COP 3502: Computer Science I
Spring 2004

– Day 3 –
A Review of Pointers in C

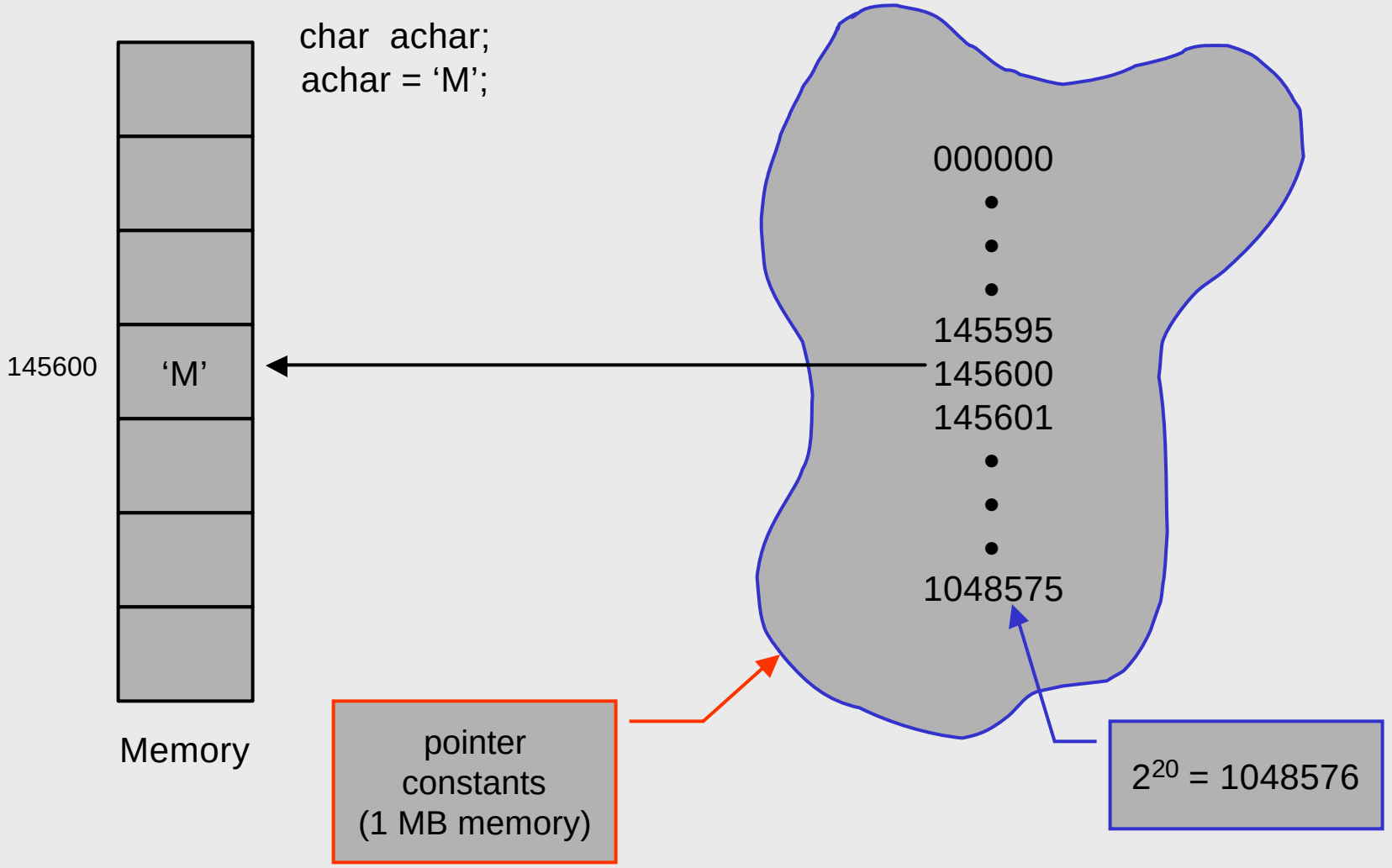
Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2004

School of Electrical Engineering and Computer Science
University of Central Florida

A Review of Pointers in C

- A pointer is a derived data type; that is, it is a data type built from one of the standard types.
- Its value is any of the addresses available in the computer for storing and accessing data.
- Pointers are built on the basic concept of pointer constants.
- Pointer constants are drawn from the set of addresses for a computer. They exist by themselves and cannot be changed by a programmer, they can only be used by the programmer.

Pointer Constants



Pointer Constants (cont.)

- In the previous slide we've declared a variable named `achar` of type `char` and assigned this variable the value of 'M'.
- At the time our program was loaded into memory for execution, the variable was assigned the location referenced by pointer constant 145600. This binding will remain in effect throughout the execution of our program. The next time the program is executed, this variable could be referenced by pointer constant 876050, or some other location in memory.
- Therefore, even though the addresses are constant, we cannot know what they will be and it is necessary to refer to them symbolically.

Pointer Constants (cont.)

- The address operator (&) provides a pointer constant to any named location in memory.
- Whenever you need a pointer value, you need only to use this operator.
- To print an address from a printf statement use the %p conversion/formatting operator.
- The sample program on the next page declares two character variables and prints their addresses (as pointer value). Try running this program on various computers to see what the results are like. Also try running it several times on the same system to see the results.

