```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MAX_LINE 80 /* 80 characters per line on input should be enough. */
#define MAX_COMMANDS 10 /* keep track of 10 most recent commands */

int num_cmds; /* global for number of most recent commands entered */
char history[MAX_COMMANDS][MAX_LINE]; /* global so it can be accessed in interrupt handler. */

/* This function will return 0 if an invalid history command has been input. 
It will just read in the next command line and separate it into distinct 
arguments, using blanks as delimiters and have the args array point to beginning of those arguments, and 
will convert them into a 
null terminated C-style strings. */

int setup(char inputBuffer[], char *args[], int *flag)
{
    int length, /* # of characters in the command line */
    i, /* loop index for accessing inputBuffer array */
    start, /* index where beginning of next command parameter is */
    temp, /* return value from atoi() */
    ct; /* index of where to place the next parameter into args[] */

    ct = 0;
    length = read(0,inputBuffer,MAX_LINE);

    /* 0 is the system predefined file descriptor for stdin (standard input), 
which is the user's screen in this case. inputBuffer by itself is the 
same as &inputBuffer[0], i.e. the starting address of where to store 
the command that is read, and length holds the number of characters 
read in. inputBuffer is not a null terminated C-string. */

    start = -1;
    if (length == 0) /* ^d was entered, end of commands */
        exit(0);

    if ( (length < 0) && (errno != EINTR) ) {
        perror("error reading the command");
        exit(-1); /* terminate with error code of -1 */
    }

    inputBuffer[length] = '\0'; /* turn it into a null terminated C style string */
    if (inputBuffer[0] == '!' ) {
        if (inputBuffer[1] == '!' ) {
            if (num_cmds == 0) {
                printf(" No commands in the history - !! not allowed.\n");
            return 0;
            } else {
                strcpy(inputBuffer, history[(num_cmds-1)%MAX_COMMANDS]);
            } else {
                temp = atoi(&inputBuffer[1]);
        }
    } else { /* continue with the next argument */
```
if (temp < 1 ||
   temp > num_cmds || temp <= num_cmds - MAX_COMMANDS){
   printf(" command %d is not in the correct range!\n",temp);
   return 0;
}
strcpy(inputBuffer, history[(temp-1)%MAX_COMMANDS]);
}
length = strlen(inputBuffer);
}
num_cmds++;
strcpy(history[(num_cmds-1)%MAX_COMMANDS],inputBuffer);

for (i = 0; i < length; i++) {
   if (inputBuffer[i] == ' &') {
      inputBuffer[i] = '\0';
      *flag = 1;
      --length;
      break;
   }
}

for (i=0;i<length;i++){ /* examine every character in the inputBuffer */
   switch (inputBuffer[i]){
      case ' ': /* argument separators */
         if (start != -1){
            args[ct] = &inputBuffer[start]; /* set up pointer */
            ct++;
         }
         inputBuffer[i] = '\0'; /* put in a null to end the argument */
         start = -1;
         break;
      case '\n': /* same comment as for ' ',"t" */
         if (start != -1){
            args[ct] = &inputBuffer[start];
            ct++;
         }
         inputBuffer[i] = '\0';
         start = -1;
         args[ct] = NULL; /* no more arguments to this command */
         break;
      default : /* some other character */
         if (start == -1)
            start = i;
   } /* end of switch */
   } /* end of for */
   args[ct] = NULL;
}
} /* end of setup routine */

**
* This function handles when we receive a <control><C>
*/
void sigint_handler()
{
   int i;

   if (num_cmds < 1)
printf("Sorry no commands to list!!!\n");

else {
    printf("\n\n");
    for (i = (num_cmds >= MAX_COMMANDS) ? num_cmds - MAX_COMMANDS:0; i < num_cmds; i++)
        printf("command #%d is %s\n",i+1,history[i%MAX_COMMANDS]);
}

printf(" COMMAND --\n");
}

int main(void)
{
    int flag;               /* equals 1 if a command is followed by '&' */
    char *args[MAX_LINE/2]; /* command line (of 80) must have < 40 arguments */
    int child,             /* process id of the child process */
    valid,                /* return value from setup */
    status;               /* result from execvp system call*/
    char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
    num_cmds = 0;

    /** set up the signal handler */
    struct sigaction handler;
    handler.sa_handler = sigint_handler;
    handler.sa_flags = 0;
    sigemptyset(&handler.sa_mask);
    sigaction(SIGINT, &handler, NULL);

    while (1){ /* Program terminates normally inside setup */
        flag = 0;
        printf("COMMAND->\n");
        fflush(stdout);
        valid = setup(inputBuffer,args,&flag); /* get next command */
        if (valid == 0) continue; /* error message will be generated inside setup for invalid history command */
        child = fork(); /* creates a duplicate process */
        switch (child) {
        case -1: /* error - could not fork child process */
            perror("could not fork the process");

            /* perror is a library routine that displays a system error message, according to the value of the system variable "errno" which will be set during a function (like fork) that was unable to successfully complete its task. */

            break;

        case 0: /* here is the child process */
            status = execvp(args[0],args);
            if (status != 0){
                perror(" error in execvp");
                exit(-2); /* terminate this process with error code -2 */
            }
            break;

        default :
            if(flag==0) /* handle parent, wait for child */
                while (child != wait((int *) 0)) ;

            break;
        }
    }