Introduction

Chapter 1, part 2

Review of C

1.5 Review of C

C contains four basic data types:

`char` – **Primarily used to store ASCII symbols.**

(signed, 8-bit integer on your machine.)

`int` – **Primarily used to store integer values.**

(signed 32-bit integer on your machine.)

`float` – **Used to store real numbers.**

(32-bit IEEE 754-2008 format on your machine.)

`double` – **Used to store real numbers with twice the precision of** `floats`**.**

(64-bit IEEE 754-2008 format on your machine.)

And four modifiers: `short`, `long`, `signed`, `unsigned`

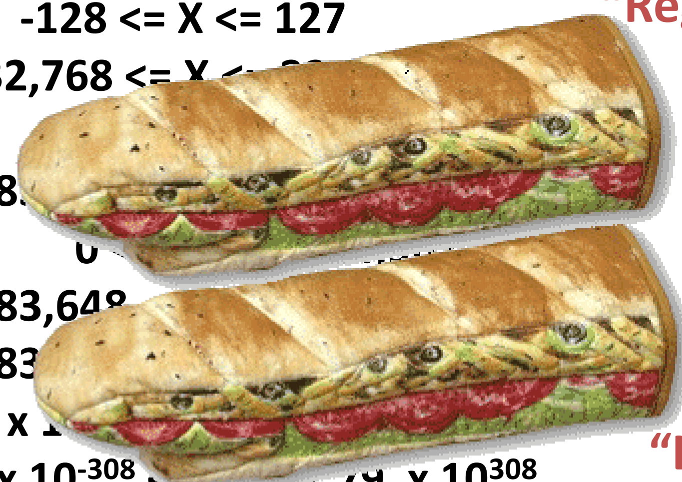
C Types

Borland C++ Data Types

"ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For C++Builder, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of internal representations for the various data types."

Type	Size (bits)	Range
`unsigned char`	8	0 <= X <= 255
`char`	8	-128 <= X <= 127
`short int`	16	-32,768 <= X [illegible]
`unsigned int`	32	[illegible]
`int`	32	-2,147,48[illegible]
`unsigned long`	32	0 [illegible]
`enum`	32	-2,147,483,648 [illegible]
`long`	32	-2,147,483[illegible]
`float`	32	1.18 x 1[illegible]
`double`	64	2.23 x 10^{-308} [illegible] 1.79 x 10^{308}
`long double`	80	3.37 x 10^{-4932} < |X| < 1.18 x 10^{4932}

"Regular"



"Large"

Types Example

types.c

```c
#include <stdio.h>

void main(void)
{   char c = 'A';
    int i = 65;
    float f = 54.123456789;
    double d = 54.123456789;

    printf("c= %c  i= %d  f= %12f  d= %12lf\n", c, i, f, d);
    printf("c= %d  i= %c  f= %12.9f  d= %12.9lf\n", c, i, f, d);

    printf("i= %f  i= %12.9lf\n", i, i);
    printf("f= %d  d= %d\n", f, d);
}
```

```
c= A  i= 65  f=    54.123455  d=    54.123457
c= 65  i= A  f= 54.123455048  d= 54.123456789
i= 0.000000  i= 54.123455048
f= 1610612736  d= 1078661069
```

C Arithmetic

C provides the following arithmetic operators:

+ - Addition

− - Subtraction

*** - Multiplication**

/ - Integer Division

% - Modulo Division (Remainder of Integer Division)

++ - Incrementing

−− - Decrementing

Notice that there is no exponentiation operator.

(As ^ in BASIC.)