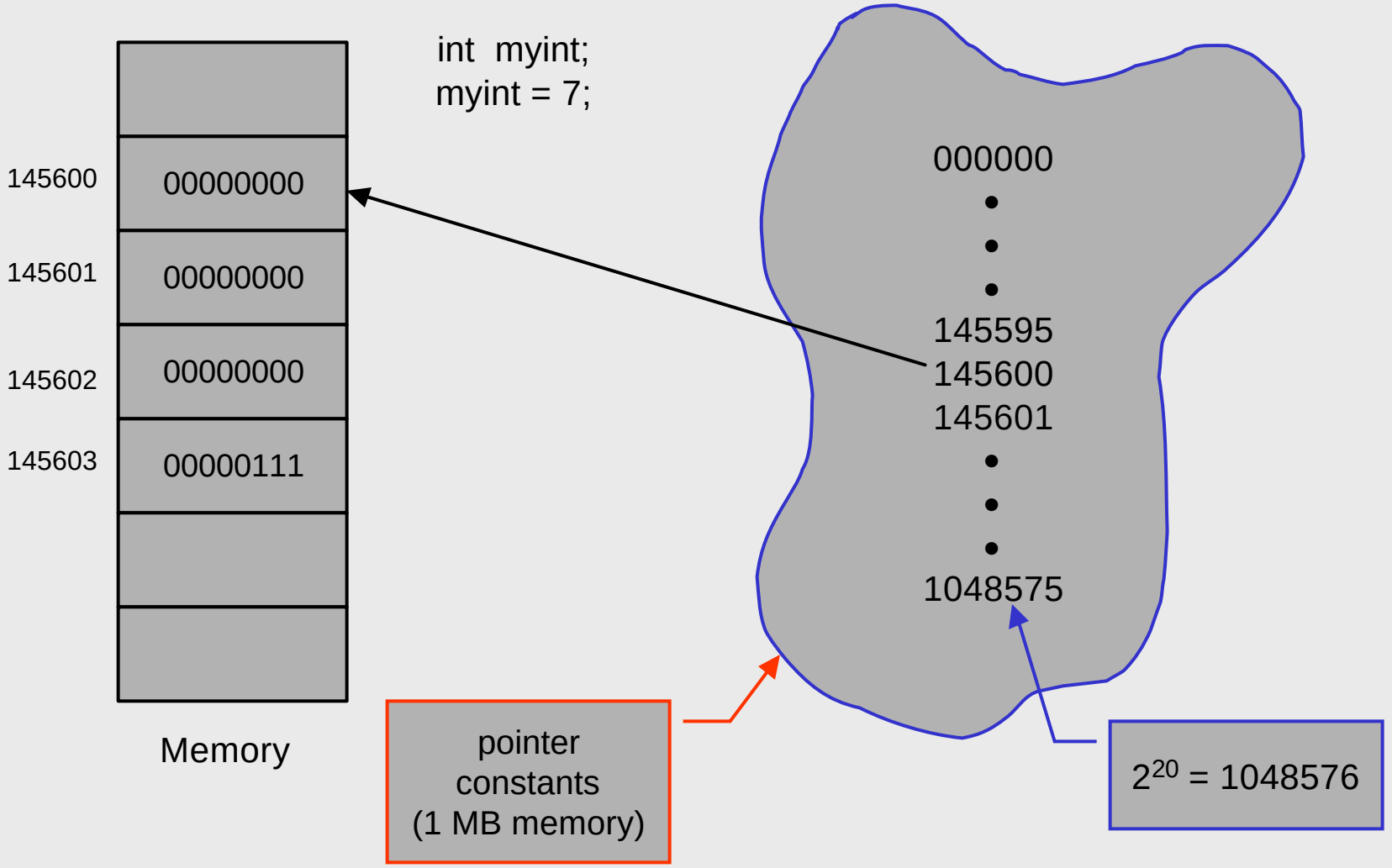
Pointer Constant Example

```
/* print character addresses */
#include <stdio.h>
int main (void)
{
      /* local definitions */
      char a;
      char b;

      /* processing statements */
      printf (“%p %p\n”, &a, &b);
      return 0;

}  /* end main */
```


Pointers to Integers

- Our discussion thus far has used characters as the data type to which our pointer is referencing. How does the picture change if the data type is changed to `integer`?
- On most computers, integers occupy either 2 or 4 bytes. Let's assume we are using a system with 4-byte integers. This means that every integer occupies four memory locations.
- In most computer systems the location of the first byte is used as the memory address of the variable.
- For characters which occupy only 1 byte, its location is its address. For integer, the address is the first byte of the four memory locations that are used to hold the integer value.

Pointer Constants to Integers



Super Brief Review of Binary Numbers

- Base or radix 2 number system
- **B**inary dig**it** is called a bit.
- Numbers are 0 and 1 only.
- Numbers are expressed as powers of 2.
- $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$, $2^{11} = 2048$, $2^{12} = 4096$, $2^{12} = 8192$, …

