C Basics 8: Errors and Error Handling

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. **Errors and Error Handling**
    - **error returns**
    - **error checking code structure**
    - **stdio errors**
    - **errno**
    - **error messages**

C Basics 8: Errors and Error Handling

Error Returns

C does not have an **exception mechanism**, so errors are indicated by the **return value** of a function, typically one of:

- `NULL` – if the return is a pointer/string;
- -1 – if the return is an integer;
- `EOF` – for `char`/`int` return values in stdio;

Calls to functions will often need to be *wrapped* with an `if` that tests whether an error occurred and takes appropriate action:

```
if ((fptr = fopen(file,"r")) == NULL) {
  fprintf(stderr,"Error opening file %s\n",file);
  exit(EXIT_FAILURE);
}
```

Error Checking Code Structure

Because so many calls must be checked, we generally *test for the error return* and then print a message plus exit/return:

```
//Open file:
if ((fptr = fopen(file,"r")) == NULL) {
  fprintf(stderr,"Error opening file %s\n",file);
  exit(EXIT_FAILURE); }

//Read char from file:
if ((next = fgetc(fptr)) == EOF && ferror(fptr)) {
  fprintf(stderr,"Error reading from file %s\n",file);
  exit(EXIT_FAILURE); }

//Print char to stdout:
if (putchar(next)) == EOF) {
  fprintf(stderr,"Error writing to stdout\n");
  exit(EXIT_FAILURE); }

//Done, successfully:
exit(EXIT_SUCCESS);
```

Error Checking Code Structure (contd.)

Notice how *testing for the non-error return* instead results in nested if-then-else's that are very hard to read:

```
if ((fptr = fopen(file,"r")) != NULL)

  if ((next = fgetc(fptr)) != EOF)

    if (putchar(next)) != EOF)
      exit(EXIT_SUCCESS);

    else {
      fprintf(stderr,"Error writing to stdout\n");
      exit(EXIT_FAILURE); }

  else {
    fprintf(stderr,"Error reading from file %s\n",file);
    exit(EXIT_FAILURE); }

else {
  fprintf(stderr,"Error opening file %s\n",file);
  exit(EXIT_FAILURE); }
```

stdio Functions and Errors

Functions in stdio may use the same return value for both an error and an *end-of-file* condition (not an error).

E.g., `int fgetc(FILE *stream)`
Return: "the character read as an unsigned char cast to an int or EOF on end of file or error."

Two stdio functions (predicates) can be used to distinguish between end-of-file and error:

- `feof(FILE *stream)` – has `stream` encountered end-of-file?
- `ferror(FILE *stream` – has `stream` encountered an error?

errno

`errno` is **global variable** containing an **error code**:

- it is *zero* when no errors have occurred, or
- it is a *positive integer* identifying the most recent error

When an error occurs during a **system call** (and many library functions make system calls to do their work), the kernel sets `errno`.

The header file **errno.h** defines `errno` and symbolic constants for the set of possible error codes:
e.g., `EACCESS` is the “permission denied” error.

See the man page for `errno.h` for a list of possible errors and their symbolic names.

errno (contd.)

Man pages for system calls and many library functions have an “ERRORS” section that lists the errors that could occur with the call.

In code involving system calls, `errno` can be tested to check for an error.

The OS never zeros `errno` out after an error, so if execution is to continue after an error, it is often necessary to reset `errno` to zero (through assignment).

Error Messages

When printing **error messages**, it is best to provide as much information as possible.

This helps a user understand what occurred and how to fix it.

For example, suppose a call to `fopen()` fails.

Consider a message like:
“Error opening file”

This hardly helps the user; was the wrong file being opened, were there permissions problems, did the file not exist, etc.?

A much better message would be:
“Error opening file test.text: permission denied”

This makes it clear what file could not be opened, and why.

Error Messages (contd.)

C provides two library functions that will print the **system error message strings** that describe an error code:

```
  void perror(const char *s)
  char *strerror(int errnum)
```

`perror()` takes a string that becomes the prefix to the message:

```
  if ((fptr = fopen(file,"r")) == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE); }
```

Message would be like:
“`Error opening file: permission denied`”

This is generally considered the most basic sort of reasonable error message.

Error Messages (contd.)

`strerror()` provides the system error message string so that it can be used to construct more informative error messages:

```
  if ((fptr = fopen(file,"r")) == NULL) {
    fprintf(stderr,"%s: Error opening file %s: %s\n",
            argv[0],file,strerror(errno));
    exit(EXIT_FAILURE); }
```

Message would be like:
“`prog: Error opening file test.txt: permission denied`”

This is a better message because it identifies the program having the issue, and also identifies the name of the file that could not be opened.

Error Messages (contd.)

In addition to these C standard error reporting functions, GCC/Glib provide two additional, non-standard, error reporting functions:

```
void error(int status, int errnum, const char *format, ...)
void error_at_line(int status, int errnum, const char *filename,
                   unsigned int linenum, const char *format, ...)
```

`error()` makes it a bit easier to accomplish what we showed with the `strerror()` example:

```
if ((fptr = fopen(file,"r")) == NULL)
  error(EXIT_FAILURE,errno,"Error opening file %s",file);
```

`error()` automatically prepends the program name (followed by colon, space) and appends a colon, space, and system error message.

In addition, if its `status` argument is non-zero, it then calls `exit(status)` to terminate the program.

Error Messages (contd.)

`error_at_line()` adds the parameters `filename` and `linenum`.

Its output differs in that after the program name, a colon, the value of `filename`, a colon, and the value of `linenum` get inserted in the output.

The preprocessor values `__FILE__` and `__LINE__` are typically used as the `filename` and `linenum` arguments.