Several New Things

- 2’s complement representation of negative integers
- The data size flag in the conversion specification of printf
- Recast of &n to \textit{unsigned long} to get the address
Casts

- \((T) \ x\)
  - A cast converts the value held in variable \(x\) to type \(T\)
  - With pointers, casts do not affect the content of the variable pointed (merely an indication to the compiler):
    - \(\text{char}^* \ c; \ \text{int}^* \ i;\)
    - \(i = (\text{int}^*) \ c;\)
Flags (in any order), which modify the specification:

- `-`, which specifies left adjustment of the converted argument in its field.
- `+`, which specifies that the number will always be printed with a sign.
- `0`: for numeric conversions, specifies padding to the field width with leading zeros.
- `#`, which specifies an alternate output form. For `0`, the first digit will become zero. For `x` or `X`, `0x` or `0X` will be prefixed to a non-zero result.
2’s complement

• We won’t get into lots of examples of computer arithmetics
• But suffices to say that, under 2’s complement
  – The binary representation of negation of an integer $n$ is obtained by
    • First negate $n$ bit-wise
    • Add 1 to the least significant bit
  
• Example (8-bit data representation)
  – 1’s bit pattern is 0000 0001
  – Negation result: 11111110
  – Add binary 1, we get: 11111111

• Example
  – What is -1’s negation under 2’s complement?
  – What is 0’s negation under 2’s complement?
#include <stdio.h>

main() {

    int c = 0, in = 0;
    char buf[2048]; char *p = buf;

    while((c = getchar()) != EOF)
        *p++=c;
    *p++ = '\0';
    printf("buffer is \t %s\n", buf);

}
• You can increment a pointer
• But you cannot increment an array name
Pointer increment

- Earlier we used p++ to step through an array
- In C, a pointer is simply a memory address
  - How much does p++ increment the address?
  - We now use printf to investigate
Stepping through an integer array

#include <stdio.h>

main() {

    int i, ndigit[10], *p, *end;
    for (i = 0; i < 10; ++i)
        ndigit[i] = i;
    end = &ndigit[10];
    p = ndigit;
    while (p != end) {
        printf("ndigit is at \t \%lX\n", (unsigned long) p);
        printf("ndigit is at \t \%p\n", p);
        p++;
    }
}
/* %p display is “implementation-dependent” */
Stepping through an array of char

#include <stdio.h>

main() {
    int i;
    char ndigit[10], *p, *end;
    for (i = 0; i < 10; ++i)
        ndigit[i] = i;
    end = &ndigit[10];
    p = ndigit;
    while (p != end) {
        printf("ndigit is at \t %lx\n", (unsigned long) p);
        p++;
    }
}
Pointer to Pointer

- `char c;` declares a variable of type character
- `char* pc;` declares a variable of type pointer to character
- `char** ppc;` declares a variable of type pointer to pointer to character
- `c = 'a';` initialize a character variable
- `pc = &c;` get the address of a variable
- `ppc = &pc;` get the address of a variable

- `c == *pc == **ppc`
#include <stdio.h>

main() {

    int i, ndigit[10], **q, *p, *end;
    for (i = 0; i < 10; ++i)
        ndigit[i] = i;
    end = &ndigit[10];
p = ndigit;
    q = &p;
    while (*q != end) {
        printf("ndigit is at \t %\lX\n", (unsigned long) * p);
        (*q)++;
    }
}
**Aliases**

- In the previous example, \(*q\) and \(p\) are stored at the identical address
- Modifying \(*q\) will therefore also modify the value of \(p\)
- Two memory references that access the same memory location are called *aliases*
  - We can also say the two memory reference expressions are aliases
  - The simplest case is when two variable names are aliases
    - For example function parameter \(p\) and \(q\) may be pointers to the same location
- More examples of aliases
  - `int ndigit[10], *p;`  \(*p\) and `ndigit[0]` are aliases, *(p++) and `ndigit[1]` are aliases
• Aliases make it difficult to keep track of variable values
• Aliases make it difficult for the compiler to generate efficient machine code
• In the old days, it is more efficient to use pointers to step through an array than using array indices
  – This is not necessarily the case anymore
  – It is better to use array indexing if possible
    • Compilers are better at analyzing array indexing if there are no potential aliases
Quiz #1 (will be graded)

- Warm up your clickers...
• Consider the scope of variable i. What will this program print?

```c
int main()
{
    int i = 42;
    if (1) {
        int i = 0;
    }
    printf("%d", i);
}
```

• (a) 42
• (b) 0
• (c) architecture dependent
Quiz #1 - 2

- If there are no syntax errors, the command `gcc –c *.c` will produce

  - (a) One executable program, a.out
  - (b) One or more *.o files
  - (c) Beautified *.c files
In the C macro \#include “abc.h”, abc.h is

(a) A C standard library function
(b) An ordinary header file
(c) A string to be inserted in the program source code