Arithmetic Example

```
#include <stdio.h>

void main(void)
{ int i = 5, j = 6;
  int k[7];

  k[0] = i+j;
  k[1] = i-j;
  k[2] = i*j;
  k[3] = i/j;
  k[4] = i%j;
  k[5] = i++;
  k[6] = j--;

  for (i=0; i<7; i++){ printf("k[%d] = %d ", i, k[i]);}
}
```

arith.c

What happened to the increment?

```
k[0] = 11 k[1] = -1 k[2] = 30 k[3] = 0 k[4] = 5
k[5] = 5 k[6] = 6
```

Increment Example

```
#include <stdio.h>

int main(void)
{ int i, j = 0, k = 0;

  printf(" i j k\n");
  for (i=0; i<10; i++)
  { printf("%d %d %d\n", i, ++j, k++);
  }

}
```

postincr.c

```
 i j k
0 1 0
1 2 1
2 3 2
3 4 3
4 5 4
5 6 5
6 7 6
7 8 7
8 9 8
9 10 9
```

Increment Problems

What does this do?

`multi-incr.c`

```
i = 0; j = 1;
do
{ printf("%d %d %d\n", i, j++, ++j);
} while (++i<10);
```

Dev C++ on Windows

```
i j++ ++j
0 2 2
1 4 4
2 6 6
3 8 8
4 10 10
5 12 12
6 14 14
7 16 16
8 18 18
9 20 20
```

gcc on Linux

```
i j++ ++j
0 2 3
1 4 5
2 6 7
3 8 9
4 10 11
5 12 13
6 14 15
7 16 17
8 18 19
9 20 21
```

Assignment Operators

Equivalent:

```
i = i + 2
i += 2
```

Most binary operators have corresponding op=
Where op is one of : + − * / %

Equivalent:

```
expr-1 op= expr-2
expr-1 = (expr-1) op (expr-2)
```

What does this evaluate to?

x[a+b] *= y + 1

x[a+b] ~~=~~ x[a+b] * y + 1

x[a+b] = x[a+b] * (y + 1)

C Loops

C provides for three different loop structures:

```
while(expr)
      statement
```

While expression is non-zero. Used for variable amount of iterations but may never be entered.

```
for(expr1; expr2; expr3)
      statement
```

```
expr1;
while (expr2) {
      statement
      expr3;
}
```

Typically used when the number of iterations are known before entering the loop.

```
do
    statement
while (expr);
```

Used for variable amount of iterations but must be done at least once.

Loops Example

loops.c

```
#include <stdio.h>
void main(void)
{   int i, j=0;
    for (i=0; i<5; i++)
    {   j += i;
        printf("i = %d, j = %d\n", i, j);
    }
    printf("\n");
    while (i<10)
    {   j += i++;
        printf("i = %d, j = %d\n", i, j);
    }
    printf("\n");
    do
    {   j += i++;
        printf("i = %d, j = %d\n", i, j);
    } while (i <15);
}
```

Loops Example Output

```
i = 0, j = 0
i = 1, j = 1
i = 2, j = 3
i = 3, j = 6
i = 4, j = 10
```

```
i = 6, j = 15
i = 7, j = 21
i = 8, j = 28
i = 9, j = 36
i = 10, j = 45
```

```
i = 11, j = 55
i = 12, j = 66
i = 13, j = 78
i = 14, j = 91
i = 15, j = 105
```

Complex `for()` Loops

The for loop can contain multiple initialization and assignment statements by putting commas between each statement:

`for.c`

```
#include <stdio.h>

void main(void)
{   int i, j=0;

    for (i=0, j=0; i<5; i++, j+=5)
    {
       printf("i = %d, j = %d\n", i, j);
    }
}
```

```
i = 0, j = 0
i = 1, j = 5
i = 2, j = 10
i = 3, j = 15
i = 4, j = 20
```

C Conditionals

The `if-else` statement is the basic conditional in C.

`==`	**Tests for equality.**
`!=`	**Tests for inequality.**
`<`	**Tests for less than.**
`>`	**Tests for greater than.**
`<=`	**Tests for less than or equal to.**
`>=`	**Tests for greater than or equal to.**
`||`	**Tests for either operand being `TRUE`.**
`&&`	**Tests for both operands being `TRUE`.**

`TRUE` is equal to *not* `FALSE`, and `FALSE` is equal to the integer `0`.

