C Pointers
- Powerful C feature but challenging to understand
- Some uses of pointers include
  - Call by reference parameter passage
  - Dynamic data structures
    - Data structures that can shrink or grow
    - Creating arrays during program execution
    - Other examples include linked lists, stacks and trees

Indirection
- Indirection = referencing a value through a pointer
- int count = 7; /* Regular int variable */
- int *countPtr; /* Pointer to int */
- countPtr = &count; /* Set countPtr to point to count */
- /* Can now use count and *countPtr interchangeably */
- count = count + 1;
- /* is the same as */
- *countPtr = *countPtr + 1;

Creating Pointers
- Pointers (like any other variables) must be declared before they can be used
- Examples
  - int *countPtr, count;
  - countPtr is declared as type int *
  - int * means a pointer to an integer value
  - "countPtr is a pointer to an int"
  - "countPtr holds the address in memory of an integer value"
- The "*" can be used to define a pointer to any C data type.

Pointer Declarations
- General format for declaring a variable as a pointer to a particular type:
  - name-of-type *nameOfPointer
- This declares nameOfPointer as a pointer
  - int *iPtr; /* pointer to type int */
  - float *fPtr;
  - char *cPtr;
  - double *dPtr;
- What are iPtr, fPtr, cPtr, and dPtr?
  - Each is a pointer to its associated data type

Pointer Declarations
- Read pointer declarations from right to left and substitute the word “address” for the * operator
  - int *iPtr;
  - *iPtr is an address of an integer
  - int *iPtr;
  - iPtr is address of an integer
  - *iPtr is an integer
**Pointer Declarations**

- Read pointer declarations from right to left and substitute the word “address” for the * operator
  
  ```
  int * * iPtr;
  ```

  - `iPtr` is an address of an address of an integer
  - `*iPtr` is an address of an integer
  - `**iPtr` is an integer
  - `***iPtr` is a syntax error

**Initializing Pointers**

- Pointers should be initialized when declared, or in an assignment statement
  - Initialization needed before they are used (just like other variables)
  - Pointer initialization values
    - NULL: Equivalent to zero, defined in stdio.h, used to indicate a pointer to “nothing”
    - A memory address (usually of another variable)
    - or from a memory allocation function

**Pointer Operators**

- Address operator, &
  - Unary operator (needs only 1 operand)
  - Returns the address of an operand

  ```
  int y = 5;
  int *yPtr;
  /* Assigns the address of the variable y to the pointer variable yPtr */
  yPtr = &y;
  ```

  - Example:

**Pointer vs. Addresses**

- Pointer variables actually hold the memory address of the location that they “point to”.

  - Memory addresses are assigned by the compiler so we generally don’t care about the specific address value stored in a pointer—we only need to know what it “points to”

**Indirection Operator**

- The "*" that we see in expressions using pointers is the indirection operator
  - Also called the dereferencing operator
  - Returns the value of the data to which the pointer is pointing

  ```
  printf("%d", *yPtr);
  /* prints 7 */
  ```

  - Example:

  ```
  #include <stdio.h>

  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\n", &a, aPtr);
    printf("The value of a is %d\n", a, *aPtr);
    printf("Showing that * and & are complements of each other: \n", &*aPtr, *&aPtr);
    return 0;
  }
  ```

  - Note: The %p format specifier allows us to print an address as a hexadecimal number
### Output from Preceding Program

The address of a is 7ffff030  
The value of aPtr is 7ffff030

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

Showing that * and & are complements of each other  
&*aPtr = 7ffff030

* &aPtr = 7ffff030

### Another Pointer Example

double flt1 = 100.0;  
double *flt_ptr;  
flt_ptr = &flt1;

- Note: flt_ptr does not store a double value but rather a pointer to another variable (flt1) that contains a double
- Pointer is useless unless it’s pointing to something
- Indirection with the * operator retrieves the value to which the pointer refers

### Pointer Assignments

```c
int *iptr, n=7, j=23, i=3;

iptr = &i;  
n = *iptr;  
*iptr = j;  
*iptr = *iptr + n + 10;

/* Trace: What is the value of *iptr? */
```

### Pointer Assignments

Solution: Trace all variable at each line of code

```c
int *iptr, n=7, j=23, i=3;

iptr = &i;  
n = *iptr;  
*iptr = j;  
*iptr = *iptr + n + 10;

*iptr = i + n + 10;
```

### Pointer Assignments

What is going on in the computer memory?

