main Function

A C program starts executing with a function called *main*.

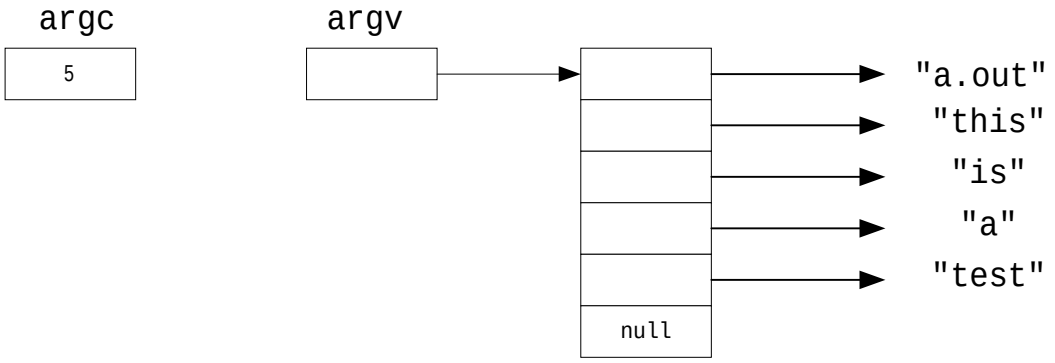
```
int main(int argc, char *argv[]);
```

argc is the number of command-line arguments.
argv is the array of pointers to the arguments.

main returns the status of the program.

- 0 - denotes success
- Non-zero - denotes an error.

```
% a.out this is a test
```




```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;

   for(i = 0; i < argc; i++) {
     printf("argv[%d]: %s\n", i, argv[i]);
   }
   return(0);
}
```

```
% ./a.out a b c
argv[0]: ./a.out
argv[1]: a
argv[2]: b
argv[3]: c
```

errno Variable

errno is set by system calls (and some library functions) to indicate what went wrong. Its value is significant only when the call returned an error (usually -1), and a library function that does succeed is allowed to change errno.

Summary of **errno** variable

Include File(s)	<errno.h>	Manual Section	3
Summary	extern int errno;		

Note that errno is undefined after a successful library call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to perror, the value of errno should be saved

perror Function

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function.

Summary of **perror** call

Include File(s)	<stdio.h>	Manual Section	3
Summary	void perror (const char *s);		
Return	Success	Failure	Sets errno

s is a pointer to a character string constant. The argument string s is printed first, then a colon and a blank, then the message and a new-line.

The function perror() serves to translate this error code into human-readable form.

exit Function

exit terminates a program normally by returning to the kernel.

Summary of **exit** call

Include File(s)	include <stdio.h>	Manual Section	3
Summary	void exit(int *status*);		
Return	Success	Failure	Sets errno
			No

status is status to returned to the kernel.

Actually, a special startup routine is called which in turn calls main.

```
exit(main(argc, argv));
```

main Function revisited

Alternative declarations for main.

```
int main(void);

void main(void);

void main(int argc, char *argv[]);
```

The second and third forms of main require an explicit call to exit.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
   int i;

   for(i = 0; i < argc; i++) {
     printf("argv[%d]: %s\n", i, argv[i]);
   }

   exit(0);
}
```

printf Function

printf accepts a series of arguments, applies a format specifier contained in the format string given by *format*, and outputs the formatted string to *stdout*.

Summary of **printf** call

Include File(s)	include <stdio.h>	Manual Section	3
Summary	int printf(const char *format [,argument,…]);		
Return	Success	Failure	Sets errno
	the number of bytes output	EOF	No

format is the format string used to format the values of *argument*.
arguments a series of arguments.

The format string is a character string that contains two types of objects – *plain characters* and *conversion specifications*.

- Plain characters are simply copied verbatim to the output stream.
- Conversion specifications fetch arguments from the argument list and apply formatting to them.

Each conversion specification begins with a %.

Type character	Input argument	Format of output
d	integer	signed decimal int
c	character	Single character
s	string pointer	Prints characters until a null-terminator is reached.
p	pointer	Prints the input argument as a pointer.

```
printf("Name: %s, Age: %d\n", "Harry", 36);
  ➔ Name: Harry, Age: 36
printf("Name: %s, Age: %d\n", "Harry");
  ➔ Crash?
printf("Name: %s, Age: %d\n", "Harry", 36, "Fred", 45);
  ➔ Name: Harry, Age: 36
```

`getopt` Function

`getopt` parses an argument vector according to a string of option specifiers one argument at a time.

Summary of **getopt** call

Include File(s)	`include <unistd.h>`	Manual Section	3

Summary	`int getopt( int argc, char **argv, char *optstring);`
	`extern char *optarg;`
	`extern int optind, opterr;`		

Return	Success	Failure	Sets `errno`
	Next option letter	-1 or '?'	No

`argc` is the number of arguments in *`argv`*.
`argv` is the array of pointers to the arguments.
Typically, `argc` and `argv` are the arguments passed to `main`.

`optstring` is a pointer to a string of valid option letters (characters) that `getopt` will recognize.

Return value	Meaning
-1	All options have been processed, or the first non-option argument has been reached.
?	An option letter has been processed that was not in the `optstring` or an option argument was specified but none was found. If `opterr` is set to one, an error is displayed on standard error.
Next option letter	The next option letter in `argv` that matches a letter in `optstring`. If the letter matched in `optstring` is followed by a colon, then `optarg` will reference the argument value.

`optind` contains the index of the next argument in `argv` to be processed. Initially set to `1`.

`getopt` processes one argument at a time.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char *optarg;
extern int  optind, opterr;

void main(int argc, char *argv[])
{
  int    c;
  static char optstring[] = "abs:";

  opterr = 0;                       /* turn off automatic
error msgs */
  while ((c = getopt(argc, argv, optstring)) != -1) {
    switch(c) {
    case 'a':
      printf("Found option a\n");
      break;
    case 'b':
      printf("Found option b\n");
      break;
    case 's':
      printf("Found option s with an argument of: %d\n",
atoi(optarg));
      break;
    case '?':
      printf("Found an option that was not in
optstring\n");
    }
  }
  if(optind < argc) {
    printf("Left off at: %s\n", argv[optind]);
  }
}
```

