

Part III: Pointers

Unit 9: Indirection and Pointers

B-1

*The choice of a point of view is the
initial act of culture.*

-Ortega y Gasset

B-2

What is a Pointer?

- A *pointer* is a variable used to hold the address of some other specific kind of object (a piece of data, a function, etc.)
- When defining a pointer, the programmer tells exactly what kind of object the pointer is designed to "point to"
- To be used correctly, a pointer must be assigned a valid address of the proper type of object
- While data objects come in various shapes and sizes, pointers are always the same size (except for memory model issues)

B-3

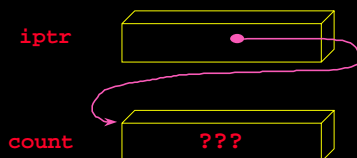
Why Pointers and Indirection?

- Support for *pointers* is one of the most sophisticated features of C, and probably the single most complex to master
- Pointers allow the programmer to:
 - Effectively represent complex data structures
 - Change values of objects passed as parameters to functions
 - Work conveniently with dynamic memory allocation
 - Optimize certain operations with arrays

B-4

Pointers

- If `count` is an integer variable, and `iptr` is a pointer to integers, then the two variables are related as shown in this diagram:



B-5

Pointer Definition

- A pointer is defined by placing an asterisk character before the identifier name for each "level of indirection" desired

```
/* Some simple pointer definitions */  
  
int *iptr;      /* iptr is a pointer to int */  
char *str;     /* str is a pointer to char */  
  
int i, j, *k;  /* i and j are simple ints, */  
              /* but k is a pointer to int */
```

B-6

The "Address of" Operator: &

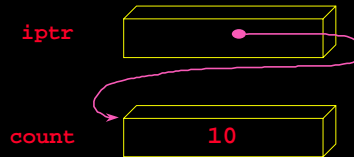
- & ("address of") is a unary operator
- Used to obtain the address of an object, so that address may be assigned to a pointer variable
- Results in an expression of type "pointer to operand-type"
- Overcomes data privacy: It is the only way to pass an object (other than an array) by reference

B-7

Simple Pointer Definition

```
int count = 10; // simple integer
int *iptr;     // pointer to int

iptr = &count; // iptr points to count
```

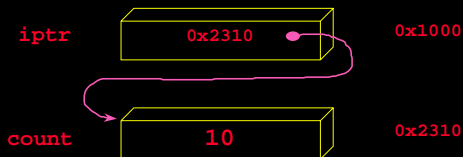


B-8

Simple Pointer Definition

```
int count = 10; /* simple integer */
int *iptr;     /* pointer to int */

iptr = &count; /* iptr points to count */
```



B-9

The Indirection Operator: *

- Obtains the value of the object "being pointed to" by its operand
- For any data type "pointer to x", the result of applying the indirection operator to that data type is an expression of type "x"
- The * operator may only be applied to pointer arguments
- * can be thought of as the "inverse" of the & operator
- Another name for * is the "Contents of" operator

B-10

Pointer Indirection

- The use of pointer indirection enables you to access the value of a variable pointed to by a pointer "indirectly" through the pointer -- similar to assembly language "indirect" memory reference
- The value of a pointer should always be treated as an address of a data object of the type the pointer was defined for
- A pointer's value may change to point to a different object *of the same type* as the pointers is defined for

B-11

Simple Indirection (ptr0.c)

```
int main()
{
    int count, x; /* simple integers */
    int *iptr;   /* pointer to int */

    iptr = &count; /* iptr points to count */
    count = 10;   /* give count a value */
    x = *iptr;   /* indirection */

    printf("count = %d, x = %d\n", count, x);
    return 0;
}

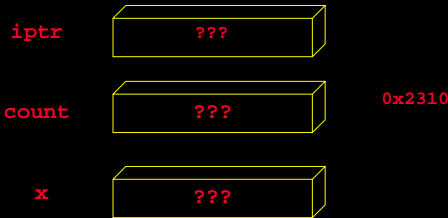
/* Output:
count = 10, x = 10 */
```

