Today

- **Pointers**
  - Pointer Variable Declarations and Initialization
  - Pointer Operators
  - Pointers to void
  - Calling Functions by Reference
  - Passing parameters by reference
  - `sizeof` function
  - Dynamic Memory Management
  - Pointer Arithmetic
  - Pointers and Arrays
  - Pointers to Functions

Variables Revisited

- What actually happens when we declare variables?
  ```c
  char a;
  ```

- C reserves a byte in memory to store `a`.

- Where is that memory? At an `address`.

- Under the hood, C has been keeping track of variables and their addresses.

Pointers

- We can work with memory addresses too. We can use variables called **pointers**.

- A **pointer** is a variable that contains the address of a variable.

- Pointers provide a powerful and flexible method for manipulating data in your programs; but they are difficult to master.

- Close relationship with arrays and strings
**Benefits of Pointers**

- Pointers allow you to reference a large data structure in a compact way.
- Pointers facilitate sharing data between different parts of a program.
  - Call-by-Reference
- **Dynamic memory allocation**: Pointers make it possible to reserve new memory during program execution.

**Pointer Variable Declarations and Initialization**

- **Pointer declarations**
  - The syntax for pointer declaration is as follows:
    
    ```
    type *identifier;
    ```
    
    e.g. `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 the address of operand
    
    ```
    int y = 5;
    int *yPtr;
    yPtr = &y; // yPtr gets address of y
    ```
  - `yPtr` “points to” `y`

**Pointer Variable Declarations and Initialization**

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

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

### Diagram

- Address of `y` is value of `yptr`
**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

- \* and & are inverses
  - They cancel each other out

---

**Program Output**

```c
#include <stdio.h>

int main()
{
    int a;
    int *aPtr;
    a = 7;
    aPtr = &a;
    printf( "The address of a is %p\n", &a );
    printf( "The value of aPtr is %p\n", aPtr );
    printf( "Showing that \* and & are inverses of each other. \*aPtr = %p, \&aPtr = %p\n", \*aPtr, \&aPtr );
    return 0;
}
```

```
The address of a is 0012FF88
The value of aPtr is 0012FF88
The value of a is 7
The value of \*aPtr is 7
Showing that \* and & are inverses of each other.
\*aPtr = 0012FF88
\&aPtr = 0012FF88
```

---

**Operator Precedences – Revisited**

<table>
<thead>
<tr>
<th>Operators</th>
<th>Precedences</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>()</code></td>
<td>left to right</td>
<td>highest</td>
<td></td>
</tr>
<tr>
<td><code>[]</code></td>
<td>left to right</td>
<td>unary</td>
<td></td>
</tr>
<tr>
<td><code>*</code>, <code>&amp;</code></td>
<td>left to right</td>
<td>multiplicative</td>
<td></td>
</tr>
<tr>
<td><code>++</code>, <code>--</code>, <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>, <code>&amp;</code>, `</td>
<td><code>, </code>^<code>, </code>&lt;&lt;<code>, </code>&gt;&gt;`</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td><code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code></td>
<td>left to right</td>
<td>relational</td>
<td></td>
</tr>
<tr>
<td><code>==</code>, <code>!=</code></td>
<td>left to right</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>left to right</td>
<td>logical and</td>
<td></td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
<td>left to right</td>
</tr>
<tr>
<td><code>? :</code></td>
<td>right to left</td>
<td>conditional</td>
<td></td>
</tr>
<tr>
<td><code>,</code></td>
<td>left to right</td>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>comma</td>
<td></td>
</tr>
</tbody>
</table>
Addressing and Dereferencing

```c
int a, b, *p;
a = b = 7;
p = &a;
printf("p = %d\n", *p);
*p = 3;
printf("a = %d\n", a);
p = &b;
*p = 2 * *p - a;
printf("b = %d\n", b);
```

Program Output

```
p = 7
a = 3
b = 11
```

Addressing and Dereferencing

```c
float x, y, *p;
x = 5;
y = 7;
p = &x;
y = *p;
```

Thus,
```
y = *p;
y = *6x;
y = x;
```

All equivalent

Pointers to void

- **void** *identifier*;
- In C, **void** represents the absence of type.
- **void** pointers are pointers that point to a value that has no specific type.
- This allows void pointers to point to any data type.
- The data pointed by void pointers cannot be directly dereferenced.
- We have to use explicit type casting before dereferencing it.