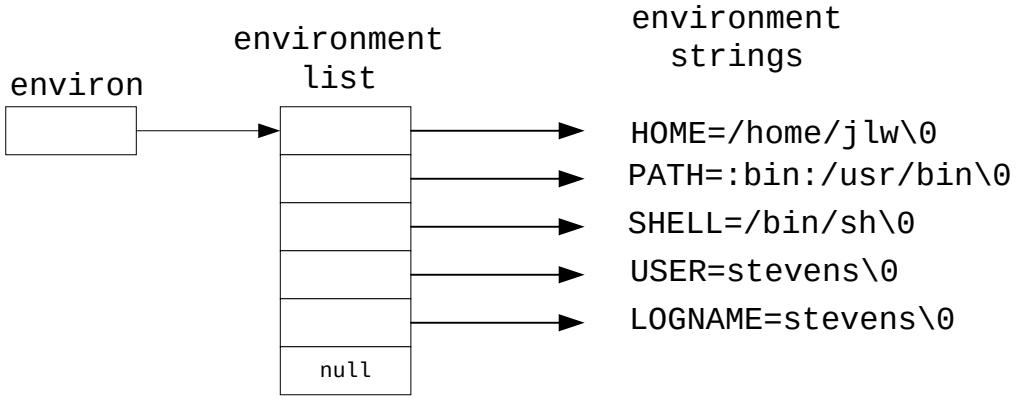
```
 % ./a.out -abc -s 34 -b joe -a
Found option a
Found option b
Found an option that was not in optstring
Found option s with an argument of: 34
Found option b
Found option a
Left off at: joe
```

environ Variable

environ points to an array of strings called the "environment" which is made available when a process begins. By convention these strings have the form "name=value".

Summary of **environ** variable

Include File(s)	<unistd.h>	Manual Section	5
Summary	extern char **environ;		



Common examples are:

NAME	VALUE
USER	The name of the logged-in user (used by some BSD-derived programs).
LOGNAME	The name of the logged-in user (used by some System-V derived programs).
HOME	A user's login directory, set by login(1) from the password file passwd(5).
LANG	The name of a locale to use for locale categories when not overridden by LC_ALL or more specific environment variables.
PATH	The sequence of directory prefixes that sh(1) and many other programs apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'. (Similarly one has CDPATH used by some shells to find the target of a change directory command, MANPATH used by man(1) to find manual pages, etc.)
PWD	The current working directory. Set by some shells.
SHELL	The file name of the user's login shell.
TERM	The terminal type for which output is to be prepared.

```
#include <stdio.h>

extern char **environ;

void main(void) {
  char **ptrs;

  for(ptrs = environ; *ptrs; ++ptrs) {
    printf("%s\n", *ptrs);
  }
}
```

```
% ./a.out
TERM=vt100
HOME=/home/css/jlw
…
```

getenv Function

getenv returns a pointer to the value of a name=value string.

Summary of **getenv** call

Include File(s)	<stdlib.h>	Manual Section	3
Summary	char *getenv(char *name);		
Return	Success	Failure	Sets errno
	Pointer to the value in the environment	NULL	

name of the variable whose value you are trying to retrieve.

Display the contents of the TERM variable:

```
#include <stdio.h>
#include <stdlib.h>

void main(void) {
  char *c_ptr, *msg;

  c_ptr = getenv("TERM");
  msg = c_ptr == NULL ? "Not Found" : c_ptr;
  printf("The variable TERM is %s\n", msg);
}
```

```
% echo $TERM
vt100
% ./a.out
The variable TERM is vt100
```

putenv Function

putenv takes a string of the form "name=value" and places it in the environment list. If the name already exists, its old definition is first removed.

Summary of **putenv** call

Include File(s)	include <stdlib.h>	Manual Section	3
Summary	int putenv(char *string);		
Return	Success	Failure	Sets errno
	0	Non-negative integer	

string is of the form "name=value".

Add and then display the contents of the MYVAR variable.

```
#include <stdio.h>
#include <stdlib.h>

void main(void) {
  char *c_ptr;
  char *msg;

  if(putenv("MYVAR=myvalue") < 0) {
    perror("Error assigning MYVAR");
  }
  c_ptr = getenv("MYVAR");
  msg = c_ptr == NULL ? "Not Found" : c_ptr;
  printf("The variable MYVAR is %s\n", msg);
}
```

Changing the environment only affects the current environment! (*FORK?)*

```
% ./a.out
The variable TERM is myvalue
% echo $MYVAR
MYVAR: Undefined variable.
```