B-12

Simple Indirection

```
int count, x; /* simple integers */
int *iptr; /* pointer to int */

// iptr = &count; /* iptr points to count */
// count = 10; /* give count a value */
// x = *iptr; /* x gets value of count */
```

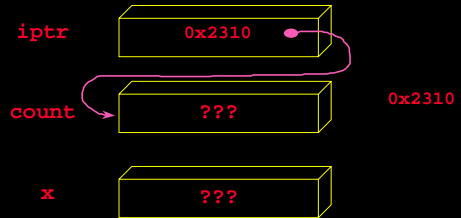


B-13

Simple Indirection

```
int count, x; /* simple integers */
int *iptr; /* pointer to int */

iptr = &count; /* iptr points to count */
// count = 10; /* give count a value */
// x = *iptr; /* x gets value of count */
```

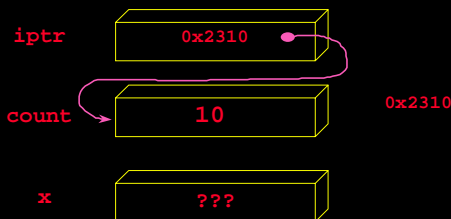


B-14

Simple Indirection

```
int count, x; /* simple integers */
int *iptr; /* pointer to int */

iptr = &count; /* iptr points to count */
count = 10; /* give count a value */
// x = *iptr; /* x gets value of count */
```

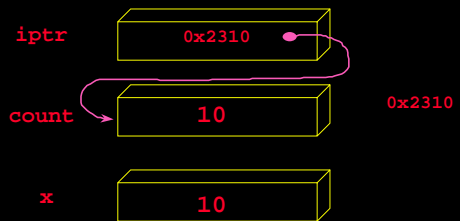


B-15

Simple indirection

```
int count, x; /* simple integers */
int *iptr; /* pointer to int */

iptr = &count; /* iptr points to count */
count = 10; /* give count a value */
x = *iptr; /* x gets value of count */
```



B-16

Pointer Example (ptr1.c)

```
/* ptr1.c: Pointer indirection and re-assignment */

int main()
{
    int i, j, *iptr;

    i = j = 100;
    iptr = &i;
    printf("%d, %d\n", i, *iptr); // 100, 100
    i = 50;
    printf("%d, %d\n", i, *iptr); // 50, 50
    iptr = &j;
    printf("%d, %d\n", i, *iptr); // 50, 100
    i = *iptr;
    printf("%d, %d\n", i, j); // 100, 100
    return 0;
}
```

B-17

Indirect Assignment

- An indirection operator may appear on the left side of assignment expressions
- The construct:

`*ptr = value;`

says to assign the value of `value` to the memory location whose address is contained as the *value* of the pointer `ptr`

- The data types of `value` and whatever `ptr` was defined as pointing to should agree

B-18

Indirect Assignment (ptr3.c)

```
int main()
{
    int count = 20, x; /* simple integers */
    int *iptr;        /* pointer to int */

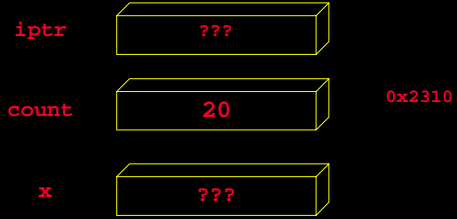
    iptr = &count;   /* iptr points to count */
    x = 10;          /* give x a value */
    *iptr = x;       /* count gets value of x */
                    /* indirectly "through" */
                    /* iptr */
    printf("count = %d, x = %d\n", count, x);
    return 0;
}
/* Output:
   x = 10, count = 10 */
```

B-19

Indirect Assignment

```
int count = 20, x; /* simple integers */
int *iptr;        /* pointer to int */

// iptr = &count; /* iptr points to count */
// x = 10;        /* give count a value */
// *iptr = x;     /* count gets value of x */
```

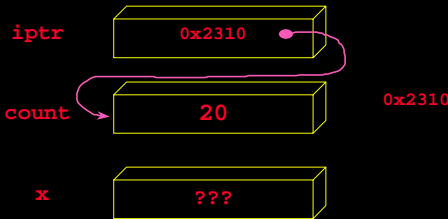


B-20

Indirect Assignment

```
int count = 20, x; /* simple integers */
int *iptr;        /* pointer to int */

iptr = &count;   /* iptr points to count */
// x = 10;      /* give count a value */
// *iptr = x;   /* count gets value of x */
```