<table>
<thead>
<tr>
<th>Declarations and initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int k=3, j=5, *p = &amp;k, *q = &amp;j, *r;</td>
</tr>
<tr>
<td>double x;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p == &amp;k</td>
<td>p == (&amp;k)</td>
<td>1</td>
</tr>
<tr>
<td>p = x + 7</td>
<td>p = (x + 7)</td>
<td>illegal</td>
</tr>
<tr>
<td>* * p</td>
<td>(* (*p))</td>
<td>3</td>
</tr>
<tr>
<td>r = &amp;x</td>
<td>r = (&amp;x)</td>
<td>illegal</td>
</tr>
<tr>
<td>7 * * p / *q + 7</td>
<td>((7 * (*p)) / (*q)) + 7</td>
<td>11</td>
</tr>
<tr>
<td>* (r = &amp;x) * = *p</td>
<td>(* (r = (&amp;x))) * = (*p)</td>
<td>15</td>
</tr>
</tbody>
</table>
### Pointers to void

```c
int x = 4;
void *q = &x;
int *p = q;
int i = *p;
int j = *(int*)q;
```

<table>
<thead>
<tr>
<th>Legal assignments</th>
<th>Illegal assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 0;</td>
<td>p = 1;</td>
</tr>
<tr>
<td>p = (int *) 1;</td>
<td>v = 1;</td>
</tr>
<tr>
<td>p = v = q;</td>
<td>p = q;</td>
</tr>
<tr>
<td>p = (int *) q;</td>
<td></td>
</tr>
</tbody>
</table>

### Declarations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int *p;</td>
<td></td>
</tr>
<tr>
<td>float *q;</td>
<td></td>
</tr>
<tr>
<td>void *v;</td>
<td></td>
</tr>
</tbody>
</table>

### 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
    ```c
    void double_it(int *number) {
        *number = 2 * (*number);
    }
    ```
  - **number** used as nickname for the variable passed

### Passing parameters by reference

```c
void SetToZero (int var) {
    var = 0;
}
```

- You would make the following call:
  ```c
  SetToZero(x);
  ```
- This function has no effect whatever. Instead, pass a pointer:
  ```c
  void SetToZero (int *ip) {
      *ip = 0;
  }
  ```
  You would make the following call:
  ```c
  SetToZero(&x);
  ```

This is referred to as **call-by-reference**.
/* Cube a variable using call-by-reference 
with a pointer argument */
#include <stdio.h>

void cubeByReference( int * ); /* prototype */

int main()
{
    int number = 5;
    printf( "The original value of number is %d", number );
    cubeByReference( &number );
    printf( "The new value of number is %d\n", number );
    return 0;
}

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

Program Output

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

/* Swapping arguments (incorrect version) */
#include <stdio.h>

void swap ( int p, int q );

int main ( void )
{
    int a = 3;
    int b = 7;
    printf("%d  %d\n", a, b);
    swap(a,b);
    printf("%d  %d\n", a, b);
    return 0;
}

void swap ( int p, int q )
{
    int tmp;
    tmp = p;
    p = q;
    q = tmp;
}

/* Swapping arguments (correct version) */
#include <stdio.h>

void swap ( int *p, int *q);

int main ( void )
{
    int a = 3;
    int b = 7;
    printf("%d  %d\n", a, b);
    swap(&a, &b);
    printf("%d  %d\n", a, b);
    return 0;
}