C Conditionals

The constant values **TRUE** and **FALSE** themselves are not explicitly defined in C.

TRUE and **FALSE** may be defined in a header file as below.

```
#define FALSE 0
#define TRUE  !FALSE
/* or */
#define FALSE 0
#define TRUE  1
```

TRUE/FALSE Example

```c
#include <stdio.h>
int a = 0, b = 1, c = 2;
int main(void)
{   printf("!a = %d\n!b = %d\n!c = %d\n", !a, !b, !c);
    printf("(a == b) = %d\n", (a == b));
    printf("(a != b) = %d\n", (a != b));
    printf("(a < b) = %d\n", (a < b));
    printf("(a > b) = %d\n", (a > b));
    printf("(a || b) = %d\n", (a || b));
    printf("(a || !b) = %d\n", (a || !b));
    printf("(a && b) = %d\n", (a && b));
    printf("(b && c) = %d\n", (b && c));
}
```

```
!a = 1
!b = 0
!c = 0
(a == b) = 0
(a != b) = 1
(a < b) = 1
(a > b) = 0
(a || b) = 1
(a || !b) = 0
(a && b) = 0
(b && c) = 1
```

truefalse.c

Conditional Example

```
#include <stdio.h>

void main(void)
{   int i;

    for (i=0; i<10; i++)
    {   printf("%d: ", i);
        if (i > 4) printf("gt4 ");
        if (i != 2) printf("ne2 ");
        if (i <= 5) printf("le5 ");
        if (i % 2 == 0) printf("even ");
        if ((i > 4) && (i != 2)
            && (i <= 5) && (i%2 == 0))
        { printf("All\n");
          break;
        }
        else printf("\n");
    }
}
```

ifs.c

```
0: ne2 le5 even
1: ne2 le5
2: le5 even
3: ne2 le5
4: ne2 le5 even
5: gt4 ne2 le5
6: gt4 ne2 even
7: gt4 ne2
8: gt4 ne2 even
9: gt4 ne2
```

Precedence and Associativity of && and ||

- Precedence rules in C are complex
- C evaluates conditional tests in this order and from left to right

Order	Operator
1	`<` `<=` `>` `>=`
2	`==` `!=`
3	`&&`
4	`||`

```
if(a == b && c < d || e != 2)
```

What if a, b, c, d, e are function calls?

More examples in `ifs.c` and `truefalse.c`

- &&: evaluate left operand (including all side effects) and stop if false
- ||: same but stop if true

```
if ( ((a == b) && (c < d)) || (e != 2))
```

Comparing Floats for Equality is Dangerous

- This works if `x` is never changed:

```
double x = FLT_MAX;
if (x == FLT_MAX)   // true
```

- However, tests for equality can fail:

```
double y = sqrt(2.0);
x = x / y;
x = x * y;
if (x == FLT_MAX)
      // may not be true
```

- Machine dependent

floatex.c

Blocks

Individual statements may be grouped together by braces: `{ }`.

When placed after a conditional all or none will be executed. If no brace appears after a conditional, then the next single statement will be conditionally executed.

```
if (i == 0)
   a = 1;
b = 2;

if (i == 0) {
   a = 1;
   b = 2;
}
```

Beware:

```
if (i == 0);
   a = 1;
b = 2;
```

Complex C Structures

The `switch` statement is "multiple `if`" statement which can be used to chose from one of many actions depending on an integer value.

```
switch (Integer)
{ case 0: Do_Zero(); break;
  case 1:
  case 2: Do_OneAndTwo(); break;
  case 20: DoTwenty(); break;
  default: return();
}
```

Switch Example

switch.c

```
#include <stdio.h>
int main(void)
{   int i;
    for (i=0; i<10; i++)
    {
        switch (i)
        { case 0: printf("0"); break;
          case 1: printf("1");
          case 2: printf("2");
          case 3: printf("3"); break;
          case 9: printf("9"); break;
          default: printf("\n");
        }
    }
}
```

```
0123233



9
```