Conversion of Binary to Decimal

Example: convert $(110010)_2$ to decimal

$(110010)_2$

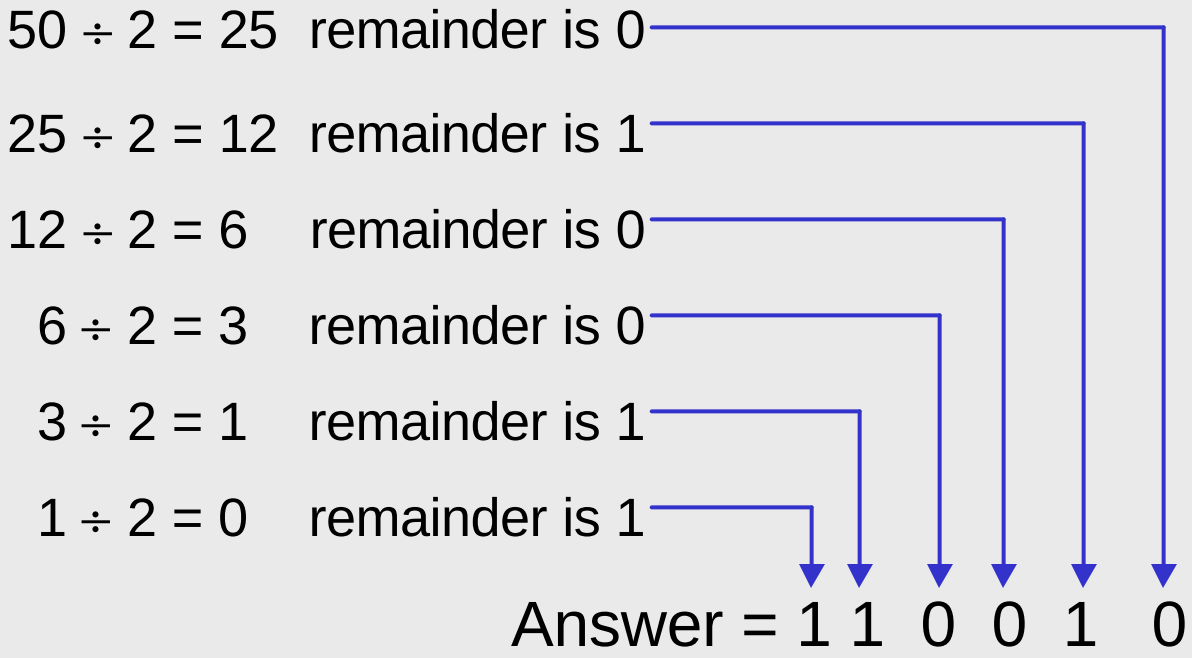
$= (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 32 + 16 + 0 + 0 + 2 + 0$