void swap ( int *p, int *q )
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
/* This function separates a number into three parts: a sign (+, -, * or blank), a whole number magnitude and a fraction part.
* Preconditions: num is defined; signp, wholep and fracp contain
* addresses of memory cells where results are to be stored
* Postconditions: function results are stored in cells pointed to by
* signp, wholep, and fracp
*/
void separate(double num, char *signp, int *wholep, double *fracp)
{
    double magnitude;
    if (num < 0)
        *signp = '-';
    else if (num == 0)
        *signp = ' ';
    else
        *signp = '+';
    magnitude = fabs(num);
    *wholep = floor(magnitude);
    *fracp = magnitude - *wholep;
}

int main()
{
    double value;
    char sn;
    int whl;
    double fr;
    /* Gets data */
    printf("Enter a value to analyze:");
    scanf("%lf", &value);
    /* Separates data value in three parts */
    separate(value, &sn, &whl, &fr);
    /* Prints results */
    printf("Parts of %.4f
", value, sn);
    printf("whole number magnitude: %d
", whl);
    printf("fractional part : %.4f
", fr);
    return 0;
}

Example
/* This program puts values into an array, sorts the values into
ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int *array, const int size );
void swap( int *element1Ptr, int *element2Ptr );

int main()
{
    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    int i;
    printf( "Data items in original order:
" );
    for ( i = 0; i < SIZE; i++ )
        printf( "%d", a[ i ] );
    bubbleSort( a, SIZE ); /* sort the array */
    printf( "Data items in ascending order:
" );
}
sizeof function

- **sizeof**
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if sizeof( int ) equals 4 bytes, then
    ```c
    int myArray[ 10 ];
    printf( "%d", sizeof( myArray ) );
    ```
    will print 40
  - **sizeof** can be used with
    - Variable names
    - Type name
    - Constant values
Example

```c
/* Demonstrating the sizeof operator */
#include <stdio.h>

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

    printf( "
        sizeof c = %d
        sizeof(char)  = %d
        sizeof s = %d
        sizeof(short) = %d
        sizeof i = %d
        sizeof(int)  = %d
        sizeof l = %d
        sizeof(long) = %d
        sizeof f = %d
        sizeof(float) = %d
        sizeof d = %d
        sizeof(double) = %d
        sizeof ld = %d
        sizeof(long double) = %d
        sizeof array = %d
        sizeof ptr = %d
",
        sizeof c, sizeof(char),
        sizeof s, sizeof(short),
        sizeof i, sizeof(int),
        sizeof l, sizeof(long),
        sizeof f, sizeof(float),
        sizeof d, sizeof(double),
        sizeof ld, sizeof(long double),
        sizeof array, sizeof ptr);

    return 0;
}
```

Dynamic Memory Management

- **Static memory allocation**: space for the object is provided in the binary at compile-time.
- **Dynamic memory allocation**: blocks of memory of arbitrary size can be requested at run-time.

- The four dynamic memory management functions are `malloc`, `calloc`, `realloc`, and `free`.
- These functions are included in the header file `<stdlib.h>`.
Dynamic Memory Management

- **void *malloc(size_t size);**

  - allocates storage for an object whose size is specified by `size`:
    - It returns a pointer to the allocated storage,
    - `NULL` if it is not possible to allocate the storage requested.
    - The allocated storage is not initialized in any way.

  - e.g. `float *fp, fa[10];`
    `fp = (float *) malloc(sizeof(fa));`
  allocates the storage to hold an array of 10 floating-point elements, and assigns the pointer to this storage to `fp`.

Dynamic Memory Management

- **void *calloc(size_t nobj, size_t size);**

  - allocates the storage for an array of `nobj` objects, each of size `size`.
  - It returns a pointer to the allocated storage,
  - `NULL` if it is not possible to allocate the storage requested.
  - The allocated storage is initialized to zeros.