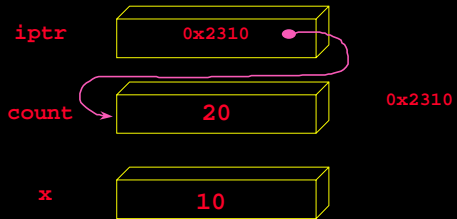


B-21

Indirect Assignment

```
int count = 20, x; /* simple integers */
int *iptr;        /* pointer to int */

iptr = &count;   /* iptr points to count */
x = 10;          /* give count a value */
// *iptr = x;   /* count gets value of x */
```

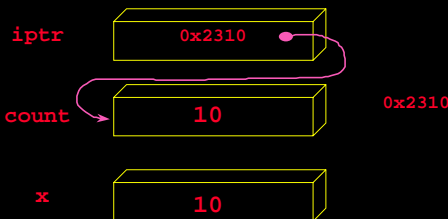


B-22

Indirect Assignment

```
int count = 20, x; /* simple integers */
int *iptr;        /* pointer to int */

iptr = &count;   /* iptr points to count */
x = 10;          /* give count a value */
*iptr = x;       /* count gets value of x */
```



B-23

Indirect Assignment

```
/*
 * More assigning "through" pointers
 */
```

```
int i, *ip;

ip = &i; /* ip points to i */
i = 100; /* set i to 100 */

*ip = 50; /* set i to 50 */

/* is this legal? */
*i = 50; /* why or why not? */
```

B-24

Pointer Example with Indirect Assignment (ptr2.c)

```
/* ptr2.c: Indirect assignment through pointers */
int main()
{
    int i, j, *iptr;

    i = j = 100;
    iptr = &i;
    printf("%d, %d\n", i, j);        // 100, 100
    *iptr = 50;
    printf("%d, %d\n", i, j);        // 50, 100
    iptr = &j;
    printf("%d, %d\n", i, *iptr);    // 50, 100
    *iptr = 275;
    printf("%d, %d\n", i, j);        // 50, 275
    return 0;
}
```

B-25

Passing Function Arguments by Reference

- Recall that each value supplied in the call to a function is *copied* into the corresponding formal parameter within the function
- If the parameter being passed represents the *address* of an object (via a pointer to that object) rather than a copy of the object's value, the function has a "handle" on that object and could possibly change the object's value
- This arrangement, where a function parameter's value is actually a *pointer* to an object, is called *call by reference*

B-26

Call-by-Reference Parameters

```
// Example of passing a pointer in a function call
void set_sum(double *result, double val1, double val2);

int main()
{
    double sum;
    double *sum_pointer = &sum;
    set_sum(sum_pointer, 2.2, 3.5);
    printf("sum = %f\n", sum); // prints: sum = 5.7
    return 0;
}

void set_sum(double *result, double val1, double val2)
{
    *result = val1 + val2;
}
```

B-27

Call-by-Reference Parameters

```
// Example of passing a pointer in a function call
void set_sum(double *result, double val1, double val2);

int main()
{
    double sum;

    set_sum(&sum, 2.2, 3.5); // pass ptr to sum
    printf("sum = %f\n", sum); // prints: sum = 5.7
    return 0;
}

void set_sum(double *result, double val1, double val2)
{
    *result = val1 + val2;
}
```

B-28

Call-by-Reference (bump.c)

```
// Another example of passing by reference
void bump (int *); // prototype for bump()

int main()
{
    int x = 50;

    printf("x = %d\n", x);
    bump(&x); // generate and pass pointer to x
    printf("x now = %d\n", x); // shows 51
    return 0;
}

void bump (int *intp)
{
    (*intp)++; // What effect would taking out
              // the parentheses have?
}
```

B-29

Pointer Assignment Pitfall

```
/* Example: improper use of pointers */

int i, *iptr; /* iptr points to i */

i = 50;

*iptr = 100; /* Does this change i? why? */
```