1. int *iptr, n=7, j=23, i=3;
2. iptr = &i;  
3. n = *iptr;  
4. *iptr = j;  
5. *iptr = *iptr + n + 10;

<table>
<thead>
<tr>
<th>Result of line 1.</th>
<th>Result of line 2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iptr</td>
<td>Address</td>
</tr>
<tr>
<td>j</td>
<td>6000</td>
</tr>
<tr>
<td>i</td>
<td>6004</td>
</tr>
<tr>
<td></td>
<td>6012</td>
</tr>
<tr>
<td></td>
<td>6020</td>
</tr>
</tbody>
</table>

| *iptr | Address | Value |
| j | 6000 | 7 |
| i | 6004 | 23 |
| | 6012 | 3 |
| | 6020 | |

### Pointer Assignments

What is going on in the computer memory?

1. int *iptr, n=7, j=23, i=3;
2. iptr = &i;  
3. n = *iptr;  
4. *iptr = j;  
5. *iptr = *iptr + n + 10;

<table>
<thead>
<tr>
<th>Result of line 3.</th>
<th>Result of line 4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iptr</td>
<td>Address</td>
</tr>
<tr>
<td>j</td>
<td>6000</td>
</tr>
<tr>
<td>i</td>
<td>6004</td>
</tr>
<tr>
<td></td>
<td>6012</td>
</tr>
<tr>
<td></td>
<td>6020</td>
</tr>
</tbody>
</table>

| *iptr, i | Address | Value |
| j | 6000 | 3 |
| i | 6004 | 23 |
| | 6012 | 3 |
| | 6020 | |
### Pointer Assignments

What is going on in the computer memory?

1. int *iptr, n=7, j=23, i=3;
2. iptr = &i;
3. n = *iptr;
4. *iptr = j;
5. *iptr = *iptr + n + 10;

Result of line 4.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>6012</td>
</tr>
<tr>
<td>6004</td>
<td>3</td>
</tr>
<tr>
<td>6008</td>
<td>23</td>
</tr>
<tr>
<td>6012</td>
<td>23</td>
</tr>
<tr>
<td>6016</td>
<td>6020</td>
</tr>
</tbody>
</table>

Result of line 5.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>6012</td>
</tr>
<tr>
<td>6004</td>
<td>3</td>
</tr>
<tr>
<td>6008</td>
<td>23</td>
</tr>
<tr>
<td>6012</td>
<td>36</td>
</tr>
<tr>
<td>6016</td>
<td>6020</td>
</tr>
</tbody>
</table>

*iptr = 23 + 3 + 10 = 36

### Pointer Expressions

- Pointers can be used in:
  - arithmetic expressions
  - assignment expressions
  - comparison expressions
- Not all operators are valid for use with pointer variables
- Before doing arithmetic on pointer variables, be sure that you know what you’re doing

### Pointer Operators

- Pointers can be:
  - incremented (++)
  - decremented (–-)
  - increased via integer addition (+ or +=)
  - decreased via integer subtraction (- or -=)
  - one pointer may be subtracted from another

### Pointer Arithmetic

- Assume integers take up 4 memory locations (bytes)

<table>
<thead>
<tr>
<th>2048</th>
<th>2052</th>
<th>2056</th>
<th>2060</th>
<th>2064</th>
<th>2068</th>
</tr>
</thead>
</table>

```c
int v[6], *vP;
vP = v;  /* vP = ?? */
vP += 2;  /* vP = ?? */
vP = &v[3];  /* vP = ?? */
```

### Pointer Arithmetic Answer

- In conventional arithmetic 2048 + 2 = 2050
- But in pointer arithmetic, 2048 + 2 * 4 = 2056!
  - where 4 is the number of memory locations for an int data type
- A double takes 8 memory locations and

```c
double *dPtr, d[10];
dPtr = d;  /* assume array starts at address 3000 */
dPtr += 2;  /* dPtr =3000 + 2 * 8 */
```

### More Pointer Arithmetic

- Using increment to point to next location in array

```c
vPtr++;
vPtr--;;
```
- Subtracting pointers:

```c
int v[6], x;
int *vPtr1, *vPtr2;
vPtr1 = v;  /* same as: vPtr1 = &v[0] */
vPtr2 = &v[3];  /* same as vPtr2 = vPtr1 + 3 */
```

```c
x = vPtr2 - vPtr1;
printf("x = %d\n", x);
```

Prints: x = 3
Pointer Comparison

- Pointers can be compared using
  - Equality (i.e. ==)
  - Relational operators (e.g. >)
- Meaningless unless pointers point to members of the same array or some other well-understood data structure
- Most common use of pointer comparison is determining whether a pointer is NULL
  ```c
  if (vPtr != NULL) ...
  ```

Pointer Assignments

- A pointer can be assigned to another pointer only if they are of the same data type.
- Otherwise, a cast operator must be used
  ```c
  float *floatPtr;
  double *doublePtr;
  doublePtr = (double *) floatPtr;
  ```
  - This can get you into a lot of trouble. Before casting pointers, be absolutely sure that you know what you are doing.

The pitfalls of casting pointers

```c
int main() {
    float *floatPtr;
    double *doublePtr;
    float f = 23.5678;
    double d = 23.5678;
    floatPtr = &f;
    doublePtr = &d;
    printf("The value of *floatPtr is %f 
", *floatPtr);
    printf("The value of *doublePtr is %f
", *doublePtr);
    doublePtr = (double *) floatPtr;
    printf("Now, the value of *doublePtr is %f
", *doublePtr);
    return 0;
}
```
Prints:
The value of *floatPtr is 23.567801
The value of *doublePtr is 23.567800
Now, the value of *doublePtr is 478862080.000000
Why???

Pointers and Arrays

- Pointer and array variables both refer to addresses in memory
- Arrays and pointers are (almost) always interchangeable
- An (unsubscripted) array name is actually a constant pointer
  - “points at” first element of the array
- Arrays cannot be assigned a new address to an array variable
- Pointers can be used to do any operation involving subscripting, just like arrays

Subscripting vs. pointer notation for array access

- There are four different ways to access elements of an array
  - Array subscript notation
  - Pointer + offset notation using array name
  - Pointer + offset notation
  - Pointer subscript notation

Array Subscript Notation

- Array subscript notation means the form
  ```c
  int n[10];
  n[1] = 5;
  ```
- Given:
  ```c
  int j, offset, c[ ] = {20, 30, 40, 50};
  int *cPtr;
  cPtr = c;
  printf("c[1] = %d
", c[1]);
  ```
  - What does this print? */
**Pointer Offset Notation with Array Name**
- Pointer offset notation means the form:
  ```c
  int n[10];
  *(n + 1) = 5;
  ```
- Given:
  ```c
  int j, offset, c[] = {20,30,40,50};
  int *cPtr;
  cPtr = c;
  offset = 2;
  printf("*(c + %d) = %d\n", offset, *(c + offset));
  /* What does this print? */
  ```

**Pointer Offset Notation**
- Pointer offset notation means the form
  ```c
  int n[10], *nPtr;
  nPtr = n;
  *(nPtr + 1) = 5;
  ```
- Given:
  ```c
  int j, offset, c[] = {20,30,40,50};
  int *cPtr;
  cPtr = &c[0]; /*note: different init*/
  offset = 3;
  printf("*(cPtr + %d) = %d\n", offset, *(cPtr + offset));
  /* What does this print? */
  ```

**Pointer Subscript Notation**
- Pointer subscript notation means the form
  ```c
  int n[10], nPtr;
  nPtr = n;
  nPtr[1] = 5;
  ```
- Given:
  ```c
  int j, offset, c[] = {20,30,40,50};
  int *cPtr = c;
  offset = 1;
  printf("cPtr[%d] = %d\n", offset, cPtr[1]);
  /* What does this print? */
  ```

**Notational Equivalences**
- Given:
  ```c
  int b[5];
  int *bPtr;
  ```
  ```c
  bPtr = b; is same as bPtr = &b[0];
  b[3] same as *(bPtr + 3) same as *(b+3) same as bPtr[3]
  Not the same: *(bPtr + 3) not same as *bPtr + 3
  The same: &b[3] same as bPtr + 3
  ```

**Generic (void) pointer**
- `void *` pointer
  - Pointer to void (i.e. void *)
  - Generic pointer that can represent any pointer type
    - All pointer types can be assigned a pointer to void,
    - A generic pointer can be assigned to any type
  - A generic pointer can not be dereferenced
    - Simply contains a memory location for an unknown data type (i.e. # of bytes unknown by compiler)
    - Also cannot do pointer arithmetic on generic pointers
  - These will be useful later for dynamic memory allocation

