Administrivia

• Read Reek, Chapter 6: Pointers
Chapter 6, Reek

**POINTERS (CONT.)**
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
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.

- The postfix operators take precedence over the dereference operator and prefix operators.
- * and prefix ops are at the same precedence level, and associate right to left.
- ++*p increments the value at the location to which p points, and evaluates to the incremented value.
- *p++ evaluates to the value at the location to which p points, and then advances p.
- (*p) ++ evaluates to the value at the location to which p points, and then increments that value.
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 diagram](attachment:pointer_arithmetic.png)
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;
}
```
(Not in the textbooks)

MAKE
Separate Compilation

• Placing code across several files allows us to update only parts of the object code for a program when changes occur

• We have been just compiling everything into an executable, bypassing the object code step

• Ex. from project #1:

  gcc -c puzzles.c
  gcc -c public01.c
  gcc -o public01 public01.o puzzles.o

• Now, if we change **puzzles.c**, we only need to recompile half our program, and then link the object files together
The make utility

• Using make helps automate separate compilation, using a Makefile we write
• The Java compiler can tell if it has to compile other classes than the one you've told it to compile; C compilers can't
• Typing out the commands to build C programs is error-prone and can get annoying, especially if using several command-line options
• Note: this is a compressed description of Make; full manual is available online off of course information page
Example

• Recall Project #1 again:

publicX.c:
#include <stdio.h>
#include "puzzles.h"

int main() {
    size_t x = sizeof_long();
    ...
    if (is_nonzero(0))
        ...
    else
        ...
    return 0;
}

puzzles.h:
#include <stddef.h>

int is_nonzero(int);
size_t sizeof_long();
...

puzzles.c:
#include "puzzles.h"

int is_nonzero(int a) {
    ...
}
size_t sizeof_long() {
    ...
}
...
Dependencies

• Since `main()` is in `publicX.c`, we'll name the program `publicX`
• What needs to be compiled if ...
  – `publicX.c` is changed?
  – `puzzles.c` is changed?
  – `puzzles.h` is changed?
Basic dependency rules

• Executables depend on all the object files that could compose the program
• Executables made by linking object files
• Object files depend on their respective source files \((x.o \text{ depends on } x.c)\), and any header files \#included with quotes in the source file
  – \(x.o \text{ depends on } y.h \text{ if } x.c \text{ has } "\#include "y.h""
• Object files made by compiling \(*.c\) files with \(-c\) flag
Example Makefile

publicX: publicX.o puzzles.o
        gcc -o publicX publicX.o puzzles.o

publicX.o: publicX.c puzzles.h
        gcc -c publicX.c

puzzles.o: puzzles.c puzzles.h
        gcc -c puzzles.c

Simplified Makefile notes:

• Pairs of lines are called rules
• Each rule (usually) has a target, a list of dependencies, and one or more commands
• The target is the item before the colon
• The dependency list contains those items that, if changed, should cause rebuilding of the target
• Lines with commands must begin with a tab character, and commands should perform the necessary actions to rebuild the target

must be a tab character!