$= (50)_{10}$

Conversion of Decimal to Binary

Example: convert $(50)_{10}$ to binary



Note: the answer is read from bottom (MSB) to top (LSB) as 110010_2

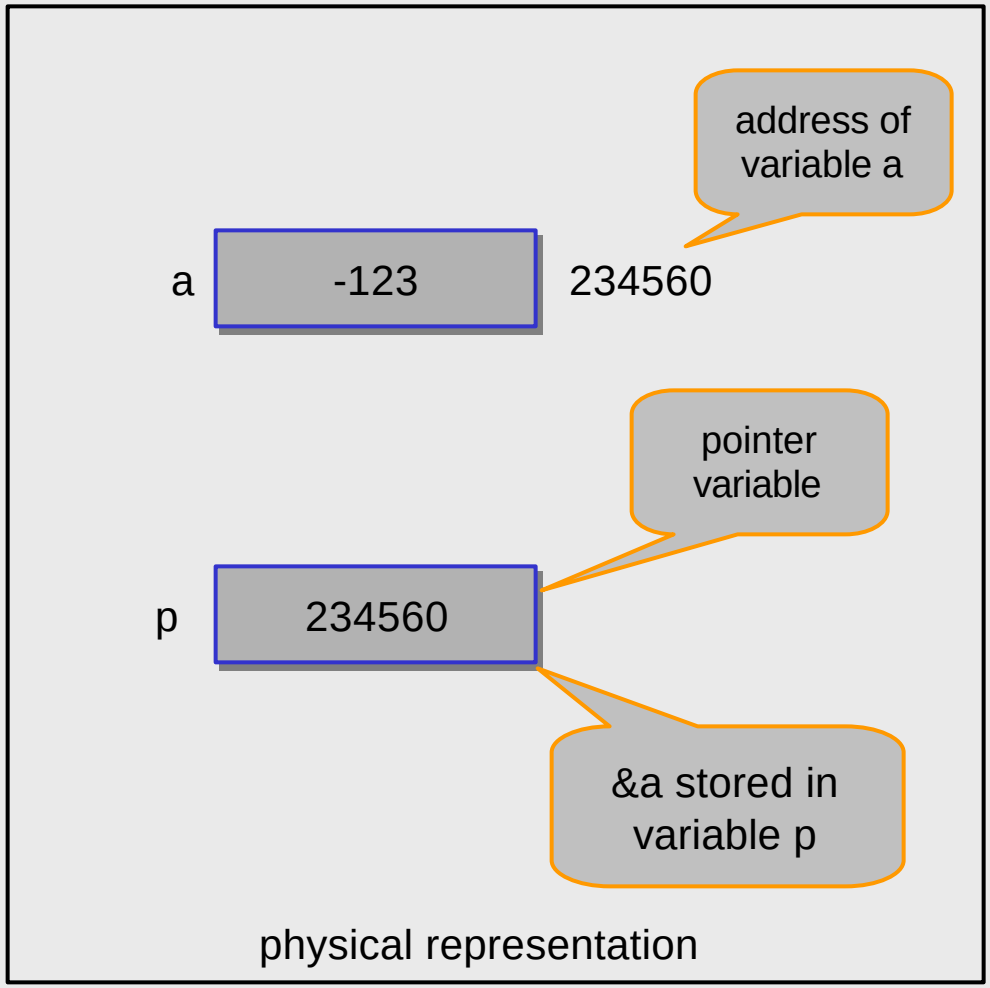
Decimal, Binary, Octal, and Hex Numbers

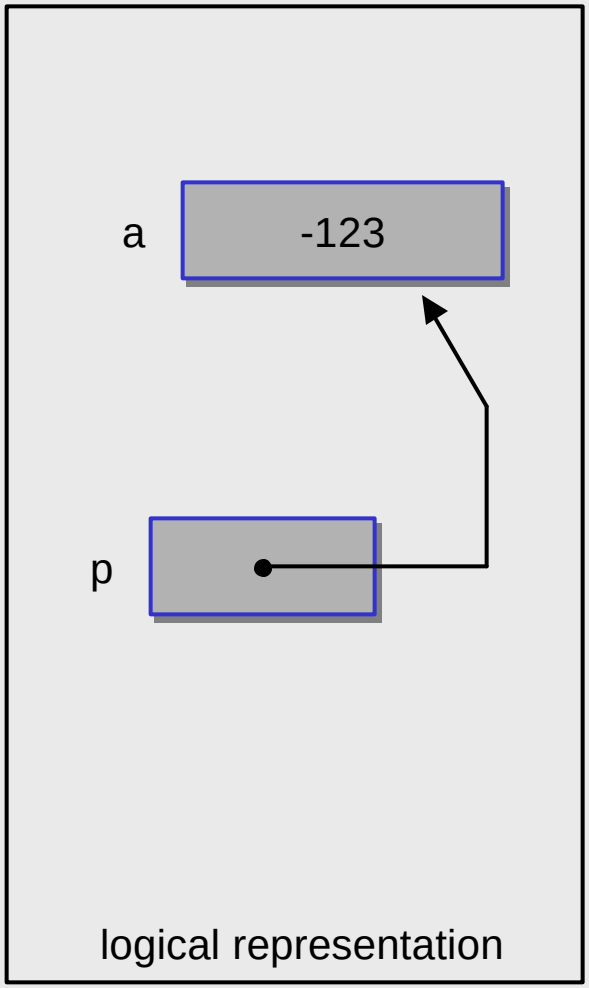
Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	0	1
2	0010	0	2
3	0011	0	3
4	0100	0	4
5	0101	0	5
6	0110	0	6
7	0111	0	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Pointer Variables

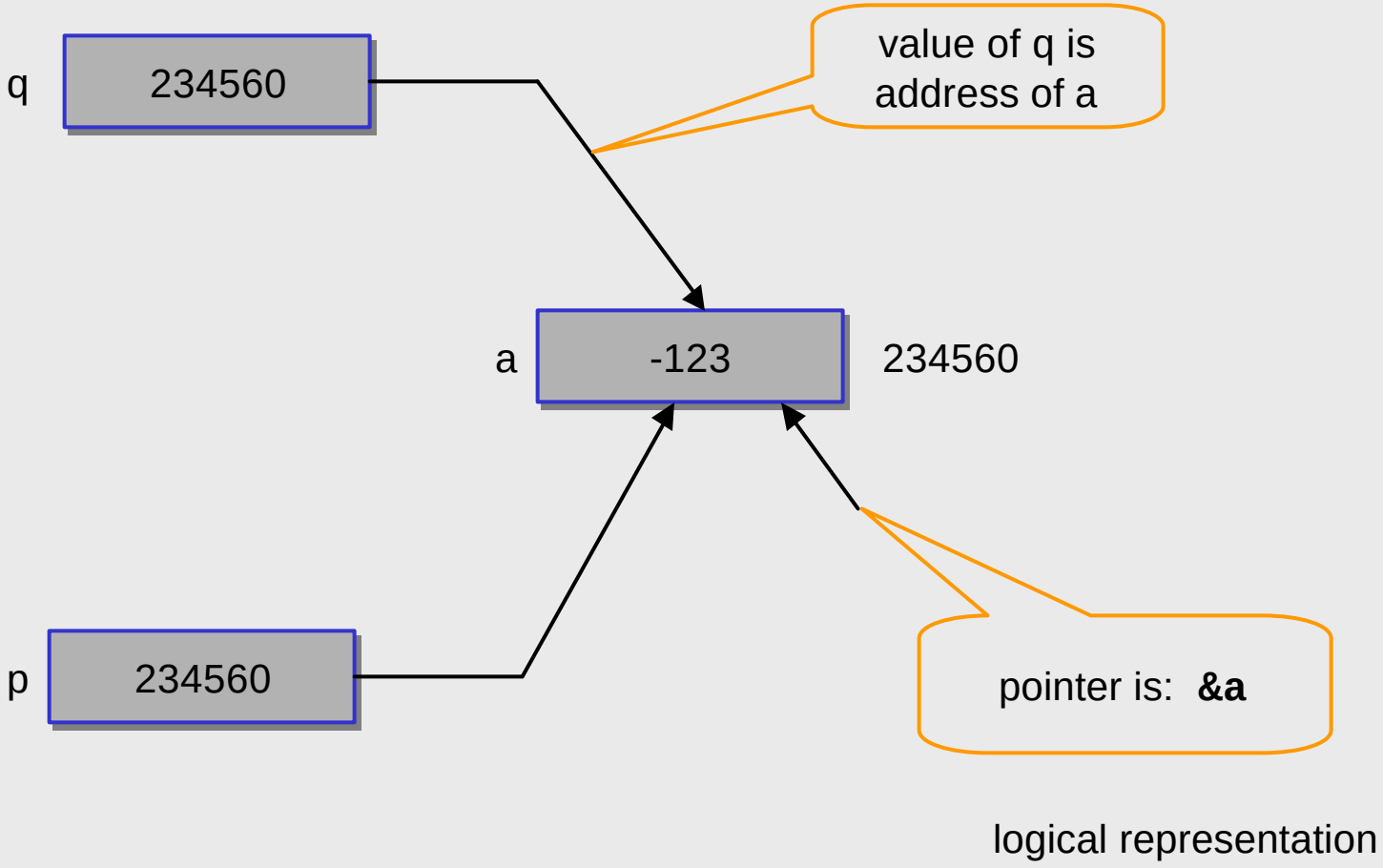
- If we have a pointer constant and a pointer value, then we can have a pointer variable.
- This means that we can store the address of a variable into another variable, which is then called a pointer variable.
- You need to be able to distinguish between a variable and its value. This distinction is shown on the next slide.
- If you have a pointer variable in C and you do not want it to reference any valid memory location, C provides the special pointer constant NULL in the <stdio.h> library.