B-30

**IMPORTANT!!!
DANGER, WILL ROBINSON!**

- Pointer variables *must be initialized* before they can be correctly used as pointers!
- Indirect assignment through uninitialized pointers creates some of the toughest C bugs to find and correct
- If you don't have a useful initial value for a pointer, initialize it to NULL; this is legal:

```
int *intp = NULL; // same as 0
```
- This allows the run-time system to diagnose indirect assignment through an uninitialized pointer ("NULL Pointer Assignment")

B-31

**Project:
pointers1**

B-32

**Pointers to char vs.
char Arrays**

```
char a[] = "efgh"; // array of 5 chars  
char *cp = "abcd"; // a char pointer
```

```
main()  
{  
    printf("cp's value is: %s\n", cp);  
    printf("a's value is: %s\n", a);  
}
```

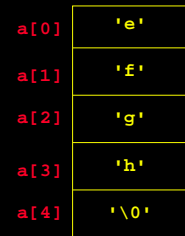
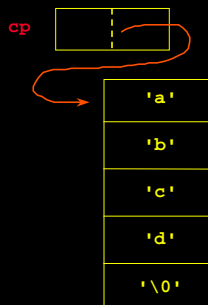
```
// output:  
//     cp's value is: abcd  
//     a's value is: efgh
```

B-33

Pointers to char vs. char Arrays

```
char *cp = "abcd";
```

```
char a[] = "efgh";
```



B-34

Re-Assigning to a char Pointer

```
char a[] = "abcd"; // array of 5 chars  
char *cp = "abcd"; // a char pointer  
...
```

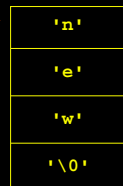
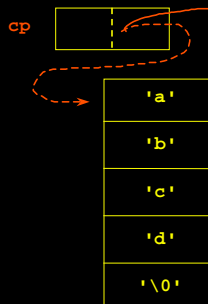
```
cp = "new"; // legal, but "loses"  
// original "abcd"
```

B-35

Re-Assigning to a char Pointer

```
char *cp = "abcd";
```

```
cp = "new";
```



B-36

Can't Assign to an Array

```
char a[] = "abcd"; // array of 5 chars
char *cp = "abcd"; // a char pointer

...

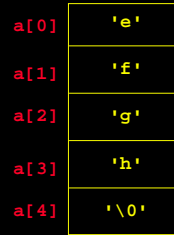
a = "hello"; // ILLEGAL! array name
              // is not a "modifiable
              // lvalue"
```

B-37

Can't Assign to an Array

```
char a[] = "efgh"; // This syntax may only
                  // be used for definitions

a = "hello"; // More than C can handle.
```



B-38

Copying over an Existing Array

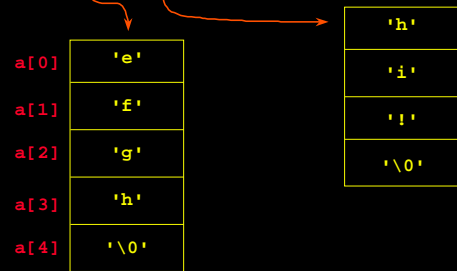
```
char *cp = "abcd"; // a char pointer
char a[] = "efgh"; // array of 5 chars
...

strcpy(a, "hi!"); // legal, but wasteful
```

B-39

Copying over an Existing Array (before)

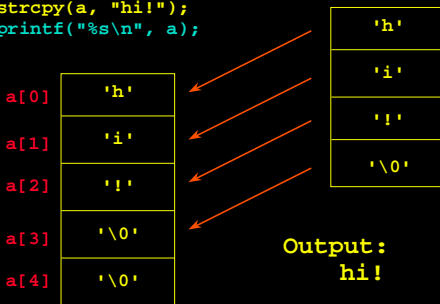
```
char a[] = "efgh";
strcpy(a, "hi!");
```



B-40

Copying over an Existing Array (after)

```
char a[] = "efgh";
strcpy(a, "hi!");
printf("%s\n", a);
```