Ternary Operator

C also provides for a ternary conditional statement using `?` and `:` which forms a compact `if-else` statement.

```
j = ((k > 0) && (n < 0)) ? a : b+1;


printf("N is going %s\n",
            (dN < 0) ? strDOWN : strUP);
```

Ternary Example

compactif.c

```c
#include <stdio.h>

int main(void)
{   int i;
    for (i=0; i<10; i++)
    {   printf("%d: ", i);
        printf((i > 4) ? "gt4 ":"");
        printf((i != 2) ? "ne2 ":"");
        printf((i <= 5) ? "le5 ":"");
        printf((i % 2 == 0) ? "even ":"");
        if ((i>4)&&(i!=2)&&(i<=5)
                  &&(i%2==0))
        { printf("All\n");
          break;
        }
        else printf("\n");
    }
}
```

```
0: ne2 le5 even
1: ne2 le5
2: le5 even
3: ne2 le5
4: ne2 le5 even
5: gt4 ne2 le5
6: gt4 ne2 even
7: gt4 ne2
8: gt4 ne2 even
9: gt4 ne2
```

C Flow Control

The `continue` and `break` statements can be used "short-circuit" statements in a loop or block.

The `continue` statement redirects program flow to the beginning of a loop.

The `break` statement redirects program flow out of the loop altogether.

Flow Control Example

flow.c

```
#include <stdio.h>

void main(void)
{   int i;

    for (i=0; i<10; i++)
    {   if (i < 4) continue;
        if ((i > 6) && (i < 9)) break;
        printf("%d\n",i);
    }
}
```

```
4
5
6
```

goto

Though your programming professor might not have even mentioned it, C does have a `goto` instruction which is used along with a label.

It is never necessary, however, and once used changes your program from structured code to “spaghetti code.”

goto.c

```
for (i=0; i<ROWS; i++)
{ for (j=0; j<COLS; j++)
  { printf("Enter a[%d][%d]: ", i, j);
    if ((c = getchar()) == 'q') goto exit;
    else { printf("\n"); a[i][j] = c; }
  }
}
exit: printf("Done.\n");
```

Spaghetti Code

spaghetti.c

```
int main(void)
{
   goto d;
a: printf("A");
   goto c;
b: printf("B");
   return;
c: printf("C");
   goto e;
d: printf("D");
   goto a;
e: printf("E");
   goto b;
}
```

`goto` vs. structure

```c
void func()
{
  if (!doA())
    goto exit;
  if (!doB())
    goto cleanupA;
  if (!doC())
    goto cleanupB;
  // everything ok
  return;
cleanupB: UndoB();
cleanupA: UndoA();
exit:     return;
}
```

```c
void func()
{
  if (doA())
  { if (doB())
    { if (!doC())
      { UndoA();
        UndoB();
      }
    }
    else
    { UndoA();
    }
  }
return;
}
```

`switch` and `goto`

The reason some programmers don't like the `switch` statement is because it really contains `goto`s.

switch2.c

```
void dsend(int count)
{ int n;
  if (!count) return; else n = (count + 7) / 8;
  switch (count % 8)
  { case 0: do { puts("case 0");
    case 7:      puts("case 7");
    case 6:      puts("case 6");
    case 5:      puts("case 5");
    case 4:      puts("case 4");
    case 3:      puts("case 3");
    case 2:      puts("case 2");
    case 1:      puts("case 1");
               } while (--n > 0);
  }
}
```

Output and Input

The functions `printf()` it and `scanf()` can be used to write characters to the consoled and retrieve characters from the keyboard.

```
int printf ( const char * format, ... );

int scanf ( const char * format, ... );
```

Beware of `scanf()` and buffer overflow

```
int sscanf(char *s, const char * format, ... );
```

`sscanf(s, …)` is equivalent `scanf(…)` except that the input characters are taken from the string `s`.