Pointer Variables (cont.)





Multiple Pointer Variables

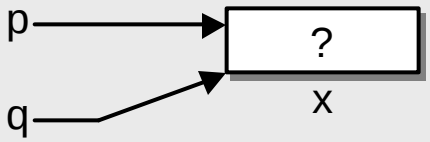
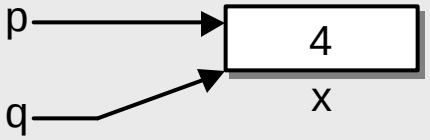
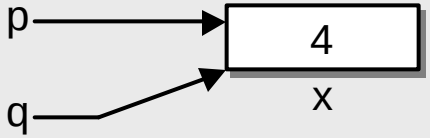
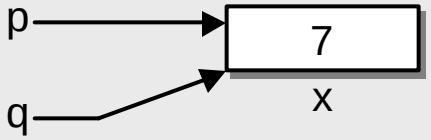
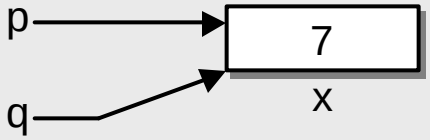
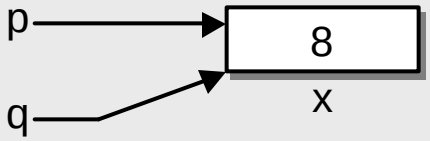
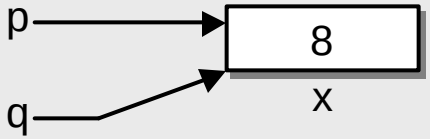
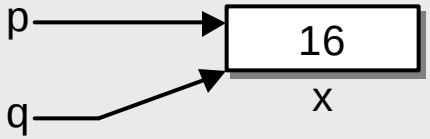
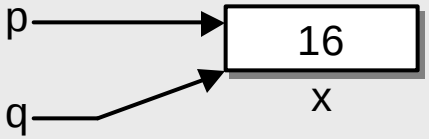
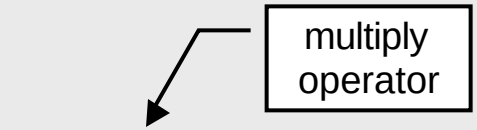
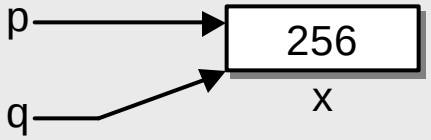


logical representation

Accessing Variables Through Pointers

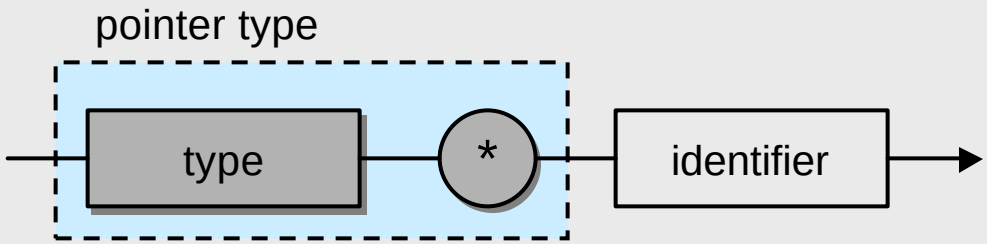
- Once you have a variable and a pointer to that variable, the question becomes: how do I relate the two, or in other words, how do I use the pointer?
- Answer: Use the indirection operator (*) in C. This is a unary operator whose operand must be a pointer value.
- When you deference a pointer, you are using its value to reference (address) another variable. The result is an expression that can be used to access the pointed variable for the purposed of inspection or alteration.
- Using the previous example, suppose we want to add 1 to the value of a. Then any of the following statements will work (assuming p = &a):

```
a++;   a = a + 1;   *p = *p +1;   (*p)++;
```