B-41

Verifying Pointer Usage

- There is a handy trick you can use to check the type of a pointer expression
- Note that a pointer's *definition* often mirrors the way the pointer is later used within expressions
- To check a pointer expression's type, look at the pointer's definition and cover up the parts resembling the expression
- What *remains* of the definition is the pointer expression's type

B-42

Verifying Pointer Usage

```
// Examples of how to check for legal
// pointer usage

char *p1;           // a ptr to char
int *p2[10];       // array of 10 ptrs to int
int i, **ipp;      // ipp is a ptr to ptr to int
                  // Go back to the definitions:

p1 = 'c';          // cover "p1", you get "char **",
                  // so this is illegal

p2[3] = &i;        // cover p2[10], get "int **",
                  // &i is also "int **", so OK.

ipp = &p2[2];      // cover p2[10], get "int **",
                  // address of that is "int **",
                  // just like ipp. This is OK.
```

B-43

Project: pointers2

B-44

Pointer Arithmetic

- Addition and subtraction operations involving pointers have some “special” properties
- Pointer Addition
 - Addition may be performed on one pointer and one integral operand
 - The integral operand is “scaled” by the size (in bytes) of the object type pointed to by the pointer operand
 - For example: Adding an `int` pointer and an integer causes the integer to be multiplied by 4 (for 32-bit integers) before the addition
 - The result of the operation has type “pointer to ...”

B-45

Pointer Arithmetic

- Pointer Subtraction
 - If subtracting a pointer and an integer, the **integer value is scaled** just as in pointer addition
 - Subtracting two pointers is allowed, but only if the two pointers point to objects of the **same data type**
 - When subtracting two pointers, the difference of the two unmodified pointer values is computed, but the result **is then scaled** (divided) by the size of the object being pointed to

B-46

Array Names are “Special”

- When an expression of type “array of *foo*” appears *without* a trailing square bracket, then that expression almost always *acts as if* it has type “pointer to *foo*”
- The value of such an expression is a pointer to the first element of the array
- This property is usually illustrated by noting that, for any array `a`:
 `a`
evaluates as
 `&a[0]`

B-47

Array Names

```
// Examples of the properties of array names

char string[] = "hello, there";
char *sp;

sp = string;       // Both sides pointers to char
sp = &string[0];  // Same thing exactly

sp = "another string"; // sp points elsewhere
*sp = *string;         // All three of these
*sp = *(&string[0]);   // are exactly
*sp = string[0];      // equivalent
```

B-48

Array Names are “Special”

- Arrays are the exception to the rule that function parameters are always passed by value. Arrays are passed *by reference*
- This is natural, once you get accustomed to the fact that an array name without a subscript acts just like a pointer
- The only way to pass an array by *value* (that is, force a copy to be passed to a function) is to embed the array within a structure (Unit 13) and pass the structure

B-49

Pointers and Arrays

- Many kinds of programs can be implemented using either pointers or arrays
- How do you know which is most appropriate for any given application?
 - Iteration through an array usually calls for subscripting, so use arrays
 - String processing is often clearer with pointers

B-50

Pointers and Arrays

```
#define SIZE 100

int i;
float array[SIZE], *fp;

/* initialize array elements (array version) */

for (i = 0; i < SIZE; i++)
    array[i] = 3.14159;

/* do it again, with pointers: */

for (fp = array; fp != &array[SIZE]; fp++)
    *fp = 3.14159;

/* plain array version more appropriate here */
```

B-51

Pointers and Arrays

```
/* String processing with array subscripting:
   Scan a string for terminating NUL,
   and set zptr to point to it: */

char str[132];          /* String buffer      */
char *zptr;            /* To point to the NUL */
int i;                /* loop variable      */

for (i = 0; i < 132; i++)
{
    if (str[i] == '\0') /* found NUL? */
    {
        zptr = &str[i]; /* if so, set */
        break;          /* zptr      */
    }
}
```

B-52

Pointers and Arrays

```
//
// Same example, using pointers:
//

char str[132]; // A string buffer
char *zptr;    // to be the result ptr

zptr = str;    // same as: zptr = &str[0]

while (*zptr != '\0') // same as: while (*zptr)
    zptr++;           // increment while non-null
```