`printf()` Format Tags

`%[flags][width][.precision][length]specifier`

specifier	Output	Example
`c`	Character	`Y`
`d` or `i`	Signed Decimal Integer	`365`
`e`	Scientific Notation	`3.6524e+2`
`E`	Scientific Notation	`3.6524E+2`
`f`	Decimal Floating Point	`365.24`
`g`	Use the Shorter of `%e` or `%f`	`365.24`
`G`	Use the Shorter of `%E` or `%F`	`365.24`
`o`	Signed Octal	`555`
`s`	String of Characters	`1 Year`
`u`	Unsigned Decimal Integer	`365`
`x`	Unsigned Hexadecimal Integer	`16d`
`X`	Unsigned Hexadecimal Integer with Capitals	`16D`
`p`	Pointer Address	`AC00:F004`
`n`	Nothing Printed. Argument is `int *` which stores number of bytes written	
`%`	The `%` is written	`%`

Format Tags Example
printf.c

```
int main(void)
{ const float x =
  int i;

  printf("Charact
  printf("String

  printf("Scienti
  printf("Float =

  printf("Either
  printf("Either

  printf("Octal =
  printf("Hex = %

  printf("Address
  printf("Nothing = %n:", &i);
  printf("i = %d\n", i);

  printf("%% = %d\n", '%');
}
```

```
Character = A A ╙
String = Go Tigers
Scientific = 3.652400e+002 or 3.652400E+002
Float = 365.239990
Either = 365.24 or 365.24
Either = 3e-007 or 3E-007
Octal = 555
Hex = 16d or 16D
Address of x = 0027FF44
Nothing = :i = 10
% = 37
```

`printf()` Format Flags

`%[flags][width][.precision][length]specifier`

`flags`	Affect
–	Left-justify in the field. (Right-justification is default.)
+	Forces a plus-sign to be printed.
space	If non-negative, forces a blank space to printed instead of **+**.
#	Specifies `0`, `0x`, or `0X` is written when printing octal or hex.
0	Pads left with `0`'s instead of sp

`flags`.

```
int main(void)
{ const int i = 365, j = -365;

  printf("-   :%10d\n     %-10d\n\n",
  printf("+   :%10d\n     %+10d\n\n",
  printf("0   :%10d\n     %010d\n\n",
  printf("' ': %d\n     % d\n\n", i,
  printf("#   :%#o     %#x     %#X\n\n
}
```

```
-   :       365
    365

+   :       365
           +365

0   :       365
    0000000365

' ': 365
    -365

#   :0555    0x16d    0X16D
```

`printf()` Format Width

`%[flags][width][.precision][length]specifier`

`width`	Affect
number	Minimum number of characters to be printed
`*`	The width is specified by an argument preceding the number argument.

`width.c`

```
int main(void)
{ const int i = 1492;

  printf("%d\n%3d\n%4d\n%5d\n%6d\n\

  printf("%d\n%*d\n%*d\n%*d\n%*d\n\
                        i, 3, i,

}
```

```
1492
1492
1492
 1492
  1492

1492
1492
  1492
     1492
        1492
```

`printf()` Format Precision

`%[flags][width][.precision][length]specifier`

`precision`	Affect
.number	For integer specifiers (**`d, i, o, u, x, X`**): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. A *precision* of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed **after** the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no *precision* is specified, the default is 1. If the period is specified without an explicit value for *precision*, 0 is assumed.
`.*`	The precision is specified by an argument preceding the number argument.