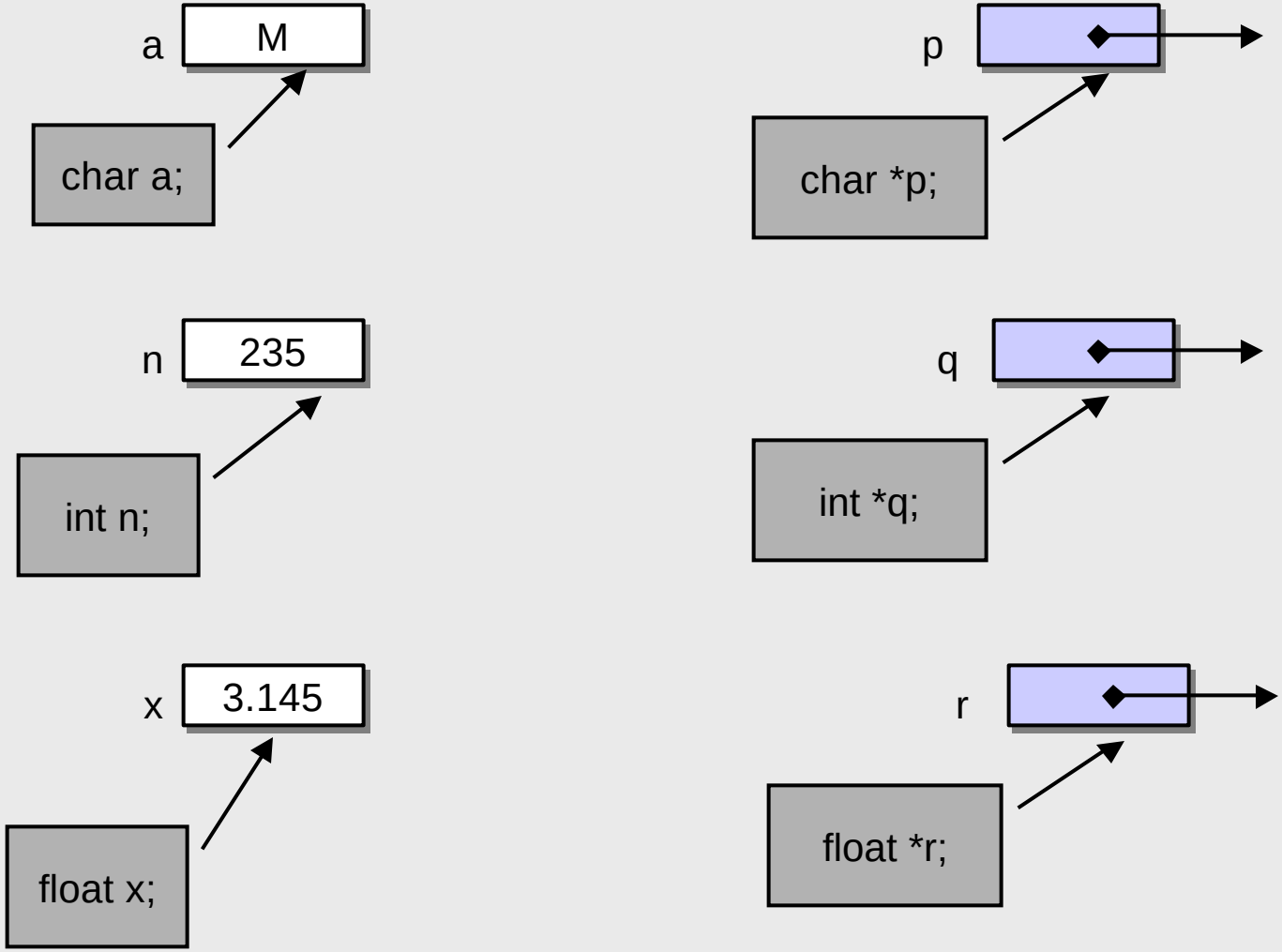

before	statement	after
	x = 4;	
	x = x + 3;	
	*p = 8;	
	*&x = *q + *p;	
	 x = *q * *p;	

Declaring Pointers

- The symbol used for the indirection operator is also used as a syntactic token to indicate the declaration of a pointer variable.
- The format in C is:



- The next slide illustrates the declaration of several different pointer variables. Their corresponding data variables are also shown for comparison.


a
M
char a;
p
char *p;
n
235
int n;
q
int *q;
x
3.145
float x;
r
float *r;

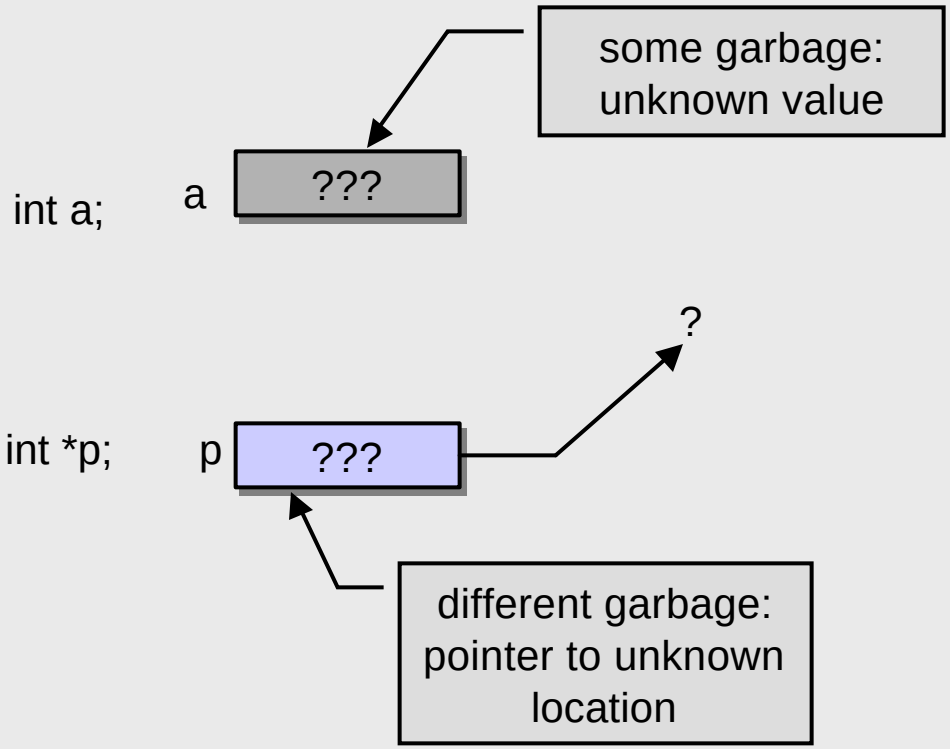
Initializing Pointers

- The C language does not initialize variables. When the program begins execution, all uninitialized variables have unknown garbage in them.
- The same is true for pointers. When the program begins execution, uninitialized pointers will contain some unknown value that will be interpreted as a memory address.
  - Most likely, this value will either not be a valid address on the computer system you are using, or if it is, it will not be valid for the memory which has been allocated to your program. If the address does not exist, you will get an immediate run-time error. If it is a valid address, you often, but unfortunately not always, get a run-time error. (Its better to get the error when you use an invalid pointer than to have the program execute in error.

