Pass-by-Pointer

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

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

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

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

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

Pass-by-Pointer

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

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

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

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

      int Current = pA[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);
```

Returning a Pointer (Good)

Pointers can also be used as return values:

```
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
}
```

```
. . .
double *Array = createArray(1000);
. . .
```

Returning a Pointer (Bad)

But… NEVER return a pointer to an automatic local object:

```
int* F() {

   int Local = rand() % 1000;

   // Local ceases to exist when F()
   // executes its return, since Local has
   // automatic storage duration.

   return &Local;
}
```

```
. . .
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:

```
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.

```
void findExtrema(const int* const pA, int Sz, int* const pMin,
                  int* const pMax);
```

Using const with Pointers

Here's an improved version of the findExtrema() function:

```
void findExtrema(const int * const pA, // 1
                 int Sz,
                 int * const pMin,     // 2
                 int * const pMax) {
   . . .
}
```

1: Now, the function cannot make pA point to anything else, nor can it change the values in the array that pA points to.

2: Now, the function cannot make pMin or pMax point to anything else, but we do need to let it change the values of the targets of pMin and pMax.

void Pointers

In C, a pointer may be declared of type void:

```
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:

```
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.
```

Pointers to Pointers

A pointer can point to a pointer. One use of this is to pass a pointer so that a function can modify it:

```
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;
}
```

```
. . .
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".)

Pointer Casts

Pointer typecasting can be used to define the amount of data dereferencing yields.

Suppose that you run a program and give it your PID as a parameter:

```
CentOS > prog wmcquain
```

```
argv[0] --> "prog"
argv[1] --> "wmcquain"
```

Then suppose the code in `main()` does this:

```
uint32_t limit = (uint32_t)(*(uint32_t*)argv[1]);
```

Pointer Casts

```
 . . . (uint32_t*)argv[1]);
```

The pointer cast takes the pointer `argv[1]` and produces a nameless pointer of type `uint32_t*`

```
. . . *(uint32_t*)argv[1]);
```

Dereferencing that pointer yields 4 bytes of data, because the target of a `uint32_t*` is 4 bytes in size.

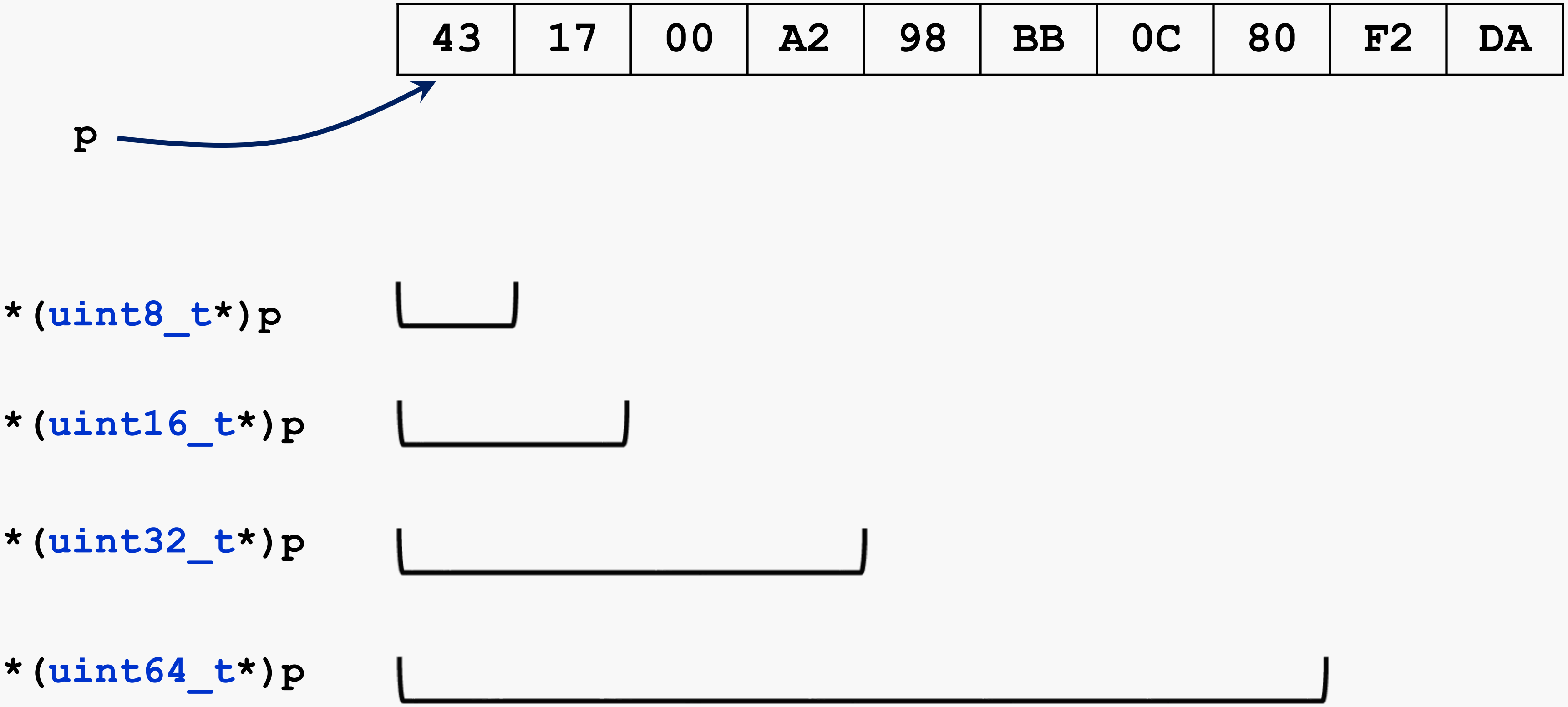
```
 . . .(uint32_t)(*(uint32_t*)argv[1]);
```

The final typecast tells the compiler to interpret those 4 bytes as representing an unsigned integer value.

```
"wmcq" --> 77 6D 63 71            0x71636D77 --> 1902341495
```

Pointer Casts

Suppose the pointer p points to the beginning of a memory region:



43	17	00	A2	98	BB	0C	80	F2	DA

```
*(uint8_t*)p
*(uint16_t*)p
*(uint32_t*)p
*(uint64_t*)p
```