**Using the const qualifier with pointers**
- The const qualifier tells the compiler that the value of a particular variable should not be modified.
- Six possibilities exist for using (or not using) const with function parameters
  - two with call-by-value parameter passing
  - four with call-by-reference parameter passing
- Use principle of "least privilege" to choose which one to use.
  - Give a function enough access to the data in its parameters to accomplish its specified task, but no more.
Using the const qualifier

- Different ways to use the const qualifier to pass variables to a function using call-by-value
  - function1(int x)
    - read right to left: x is an integer
    - can change value of x in function
  - function2(const int x)
    - read right to left: x is a constant integer
    - can’t change value of x in function

- Different ways to use the const qualifier to pass variables to a function, call-by-reference
  - function3(int *xPtr)
    - read right to left: xPtr is an address of an integer, *xPtr is an integer
    - can change address stored in xPtr and can change the data stored at that address
  - function4(int * const xPtr)
    - read right to left: xPtr is an address to a constant integer, *xPtr is a const int
    - can change address stored in xPtr but can’t change the data stored at that address
  - function5(const int * xPtr)
    - read right to left: xPtr is an address to a constant integer, *xPtr is a const int
    - can change address stored in xPtr but can’t change the data stored at that address
  - function6(const int * const xPtr)
    - read right to left: xPtr is a constant address to a constant int, *xPtr is a const int
    - can’t change address stored in xPtr and can’t change the data stored at that address

Principle of Least Privilege

- If a variable does not (or should not) change in the body of a function to which it is passed, the variable should be declared const to ensure that it is not accidentally modified.
- Example:
  - Consider a function that takes a single-subscripted array and its size arguments and prints the array.
  - Neither the size of the array nor its contents should change in the function body.

Four ways to pass a pointer as an argument to a function

- non-constant pointer to non-constant data (highest level of access)
- constant pointer to non-constant data
- non-constant pointer to constant data
- constant pointer to constant data (lowest level of access)

/* Modified from Fig. 7.10: fig07_10.c Converting lowercase letters to uppercase letters using a non-constant pointer to non-constant data */
#include <stdio.h>
#include <ctype.h>
define STRSIZE 80;

void convertToUppercase( char *sPtr ); /* prototype */

int main() {
    char *string;
    string = (char *)malloc(sizeof(char)*STRSIZE);
    strcpy(string, "characters and $32.98");
    printf( "The string before conversion is: %s", string );
    convertToUppercase( string );
    printf( "\nThe string after conversion is: %s\n", string );
    return 0; /* indicates successful termination */
} /* end main */

/* convert string to uppercase letters */
void convertToUppercase( char *sPtr ) {
    while ( *sPtr != ‘\0’ ) {
        if ( islower( *sPtr ) ) { /* if lowercase, */
            *sPtr = toupper( *sPtr ); /* convert to */
            ++sPtr; /* move sPtr to the next character */
        } /* end while */
    } /* end function convertToUppercase */

Program Output:
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
/* Fig. 7.11: fig07_11.c
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() { /* initialize char array */
    char string[ ] = "print characters of a string";
    printf( "The string is:
" );
    printCharacters( string );
    printf( "\n" );
    return 0; /* indicates successful termination */
} /* end main */

/* sPtr cannot modify the character to which it points,
i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr ) {
    /* loop through entire string */
    for ( ; *sPtr != '\0'; sPtr++ ) {
        printf( "%c", *sPtr );
    } /* end for */
} /* end function printCharacters */

Program Output:
The string is:
print characters of a string

/* Fig. 7.12: fig07_12.c
Attempting to modify data through a
non-constant pointer to constant data. */
#include <stdio.h>

void f( const int *xPtr ); /* prototype */

int main() {
    int y; /* define y */
    f( &y ); /* f attempts illegal modification */
    return 0; /* indicates successful termination */
} /* end main */

/* 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*/
} /* end function f */

/* Fig. 7.13: fig07_13.c
Attempting to modify a constant pointer to
non-constant data */
#include <stdio.h>

int main() {
    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 */
    return 0; /* indicates successful termination */
} /* end main */

/* Fig. 7.14: fig07_14.c
Attempting to modify a constant pointer to
constant data. */
#include <stdio.h>

int main() {
    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;
    printf( "%d\n", *ptr );
    "ptr = 7; /* error: *ptr is const; cannot assign
new value */
    ptr = &y; /* error: ptr is const; cannot assign
new address */
    return 0; /* indicates successful termination */
} /* end main */