Initializing Pointers (cont.)

- One of the more common errors in programming, particularly by beginning programmers, but also by professional programmers, is uninitialized pointers.
- These are very difficult errors to debug because the effect of the error is often delayed until later in the program's execution.
- Uninitialized pointers and the solution to this problem is illustrated in the next slide.

Initializing Pointers (cont.)




```
int a:          /* int variable */
int *p = &a; /* p is valid    */
*p = 89;       /* a = 89        */
```

to prevent uninitialized pointers

Pointers and Functions

- The C language uses the pass-by-value concept exclusively. This makes for a very useful application for pointers.
- The only direct way to return something back from a function is through the *return* value.
- Pass by reference can be simulated by passing an address and using it to refer back to data in the calling program. This is typically called pass by address.
  - Every time you want a called function to have access to a variable in the calling function, send the address of that variable to the called function and use the indirection operator to access it.

Pointers and Functions

```
/* prototypes */
void exchange (int x, int y):
int main (void)
{    int a = 5;
   int b = 9;
   exchange(a,b);
   printf("%d %d\n", a, b);
   return 0;
} /*end main */

void exchange(int x, int y)
{     int temp;
    temp = x;
    x = y;
    y = temp;
    return;
}  /* end exchange */
```

Which program correctly exchanges the values of a and b?

ANSWER

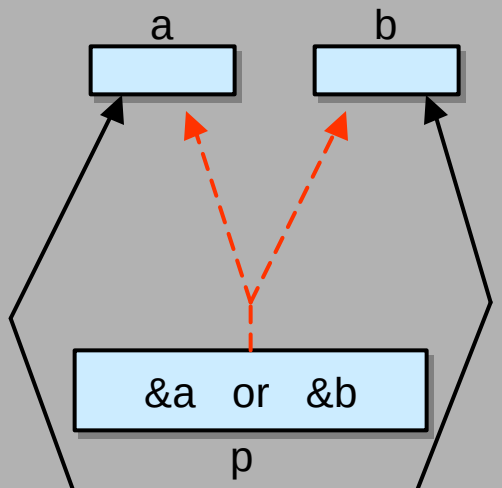
```
/* prototypes */
void exchange (int *, int *):
int main (void)
{    int a = 5;
   int b = 9;
   exchange(&a, &b);
   printf("%d %d\n", a, b);
   return 0;
} /*end main */

void exchange(int *x, int *y)
{     int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}  /* end exchange */
```


Functions Returning Pointers

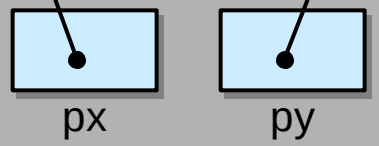
- The example on the previous page showed a function which accepted pointers as a parameter. It is also quite common for a called function to return a pointer to the calling function.
- The example on the next slide shows a function that determines the smaller of two integer values and returns a pointer to the smaller value.

Example: Functions Returning Pointers

```
/* prototypes  */
int *smaller (int  *num1, int *num2);
int main (void)
{ int a;
   int b;
   int *p;
   scanf ("%d %d", &a, &b);
   p = smaller (&a, &b);
}
```




```
int *smaller (int *px, int *py)
{
    return (*px < *py  ? px : py);
}
```



Functions Returning Pointers (cont.)

- When you return a pointer, it must point to data in the calling function or a higher level function.
- It is an error to return a pointer to a local variable in the called function, because when the function terminates, its memory may be used by other parts of the program.
- In general, it is a serious error to return a pointer to a local variable.

Pointers to Pointers

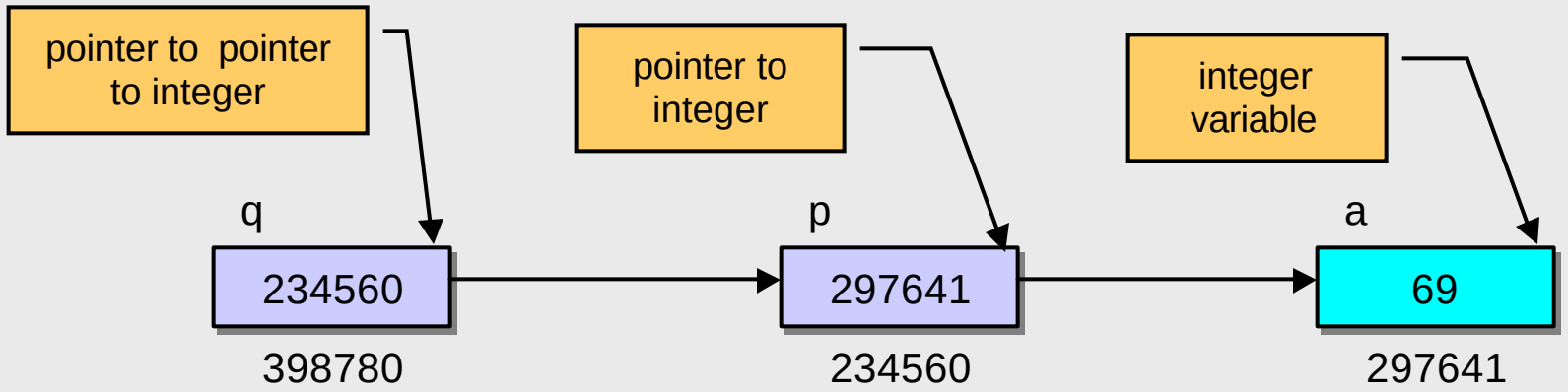
- It is possible, and with advanced data structures often necessary, to use pointers that point to other pointers.
- Using a single pointer to a data value is called indirection. When you use pointers to pointers it is called multiple-level indirection. There is no limit to the number of levels of indirection that you can use. Practically speaking, using more than two or three levels of indirection is rare.