B-53

Pointers and Arrays

```
//
// Same example, using pointers and for:
//

char str[132]; // A string buffer
char *zptr;    // to be the result ptr

for (zptr = str; *zptr; zptr++)
    ;
```

B-54

Arrays as Function Parameters

- Since passing an array to a function is *really* passing a pointer, C doesn't care whether you write a formal parameter in a function definition (or its prototype) as a pointer or as an array; both forms are 100% equivalent
- This works both ways: When calling a function, you may pass either a pointer or an array, regardless of which type appears in the prototype
- This can look rather strange...

B-55

Copying a String, Again (strcpy2a.c)

```
/* string copy made into a separate function:*/
char str1[] = "This is the source string";
char str2[100]; // destination string
void strcpy(char *src, char *dest); // prototype

int main()
{
    strcpy(str1, str2);
    printf("str1 is: %s\nstr2 is: %s\n",str1,str2);
    return 0;
}

// arrays or pointers?
void strcpy(char src[], char dest[])
{
    int i;
    for (i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];

    dest[i] = '\0';
}
```

B-56

String Copy Using Pointers (strcpy3.c)

```
/* strcpy3.c: strcpy() function using pointers: */
char str1[] = "This is the source string";
char str2[100]; // destination string
void strcpy(char src[], char dest[]); // prototype
// Prototype can stay the same (!)
int main()
{
    strcpy(str1, str2); // args become ptr to char
    printf("str1 is: %s\nstr2 is: %s\n",str1,str2);
    return 0;
}

void strcpy(char *src, char *dest) // not same as the
{ // prototype !??
    while (*src != '\0') // while not done */
    {
        *dest = *src; // copy a char */
        src++; // bump src ptr */
        dest++; // bump dest ptr */
    }
    *dest = '\0'; // terminating null */
}
```

B-57

An Improved strcpy() (strcpy4.c)

```
/* strcpy4.c: strcpy() function variation */
char str1[] = "This is the source string";
char str2[100]; // destination string
void strcpy(char src[], char dest[]); // prototype
// Prototype can stay the same (!)
int main()
{
    strcpy(str1, str2);
    printf("str1 is: %s\nstr2 is: %s\n",str1,str2);
    return 0;
}

void strcpy(char *src, char *dest)
{
    /* same thing, but combine test for null with */
    while ((*dest = *src) != '\0') /* assignment */
    {
        src++;
        dest++;
    }
    // No longer need to set terminating null!
}
```

B-58

Example: Appending Strings (strapp.c)

```
// Sample use of function to append a string
char str[100] = "This is the original string";

void strapp(char str1[], char str2[])
{
    // append str2 onto end of str1
    int i, j;
    for (i = 0; str1[i] != '\0'; i++)
        ; // locate end of str1[]
    for (j = 0; str2[j] != '\0'; j++, i++)
        str1[i] = str2[j]; // now append str2[]
    str1[i] = '\0'; // and terminate it.
}

int main()
{
    printf("String before: %s\n", str);
    strapp(str, " and this is more!");
    printf("String after: %s\n", str);
    return 0;
}
```

B-59

Arrays and Pointers: An Identity

Given any array `a` and integer value `i`, the expression

`a[i]`

is syntactically equivalent to the expression

`*(a + i)`

or, for that matter,

`*(i + a)`

and *even*, believe it or not:

`i[a]`

B-60

Pointers and Arrays (whoa.c)

```
int vals[] = { 10, 20, 30, 40, 50, 60 };

int main()
{
    int *valp;

    printf("vals[2] = %d\n", vals[2]);
    printf("**(vals + 2) = %d\n", *(vals + 2));
    printf("(2 + vals) = %d\n", *(2 + vals));
    printf("2[vals] = %d\n", 2[vals]);

    valp = &vals[2];
    printf("**valp = %d\n", *valp);
    printf("(valp + 1) = %d\n", *(valp + 1));
    printf("valp[3] = %d\n", valp[3]);
    printf("3[valp] = %d\n", 3[valp]);
    return 0;
}
```

B-61