Pointers are also used in C to enable a function to modify a variable held by the caller:

```c
void findExtrema(const int *A, int Sz, int *Min, int *Max) {
    *Min = *Max = A[0];           // prime the min/max values
    for (int idx = 1; idx < Sz; idx++) {
        int Current = A[idx];      // avoid extra array
        //    index operations
        if ( Current < *Min )
            *Min = Current;
        else if ( Current > *Max )
            *Max = Current;
    }
}
```
Pointers are also used in C to enable a function to modify a variable held by the caller:

```c
void findExtrema(const int *A, int Sz, int *pMin, int *pMax) {

    *pMin = *pMax = A[0];           // prime the min/max values

    for (int idx = 1; idx < Sz; idx++) {

        int Current = A[idx];      // avoid extra array
                                   //    index operations
        if ( Current < *pMin )
            *pMin = Current;
        else if ( Current > *pMax )
            *pMax = Current;
    }

    // calling side:

    int List[5] = {34, 17, 22, 89, 4};
    int lMin = 0, lMax = 0;

    findExtrema(List, 5, &lMin, &lMax);
}
```
Pointers can also be used as return values:

```c
double* createArray(int Sz) {
    double *p = malloc( Sz * sizeof(double));
    if ( p != NULL ) {
        for (int idx = 0; idx < Sz; idx++)
            p[idx] = 0.0;
    }
    return p;  // ownership goes to caller
}
```

```c
double *Array = createArray(1000);
...
But… NEVER return a pointer to an automatic local object:

```c
int* F() {
    int Local = rand() % 1000;
    // Local ceases to exist when F() executes its return, since Local has automatic storage duration.
    return &Local;
}
```

```c
int* p = F();
```

C:\Code> gcc-4 -o P5 -std=c99 P5.c

P5.c: In function 'F':
P5.c:32: warning: function returns address of local variable
**Pointers and const**

`const` can be applied in interesting ways in pointer contexts:

```c
int* p;                // pointer and target can both be changed
const int* p;          // pointer can be changed; target cannot
int* const p;          // target can be changed; pointer cannot
const int* const p;    // neither pointer nor target can be changed
```

In the latter two cases, unless you are declaring a parameter, you must initialize the pointer in its declaration.

This provides safety against inadvertent changes to a pointer and/or its target, and is certainly an under-used feature in C.

```c
void findExtrema(const int* const A, int Sz, int* const Min,
                 int* const Max);
```
In C, a pointer may be declared of type `void`:

```c
void* p; // target can be of ANY type; so no compile-time type-checking occurs
```

`void` pointers are not useful in many situations:

- the return value from `malloc()` is actually a `void*`
- they can be used to achieve generic programming, often with data structures, but also with a number of useful functions:

```c
void* memcpy(void* s1, const void* s2, size_t n);
```

// The memcpy function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.
// Returns: the memcpy function returns the value of s1.
A pointer can point to a pointer. One use of this is to pass a pointer so that a function can modify it:

```c
void createArray(double** const A, int Sz) {
    double* p = malloc( Sz * sizeof(double));
    if ( p != NULL ) {
        for (int idx = 0; idx < Sz; idx++)
            p[idx] = 0.0;
    }
    *A = p;
}
```

```c
double *Array;
createArray(&Array, 1000);
...```

Dereferencing a Pointer

We said earlier that dereferencing a pointer yields the target of the pointer.

But, there's a bit more to it than that… the C Standard says that:

- if the operand \( p \) points to an object then the result of \( *p \) is a *lvalue* designating the object
- if the operand \( p \) is of type "pointer to type" then the result of \( *p \) has type *type*

(An *lvalue* is "an expression … that potentially designates an object".)
Suppose that we have a region of memory initialized as shown below, and a pointer \( p \) whose target is the first byte of the region:

\[
\begin{array}{c}
\text{uint8_t } * p8 \\
\hline
00000001 \\
00000010 \\
00000011 \\
00000100 \\
00000101 \\
00000110 \\
00000111 \\
\cdots
\end{array}
\]

Then \( *p8 \) would evaluate to the single byte 00000001.
Now suppose that we have a region of memory initialized as shown below, and a pointer \( p_{16} \) whose target is the first byte of the region:

\[
\text{uint16_t* } p_{16}
\]

Then \( *p_{16} \) would evaluate to the two-byte value

\[
000000001 \quad 00000010
\]
Now suppose that we have a region of memory initialized as shown below, and a pointer \( p \) whose target is the first byte of the region:

\[
\begin{array}{c}
\text{uint8_t* } p \\
\end{array}
\]

Then we can apply a typecast to the pointer \( p \) to access two bytes:

\[
*( (\text{uint16_t*}) p )
\]

The expression above evaluates to the two-byte value:

\[
000000001 \quad 00000010
\]
To generalize, size matters: