Generic pointers, cont.

• You can't dereference a `void *` - you first need to cast or assign it to a real pointer type
  – the value obtained from a dereference depends on the type of pointer
• `NULL` is really defined as `(void *) 0`
• These allow use of generic code, but misuse can lead to the kinds of errors we've seen before:

  ```c
  void *vp;
  int *ip;
  double a = 3.14159;
  vp = &a;
  ip = vp;
  printf("%d\n", *ip);  /* -266631570 */
  ```

Notes

• Project 1 is being graded for style
  – will be returned before Project 2 due
• Project 2 posted, due Mon., Feb. 27
  – public tests posted and submit server open
• Read Reek, Chapter 6: Pointers, Chapter 11: Dynamic Memory Allocation
Type conversion with pointers

- Converting from one type to a pointer has some uses:
  ```c
  unsigned int i;
  unsigned char *ch;
  i = 0x543210ab;
  ch = (unsigned char *) &i;
  printf("%d\n", *ch);
  printf("%d\n", *(unsigned char *) &i);
  ```
- Prints out either MSB or LSB of i, depending on architecture

Type conversion, cont.

- Type conversion is very similar to what happens when we access an inactive union field
  ```c
  union {
    int i;
    double dbl;
  } a;
  double fp_val = 3.14159;
  a.dbl = 3.14159;
  printf("%d\n", a.i);  /* -266631570 */
  printf("%d\n", *(int *) &fp_val);
  ```

The const modifier

- Indicates that a variable can’t be changed, and enforced by compiler
  ```c
  int const i = 4;
  const int j = 5;
  i++;  /* ERROR */
  j++;  /* ERROR */
  ```
- Order of type specifier and const modifier does matter when dealing with pointers:
  ```c
  int i = 4, j = 5;
  const int *p = &i;  /* pointer to constant int */
  int * const q = &j; /* constant pointer to int */
  p = &j;  /* OK */
  *p += 5;  /* ERROR */
  q = &i; /* ERROR */
  *q += 23;  /* OK */
  ```
- The program cdecl can be useful for decoding some more complex declarations
Incrementing pointers

• Pointers can be incremented/decremented just like integer type variables, "moving" one element at a time
  – how much is added to the address depends on the size of the type to which the pointer points (as declared)
• Recall arrays are contiguous memory
• What does this function do?
  ```c
  int mystery(int array[]) {
    int *p = &array[0];
    int sum = 0;
    while (*p != -1) {
      sum += *p;
      p++;
    }
    return sum;
  }
  ```

Incrementing pointers, cont.

size_t strlen(const char *str) {
  size_t len = 0;
  while (*str) {
    len++;
    str++;
  }
  return len;
}
• Why can we move the str parameter?
• Why does this return the string’s length?

Pointer arithmetic

• With two pointers in the same array, we can determine how far apart they are
  ```c
  size_t strlen(const char *str) {
    const char *ptr;
    for (ptr = str; *ptr; ptr++)
      ;
    return (size_t) (ptr - str);
  }
  ```
Pointer arithmetic, cont.

• By adding an integer \( n \) to a pointer, we can get the address of the \( n^{th} \) element past the element to which the pointer currently points

```c
int arr[] = {2, 3, 5, 7, 11};
int *p = &arr[0];
int *q = p + 4;
printf("%d\n", *q);
Output: 11
```

• Only valid forms of pointer arithmetic:
  – pointer - pointer
  – pointer ± integer

Pointer arithmetic, cont.

• We can also use relational and equality operators when working with multiple pointers

```c
void sum_subarray(int array[], int idx1, int idx2) {
    int *ptr;
    int sum = 0;
    ptr = array + idx1;
    while (ptr <= array + idx2) {
        sum += *ptr;
        ptr++;
    }
    return sum;
}
```