  - e.g. `double *dp, da[10];`
    `dp = (double *) calloc(10, sizeof(double));`
  allocates the storage to hold an array of 10 `double` values, and assigns the pointer to this storage to `dp`.

Dynamic Memory Management

- **void *realloc(void *p, size_t size);**

  - changes the size of the object pointed to by `p` to `size`.
  - It returns a pointer to the new storage,
  - `NULL` if it is not possible to resize the object, in which case the object (*p*) remains unchanged.
  - The new size may be larger (the original contents are preserved and the remaining space is unitialized) or smaller (the contents are unchanged up to the new size) than the original size.

  - e.g. `char *cp;`
    `cp = (char *) malloc(sizeof("computer"));`
    `strcpy(cp, "computer");`
  discards the trailing `'\0'` and makes `cp` point to an array of 8 characters containing the null-terminated string `computer`.

    `cp = (char *) realloc(cp, sizeof("compute"));`
  discards the trailing `'\0'` and makes `cp` point to an array if 8 characters containing the characters in `computer`

    `cp=(char *)realloc(cp,sizeof("computerization"));`
  discards the trailing `'\0'` and makes `cp` point to an array of 16 characters, the first 9 of which contain the null-terminated string `computer` and the remaining 7 are uninitialized.
Dynamic Memory Management

- `void *free(void *p);`
- deallocates the storage pointed to by p, where p is a pointer to the storage previously allocated by `malloc`, `calloc`, or `realloc`.
- e.g. `free(fp);`
- `free(dp);`
- `free(cp);`

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 ints
  - `vPtr` points to first element `v[0]`
  - i.e. location 3000 (`vPtr = 3000`)
  - `vPtr += 2;` sets `vPtr` to 3008
  - `vPtr` points to `v[2]` (incremented by 2), but the machine has 4 byte ints, so it points to address 3008

```
  location 3000 3004 3008 3012 3016
  \      \    \     \      \  

  pointer variable vPtr
```

Pointer Expressions and Pointer Arithmetic

- Subtraction points
  - Returns number of elements from one to the other. If
    - `vPtr = &v[0];`
    - `vPtr2 = &v[2]; //vPtr2 = vPtr + 2;`
    - `vPtr2 - vPtr` would produce 2
  - Pointer comparison (`<`, `==`, `>`)  
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to 0
Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to void (type void *)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to void pointer
  - void pointers cannot be dereferenced

Example

```c
int SumIntegerArray(int *ip, int n)
{
    int i, sum;
    sum = 0;
    for (i=0; i < n; i++) {
        sum += *ip++;
    }
    return sum;
}
```

Assume
```
int sum, list[5];
```
are declared in the main function. We can make the following function call:
```
sum = SumIntegerArray(list, 5);
```

Example

```c
#include<stdio.h>
#include<stdlib.h>

int main(void) {
    int array, *p;
    int i, no_elements;
    printf("Enter number of elements: ");
    scanf("%d", &no_elements);
    printf("Enter the elements: ");
    array = (int*) malloc(no_elements * sizeof(int));
    for(p=array, i=0; i < no_elements; i++, p++)
        scanf("%d", p);
    printf("Elements: ");
    for(p=array, i=0; i < no_elements; i++, p++)
        printf("%d ",*p);
    printf("\n");
}
```
Example

```c
array = ( int*) realloc( array, (no_elements+2)*sizeof( int ) );

printf("Enter two new elements: ");
for(p= array, i=0; i<no_elements; i++, p++) ;
for( i<no_elements+2; i++, p++)
    scanf("%d",p);
printf("Elements: ");
for(p= array, i=0; i<no_elements+2; i++, p++)
    printf("%d ",*p);
printf("\n");
free(array);
return 0;
```

Enter number of elements: 4
Enter the elements: 2 3 4 5
Elements: 2 3 4 5
Enter two new elements: 6 7
Elements: 2 3 4 5 6 7

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 defined
  - 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