Pointers to Pointers (cont.)

```
/* declarations */
int a;
int *p;
int **q;
/*  statements */
a = 69;
p = &a;
q = &p;
printf(" %3d %3d %3d", a, *p, **q);
```

double deference: a single deference of q would be referencing p which is a pointer to an integer.

output of printf statement is:
69 69 69



Pointer Compatibility

- It is important to remember that pointers have a type associated with them. They are not just pointer types, but rather pointers to a specific type, such as an integer.
- Each pointer therefore takes on the attributes of the type to which is refers in addition to its own attributes.
- This is demonstrated by the program on the following slide, which prints the size of a pointer and the value to which it refers.

```
#include <stdio.h>
int main(void)
{    /* local definitions */
     char   c;
     char   *pc;
     int sizeofc = sizeofc;
     int sizeofpc = sizeof(pc);
     int sizeofStarpc = sizeof(*pc);
     int a;
     int *pa;
     int sizeof a = sizeof(a);
     int sizeofpa = sizeof(pa);
     int sizeofStarpa = sizeof(*pa);
     double x;
     double *px;
     intsizeofx = sizeof(x);
     intsizeofpx = sizeof(px);
     intsizeofStarpx = sizeof(*px);
```


```
    /* statements */
    printf("sizeof(c): %3d | " sizeofc);
    printf("sizeof(pc): %3d | ", sizeofpc);
    printf("sizeof(*pc): %3d\n", sizeofStarpc);
    printf("sizeof(a): %3d | " sizeofa);
    printf("sizeof(pa): %3d | ", sizeofpa);
    printf("sizeof(*pa): %3d\n", sizeofStarpa);
    printf("sizeof(x): %3d | " sizeofx);
    printf("sizeof(px): %3d | ", sizeofpx);
    printf("sizeof(*px): %3d\n", sizeofStarpx);
    return 0;
} /* end main */
```

OUTPUT:

```
sizeof(c):    1  |  sizeof(pc):     4  |  sizeof(*pc):    1
sizeof(a):    2  |  sizeof(pa):     4  |  sizeof(*pa):    2
sizeof(x):   12  |  sizeof(px):     4  |  sizeof(*px):   12
```


Explanation of Previous Example

- What is the code in the previous example telling us about pointers?
  - Note that variables `a`, `c`, and `x` are never assigned values. This means that the sizes are independent of whatever value may be in the variable. In other words, the sizes are dependent on type not value!
  - Notice that the size of the pointers is 4 in all cases, which is the size of an address on the computer on which the program was executed.
  - Notice that the size of the type that the pointer is referring to is the same as the data size. This means that the in addition to the size of the pointer, the system also knows the size of whatever the pointer is pointing to.

Explanation of Previous Example (cont.)

- With one exception, it is invalid to assign a pointer of one type to a pointer of another type, even though the values in both cases are memory addresses and would therefore seem to be fully compatible.
  - Although the addresses may be compatible because they are drawn from the same set, what is not compatible is the underlying data type of the referenced object.
- The exception to the rule is the void pointer (also called the universal pointer or generic pointer). It can be used with any pointer, and any pointer can be assigned to a void pointer. However, since a void pointer has no object type, it cannot be de-referenced. [void *pVoid;]

Casting Pointers

- You can make an explicit assignment between incompatible pointer types by using a cast, just as you can cast an `int` to a `float`.

Casting pointers is a very dangerous operation and should be done only with a very carefully thought-out design. Otherwise you have a high potential for creating mounds of garbage!

- Given the following code, unless all operations that use p involve the same cast, who knows what the results may be!

```
int a;
int *p;
  p = (char *)&a;
```


A Final Example

- As a final example of using pointers and functions together, try writing the code for the following problem yourself, before you look at the solution on the following slide.

Problem:

Write a simple function that will convert a time given in seconds into hours, minutes, and seconds.

```
/* given time in seconds convert it to hours, minutes */
/* and seconds.  return 1 on success and 0 on failure */
int secondsToHours( long time,
                    int *hours,
                    int *minutes,
                    int *seconds)
{    /* local definitions */
     long  localTime;

     /* statements  */
     localTime = time;
     *seconds = localTime % 60;
     localTime = localTime / 60;
     *minutes = localTime % 60;
     *hours = localTime / 60;
     if (*hours > 24)
        return 0;  /* error in time */
     else
        return 1;  /* successful time conversion */
} /* end secondsToHours */
```


A Final Note on Functions

Good Programming Style Tip #1

Create local variables when a value parameter will be changed within a function so that the original value will always be available for processing.

Good Programming Style Tip #2

When several values need to be returned to the calling function, use address parameters for all of them. Do not return one value and use address parameters for the others. Use the return for some other reason, such as a status flag, or make the return void.