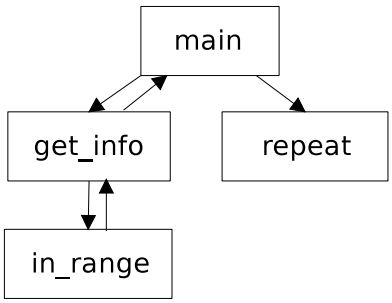
A call graph is a diagram that shows the functions in a program and uses lines to show which functions call which functions. Search the web for examples of call graphs; there are many examples and many styles. This document presents two programs and the corresponding call graphs.

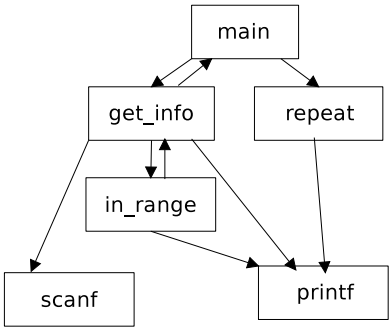
A Short Program to print "cookie":

```
int  get_info();
int  in_range(int);
void repeat(int);

int main()
{
    int n;

    n = get_info();
    repeat(n);
    return 0;
}

void repeat(int n)
{
    while( n-- > 0 )
        printf("cookie\n");
}

int get_info()
{
    int num;

    do {
        printf("Number (1..9): ");
        scanf("%d", &num);
    }
    while( ! in_range(num) );
    return num;
}

int in_range(int n)
{
    if ( n < 1 )
        printf("too few\n");
    else if ( n > 9 )
        printf("too many\n");
    else
        return 1;
    return 0;
}
```

Here is a call graph for this code:



Here is a more detailed graph:



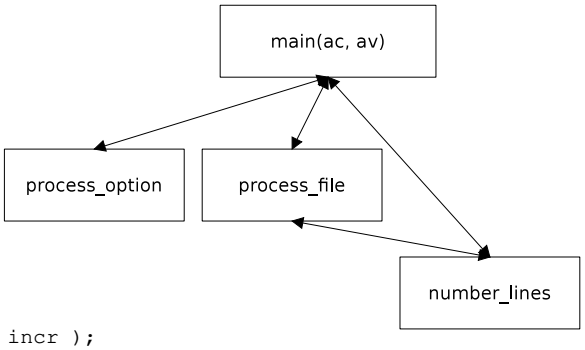
Making Call Graphs Making call graphs is a useful and common technique in software design. People use sticky notes and string, tools like any drawing program like dia or visio, websites like https://www.asciiflow.com/, https://www.asciiflow.com/legacy, miro, graph generators like graphviz.

Call graphs help you plan your code before you write it, help you document the code you have written, and help you make sense of code written by other people.

There are programs that read source code and generate call graphs. Some programs can analyze a running program to see which functions call which functions at run-time (called dynamic call graphs) and how how much time is spent executing each function.

Exercise: Make a call graph of your solution to a program you have written. Draw boxes around collections of functions to show which functions are in which files. This will show file organization in addition to logical organization.

Here is the code and a call graph for lnum3.c program that numbers lines:




```
#include <stdio.h>
#include <stdlib.h>
/*
 * lnum3.2 [-i# ] [filename..]
 * this version handles lines of any length by using getc/putchar not fgets
 */
void process_option(char *, int *);
int  process_file( char *, int, int );
int  number_lines( FILE *, int, int );

int main(int argcount, char *arglist[])
{
    int num = 1;                    // line number
    int incr = 1;                   // increment
    int pos;
    int named_files = 0;            // so far

    for( pos = 1; pos < argcount ; pos++ ){
        if ( arglist[pos][0] == '-' )
            process_option( arglist[pos], &incr );
        else {
            num = process_file( arglist[pos] , num, incr );
            named_files++;
        }
    }
    if ( named_files == 0 )
        number_lines( stdin, num, incr );
    return 0;
}
/* == process_option ==
 * process a command line option.  Currently supports -i# for increment
 * other options are reported as errors.
 *   -i47
 *   0123  number part starts at offset 2
 */
void process_option( char *opt, int *incr_p )
{
    switch ( opt[1] ){
        case 'i':    *incr_p = atoi( &opt[2] );
                     break;
        default:     fprintf(stderr, "lnum: bad option -%c0, opt[1]);
                     exit(1);
    }
}
/* == process_file ==
 * open a file, and if that works, then number the lines
 * rets: next line number
 */
int process_file( char *filename, int num, int incr )
{
    FILE *fp;                                       // stream ID
    if ( (fp = fopen(filename, "r") ) == NULL )
        fprintf(stderr, "lnum: cannot open %s0, filename);
    else {
        num = number_lines( fp, num , incr );
        fclose(fp);
    }
    return num;
}
/* == number_lines ==
 * prints from stream fp with line numbers starting at num
 * rets:  next value of num after printing
 */
int number_lines( FILE *fp, int n , int incr)
{
    int c;

    while ( ! feof(fp) ){
        printf("%6d", n);
        n += incr;
        while( ( c = getc(fp) ) != EOF && c != '0 )
            putchar(c);
        putchar('0);
   }
   return n;
}
```