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

void convertToUppercase( char *sPtr);

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

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

Example

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

The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
Example

/* 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:\n" );
    printCharacters( string );
    printf( "\n" );
    return 0;
}

Example

/* Attempting to 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 function printCharacters */

Program Output

The string is:
print characters of a string

Example

/* Attempting to modify a constant pointer to non-constant data */

#include <stdio.h>

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

    int *const ptr = &x;

    *ptr = 100; /* allowed: *ptr is not const */
    ptr = &y; /* error: ptr is const; cannot assign new address */
    return 0;
} /* end main */

Example

/* Attempting to modify a constant pointer to non-constant data */

#include <stdio.h>

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

    int *const ptr = &x;

    *ptr = 100; /* allowed: *ptr is not const */
    ptr = &y; /* error: ptr is const; cannot assign new address */
    return 0;
} /* end main */

Example

/* 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;

    f( &y );     /* f attempts illegal modification */
    return 0;
} /* 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 */

Example

/* Attempting to modify a constant pointer to non-constant data */

#include <stdio.h>

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

    int *const ptr = &x;

    *ptr = 100; /* allowed: *ptr is not const */
    ptr = &y; /* error: ptr is const; cannot assign new address */
    return 0;
} /* end main */

Example

/* 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;

    f( &y );     /* f attempts illegal modification */
    return 0;
} /* 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 */
The Relationship between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name is 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`

Pointers and Arrays

- Arrays are implemented as pointers.
- Consider:
  ```c
  double list[3];
  list[1] : is the address of the second element
  list[i] : the address of list[i] which is calculated by the formula
  base address of the array + i * 8
  ```
Example (cont.)

/* Using subscripting and pointer notations with arrays */
#include <stdio.h>
int main(void)
{
    int i, offset, b[4]={10,20,30,40};
    int *bPtr = b;
    /* Array is printed with array subscript notation */
    for (i=0; i < 4; i++)
        printf("b[%d] = %d\n", i, b[i]);

    /* Pointer/offset notation where the pointer is the array name */
    for (offset=0; offset < 4; offset++)
        printf("*(b + %d) = %d\n", offset, *(b + offset));

    /* Pointer subscript notation */
    for (i=0; i < 4; i++)
        printf("bPtr[%d] = %d\n", i, bPtr[i]);

    /* Pointer offset notation */
    for (offset = 0; offset < 4; offset++)
        printf("*(bPtr + %d) = %d\n", offset, *(bPtr + offset));
    return 0;
}

Example (cont.)

/* Copying a string using array notation and pointer notation. */
#include <stdio.h>
void copy1(char *s1, const char *s2);
void copy2(char *s1, const char *s2);
int main()
{
    char string1[10]; /* create array string1 */
    char *string2 = "Hello"; /* create a pointer to a string */
    char string3[10]; /* create array string3 */
    char string4[] = "Good Bye"; /* create a pointer to a string */
    copy1(string1, string2);
    printf("string1 = %s\n", string1);
    copy2(string3, string4);
    printf("string3 = %s\n", string3);
    return 0;
}

Example (cont.)

/* Copying a string using array notation and pointer notation. */
#include <stdio.h>
void copy1(char *s1, const char *s2);
void copy2(char *s1, const char *s2);
int main()
{
    char string1[10]; /* create array string1 */
    char *string2 = "Hello"; /* create a pointer to a string */
    char string3[10]; /* create array string3 */
    char string4[] = "Good Bye"; /* create a pointer to a string */
    copy1(string1, string2);
    printf("string1 = %s\n", string1);
    copy2(string3, string4);
    printf("string3 = %s\n", string3);
    return 0;
}
Example

/* copy s2 to s1 using array notation */
void copy1(char *s1, const char *s2)
{
    int i;
    for ( i = 0; s1[i] = s2[i] != '\0'; i++)
    {} /* end function copy1 */

