.\n&*aPtr = %p"
          "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
  getch();
  return 0;
}
```

Output Result



Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer
- ***** operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - ***number** used as nickname for the variable passed

Using the `const` Qualifier with Pointers

- **const** qualifier
  - Variable cannot be changed
  - Use **const** if function does not need to change a variable
  - Attempting to change a **const** variable produces an error
- **const** pointers
  - Point to a constant memory location
  - Must be initialized when declared
  - **int *const myPtr = &x;**
    - Type **int *const** – constant pointer to an **int**
  - **const int *myPtr = &x;**
    - Regular pointer to a **const int**
  - **const int *const Ptr = &x;**
    - **const** pointer to a **const int**
    - **x** can be changed, but not ***Ptr**

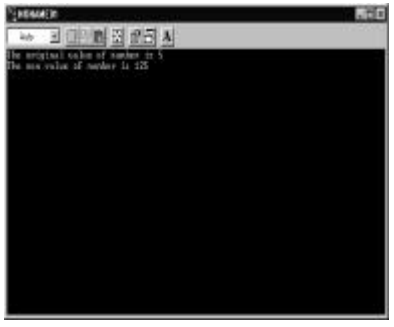
Example 6B

```
/* 6 B: Cube a variable using call-by-reference with a pointer argument */
#include<stdio.h>
#include<conio.h>
void cubeByReference( int * );  /* prototype */

int main()
{
  int number = 5;
  printf( "The original value of number is %d", number );
  cubeByReference( &number );
  printf( "\nThe new value of number is %d\n", number );

  getch ();
  return 0;
}
void cubeByReference( int *nPtr )
{
  *nPtr = *nPtr **nPtr * *nPtr;  /* cube number in main */
}
```

Output Result

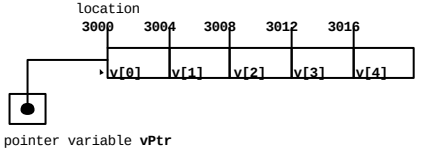


Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (**++** or **--**)
  - Add an integer to a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer Expressions and Pointer Arithmetic

- 5 element **int** array on machine with 4 byte **int**s
  - **vPtr** points to first element **v[0]**
    - at location **3000** (**vPtr = 3000**)
  - **vPtr += 2;** sets **vPtr** to **3008**
    - **vPtr** points to **v[2]** (incremented by 2), but the machine has 4 byte **int**s, so it points to address **3008**



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
```

  - **vPtr2 - vPtr** would produce 2
- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

    `bPtr = b;`
    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

      `bPtr = &b[ 0 ]`
    - Explicitly assigns **`bPtr`** to address of first element of **`b`**

The Relationship Between Pointers and Arrays

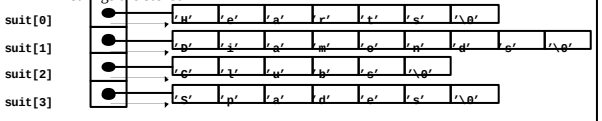
- Element **`b[ 3 ]`**
  - Can be accessed by **`*( bPtr + 3 )`**
    - Where **`n`** is the offset. Called pointer/offset notation
  - Can be accessed by **`bptr[ 3 ]`**
    - Called pointer/subscript notation
    - **`bPtr[ 3 ]`** same as **`b[ 3 ]`**
  - Can be accessed by performing pointer arithmetic on the array itself

    `*( b + 3 )`

Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

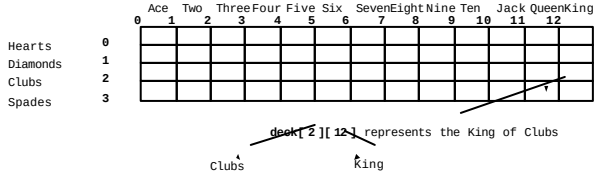
  ```
  char *suit[ 4 ] = { "Hearts", "Diamonds",
     "Clubs", "Spades" };
  ```
  - Strings are pointers to the first character
  - **`char *`** – each element of **`suit`** is a pointer to a **`char`**
  - The strings are not actually stored in the array **`suit`**, only pointers to the strings are stored



  - **`suit`** array has a fixed size, but strings can be of any size

Case Study: A Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use array of pointers to strings
  - Use double scripted array (suit, face)



  - The numbers 1-52 go into the array
    - Representing the order in which the cards are dealt

Example 6C

```
/* 6C
Card shuffling dealing program */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

void shuffle( int [][ 13 ] );
void deal( const int [][ 13 ], const char *[], const char *[] );

int main()
{
  const char *suit[ 4 ] =
    { "Hearts", "Diamonds", "Clubs", "Spades" };
  const char *face[ 13 ] =
    { "Ace", "Deuce", "Three", "Four",
      "Five", "Six", "Seven", "Eight",
      "Nine", "Ten", "Jack", "Queen", "King" };
  int deck[ 4 ][ 13 ] = { 0 };
```


Example 6C (cont')

```
  srand( time( 0 ) );

  shuffle( deck );
  deal( deck, face, suit );
  getch();
  return 0;
}

void shuffle( int wDeck[][ 13 ] )
{
  int row, column, card;

  for ( card = 1; card <= 52; card++ ) {
    do {
      row = rand() % 4;
      column = rand() % 13;
    } while( wDeck[ row ][ column ] != 0 );

    wDeck[ row ][ column ] = card;
  }
}
```


Example 6C

```
void deal( const int wDeck [][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
  int card, row, column;

  for ( card = 1; card <= 52; card++ )

    for ( row = 0; row <= 3; row++ )

      for ( column = 0; column <= 12; column++ )

        if ( wDeck [ row ][ column ] == card )
          printf( "%5s of %-8s%c",                    wFace[ column ],
wSuit[ row ],
        card % 2 == 0 ? '\n' : '\t' );
}
```


Output Result