`printf()` Precision Example

precision.c

```
#include <stdio.h>

int main(void)
{ const int i = 1492;
  const float x = 3.1415926;
  const char str[] = "Clemson Tigers";

  printf(" .0: %.d\n", i);
  printf(" .0: %.0d\n", i);
  printf(" .0: %.0d\n", 0);
  printf(" .1: %.1d\n", i);
  printf("1.3: %1.3d\n", i);
  printf("3.3: %3.3d\n", i);
  printf("8.3: %8.3d\n\n", i);

  printf(" .0: %.0f\n", x);
  printf(" .0: %.0f\n", 0);
  printf(" .1: %.1f\n", x);
  printf("1.1: %1.1f\n", x);
  printf("1.3: %1.3f\n", x);
  printf("3.3: %3.3f\n", x);
  printf("8.3: %8.3f\n\n", x);
...
```

`printf()` Format Length

`%[flags][width][.precision][length]specifier`

`length`	Affect
h	An integer argument is treated as a short int.
l	An integer argument is treated as a long, and a character and string are treated as a wide character or wide character string.
L	A float is treated as a long double.

length.c

```
int main(void)
{ const int i = 1000000L;
  const long double x = 1.1E400L;

  printf("%% d: %d\n", i);
  printf("%%hd: %hd\n\n", i);

  printf("%% d: %d\n", i);
  printf("%%ld: %ld\n\n", i);

  printf("%% E: %E\n", x);
  printf("%%LE: %LE\n\n", x);
}
```

```
% d: 1000000
%hd: 16960

% d: 1000000
%ld: 1000000

% E: -9.586512E+232
%LE: 1.100000E+400
```

Escape Sequence Formats

Furthermore, it is handy to know the ASCII escape sequences for various characters when using `printf()`.

`Syntax`	Character
`\a`	Bell (alert)
`\b`	Backspace
`\f`	Formfeed
`\n`	New Line
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\'`	Single Quotation Mark
`\"`	Double Quotation Mark
`\\`	Backslash
`\?`	Question Mark
`\ooo`	ASCII character in octal notation
`\xhh`	ASCII character in hexadecimal notation

Escape Sequence Example

escape.c

```
int main(void)
{ printf("\\a - \a");
  printf("\\b - Go T
  printf("\\n - \n\n
  printf("\\f - Go\f

  printf("\\r - \nGo
  printf("\\t -
Where\tthe\tBlue\tRi
  printf("\\v -
Where\vthe\vBlue\vRi
  printf("\\\' - Pri
  printf("\\\" - Pri
  printf("\\\\ - Pri
  printf("\\ooo - Pr
  printf("\\xhh - Pr
}
```

```
\v - Where
          the
             Blue
                 Ridge
                      Yawns
                           Its
                              Greatness

\' - Print the character 'a'

\" - Print the string "String"

\\ - Print the directory C:\Windows\System

\ooo - Print ASCII 0101 A

\xhh - Print ASCII 0x41: A
```

`puts()` instead of `printf()`

The `printf()` function is complex as already shown and has a lot of overhead. If it is not needed, the much simpler `puts()` function can be used to write a simple (non-formatted) string the console.

It's important to note, however, that `puts()` automatically puts a carriage return at the end of the string so should not be used to print a partial line.

Inputting with `scanf()`

scanf.c

```
int main(void)
{ char c; int i;
  float f; char s[256];

  printf("Input a character: ");
  scanf("%c", &c); //fflush(stdin);
  printf("The character input was %c\n", c);

  printf("Input an integer: ");
  scanf("%d", &i); //fflush(stdin);
  printf("The integer input was %d\n", i);

  printf("\nInput a float: ");
  scanf("%f", &f); //fflush(stdin);
  printf("The float input was %f\n", f);

  printf("\nInput a string: ");
  scanf("%s", s); //fflush(stdin);
  printf("The string input was %s\n", s);

  fflush(stdin);
  getchar();
}
```

Inputting with `scanf()`

DevC++

```
Input a character: abcdefghijklmnopqrstuvwxyz
The chara
Input an

Input a :

Input a s
bcdefghi
```

linux

```
Input a character: abcdefghijklmnopqrstuvwxyz
The character input was a
Input an integer: The integer input was 1606419408

Input a float: The float input was 0.000000

Input a string: The string input was
bcdefghijklmnopqrstuvwxyz
```

```
Input a character: 1234 abcd 1234 abcd
The charac
Input an :

Input a fl

Input a st
```

```
Input a character: 1234 abcd 1234 abcd
The character input was 1
Input an integer: The integer input was 234

Input a float: The float input was 0.000000

Input a string: The string input was abcd
```

fflush(stdin)