/* copy s2 to s1 using pointer notation */
void copy2(char *s1, const char *s2)
{
    /* loop through strings */
    for ( ; *(s1++) = *(s2++) != '\0';
    {} /* end function copy2 */

Arrays of Pointers

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

Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Example: `bubblesort`

- Function `bubble` takes a function pointer
- `bubble` calls this helper function
- this determines ascending or descending sorting
- The argument in `bubblesort` for the function pointer:
  ```c
  int (*compare)(int a, int b)
  ```
  tells `bubblesort` to expect a pointer to a function that takes two `ints` and returns an `int`
- If the parentheses were left out:
  ```c
  int *compare( int a, int b )
  ```
  - Defines a function that receives two integers and returns a pointer to a `int`
Example

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

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

int main() {
    int order; /* 1 for ascending order or 2 for descending order */
    int counter; /* counter */
    /* initialize array a */
    int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
    printf("Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: ");
    scanf("%d", &order);
    printf("\nData items in original order\n");
    for (counter = 0; counter < SIZE; counter++)
        printf("%5d", a[counter]);
    /* sort array in ascending order; pass function ascending as an argument */
    if (order == 1) {
        bubble(a, SIZE, ascending);
        printf("\nData items in ascending order\n");
    } else { /* pass function descending */
        bubble(a, SIZE, descending);
        printf("\nData items in descending order\n");
    } /* output sorted array */
    for (counter = 0; counter < SIZE; counter++)
        printf("%5d", a[counter]);
    printf("\n");
    return 0;
}

Example

/* multipurpose bubble sort; parameter compare is a pointer to 
the comparison function that determines sorting order */
void bubble(int work[], const int size, int (*compare)(int a, int b))
{
    int pass; /* pass counter */
    int count; /* comparison counter */
    void swap(int *element1Ptr, int *element2Ptr);
    for (pass = 1; pass < size; pass++) {
        for (count = 0; count < size - 1; count++) {
            /* if adjacent elements are out of order, swap them */
            if ((*compare)(work[count], work[count + 1])) {
                swap(&work[count], &work[count + 1]);
            }
        }
    } /* end function bubble */

Example

/* multipurpose bubble sort: parameter compare is a pointer to 
the comparison function that determines sorting order */
void bubble(int work[], const int size, int (*compare)(int a, int b))
{
    int pass; /* pass counter */
    int count; /* comparison counter */
    void swap(int *element1Ptr, int *element2Ptr);
    for (pass = 1; pass < size; pass++) {
        for (count = 0; count < size - 1; count++) {
            /* if adjacent elements are out of order, swap them */
            if ((*compare)(work[count], work[count + 1])) {
                swap(&work[count], &work[count + 1]);
            }
        }
    } /* end function bubble */

Example

/* output original array */
for (counter = 0; counter < SIZE; counter++)
    printf("%5d", a[counter]);
/* sort array in ascending order; pass function ascending as an argument */
if (order == 1) {
    bubble(a, SIZE, ascending);
    printf("\nData items in ascending order\n");
} else { /* pass function descending */
    bubble(a, SIZE, descending);
    printf("\nData items in descending order\n");
} /* output sorted array */
for (counter = 0; counter < SIZE; counter++)
    printf("%5d", a[counter]);
return 0;

Example

/* swap values at memory locations to which element1Ptr and element2Ptr point */
void swap(int *element1Ptr, int *element2Ptr)
{
    int hold; /* temporary holding variable */
    hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
} /* end function swap */
/* determine whether elements are out of order for an ascending order sort */
int ascending(int a, int b) {
    return b < a;
} /* end function ascending */
/* determine whether elements are out of order for a descending order sort */
int descending(int a, int b) {
    return b > a; /* swap if b is greater than a */
} /* end function descending */
Example

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 descending order
89   68   45   37   12   10    8    6    4    2

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 descending order
89   68   45   37   12   10    8    6    4    2

Program Output