- Works with Microsoft's compilers
- Does not work with gcc (and most other compilers?)
- The behavior of fflush(stdin) is undefined
- It is not acceptable to use fflush() for clearing keyboard input
- fflush(stdout) designed to be used on output
  – Later in course we will discover situations when it is required
  – Related to buffering on input and output streams

Another `scanf()` Example

scanf2.c

```
#include <stdio.h>

int main(void)
{ char d[4];
  int i;

  for (i=3; i>=0; i--)
  { printf("Input an intege
   scanf("%d", &d[i]);
   fflush(stdin);
  }

  puts("\n");
  for (i=0; i<4; i++)
  { printf("The integer input was %d\n", d[i]);
  }
}
```

```
Input an integer from 0 to 255: 1
Input an integer from 0 to 255: 2
Input an integer from 0 to 255: 3
Input an integer from 0 to 255: 4


The integer input was 4
The integer input was 0
The integer input was 0
The integer input was 0
```

Dangers of `scanf()`

Unfortunately, `scanf()` can produce problems with buffer overflow and end-of-line characters, so it should be avoided with gcc.

See: Why does everyone say not to use `scanf()`?

Use `fgets()` and `sscanf()`.

```
char line[MAXLINE], command[MAXLINE], restofline[MAXLINE];
int n, items;

int main(void)
{ while (fgets(line, MAXLINE, stdin) != NULL)
  { items = sscanf(line, "%s %d %s", command, &n, restofline);

    if (items == 1 && strcmp(command, "QUIT") == 0)
    { /* found exit */ break;
    }
    else if (items == 2) /* right number of commands */
    { printf("Two items: %s and %d\n", command, n);
    }
    else /* did not match any other test */
    {  printf("# %s", line);
    }
  } /* … */
```

fgets.c

Creating Random Numbers

One quality that computers don't have, and luckily so, is randomness. Therefore, it is difficult to produce randomness in a program.

Why would we need to produce randomness in a computer program?

To produce what we call "pseudo-random numbers", we can use the function `rand()` which returns a seemingly random integer uniformly distributed between `0` and `RAND_MAX`.

And to create integers in the range from `0` to `n-1`, we can simply use modulo division on the integers returned:.

```
#include <stdlib.h>

int main(void)
{ int i, n;

  for (i=0; i<10000; i++)
  { n = rand()%100;

  } /*…*/
```

random1.c

Creating Random Floats

To create random floating point numbers, we can simply divide the result by the range we desire (*making sure we use floating point division*).

What range of numbers does the following code produce?

```
int main(void)
{ int i;
  float f;

  for (i=0; i<100; i++)
  { f = (float)rand();
    f /= RAND_MAX;
    f *= 20;
    f -= 10;

    /*…*/
```

`random2.c`

"Seeding" the Generator

Though the sequence of numbers produced by `rand()` seems random, it is not. Furthermore, the *same* sequence is produced each time the function is used. That is, every time you restart the same program, the same sequence of numbers is generated.

When is this good? When is this bad?

The function `srand()` can be used to create a sequence which start with a different number. The seed parameter determines where the sequence starts.

What makes a good seed parameter?

```
int main(void)
{ int i;
  float f;

  for (i=0; i<100; i++)
  { f = (float)rand();
    f /= RAND_MAX;
    f *= 20;
    f -= 10;
```

`srand.c`

Other Random Functions

It has been shown that `rand()` does not have very good qualities for a pseudo-random number generator. Some better, and more complex, random functions include the following:

```
#include <stdlib.h>

double drand48(void); /* returns double between 0.0 and 1.0 */
void srand48(long int seedval); /* seeds the generator      */

long int lrand48(void); /* returns long between 0 and 2^31    */
long int mrand48(void); /* returns long between -2^31 and 2^31 */
```

`drand48.c`