CS240: Programming in C

Lecture 14: Errors

Errors

- We’ve already seen a number of instances where unexpected (and uncaught) errors can take place:
  - Memory buffer overflow/underflow
  - unintended casts
  - misuse of sizeof
  - structure field alignment
  - pointer arithmetic
  - dangling references

Example 1

```
int x = 2;
switch (x) {
  case 2: printf(“Two\n”);
  case 3: printf(“Three\n”);
}
```

This prints

```
Two
Three
```

Why?

Example 2

```
int x = 5;
if (x = 6)
   printf("x equals 6"\n);
```

This prints

```
x equals 6
```

Why?

Example 2

```
int x = 5;
if (x = 6)
   printf("x equals 6"\n);
```

This prints

```
x equals 6
```

Why?

The assignment x = 6 in the conditional test sets x to 6, and returns a non-zero value which executes the branch

Example 3

- Not supplying address of variable in scanf where input should be stored.
- Array addressing starts at 0
- Overloaded division
  - double half = 1/2;
    - sets half to 0 not 0.5 since 1 and 2 are integer constants.
    - need to cast to get correct behavior

```
int x = 5, y = 2;
double d = ((double) x) / y;
```

Common Semantic Errors

```
 int x = 5;
while (x > 0);
  x--
```

This is an infinite loop.
Why?

Common Semantic Errors

```
int x = 5;
while (x > 0);
  x--
```

This is an infinite loop. Why?

The semicolon after the while defines the statement to repeat -- it's simply the null statement.

Common Semantic Errors

```
#include <string.h>
int main ()
  {
     char * st;
     strcpy(st, “abc”);
     return 0;
}
```

Generates an error when executed.
Why?

Common Semantic Errors

```
#include <string.h>
int main ()
   {
       char * st;
       strcpy(st, "abc");
       return 0;
}
```

Space for `st` has not been allocated.

Change declaration to char `st[20]`

Generates an error when executed. Why?

Common Semantic Errors

```
char st1[] = “abc”;
char st2[] = “abc”;
if (st1 == st2)
   printf(“Yes”);
else
   printf(“No”)
```

---

char str[30] only has room for 29 (not 30) data characters since the last character must be null

```
char * copy_str = malloc(strlen(orig_str) + 1);
strcpy(copy_str, orig_str)
```

Common Semantic Errors

```
int count_line_size (FILE * fp) {
   char ch;
   int count = 0;
   while ( (ch = fgetc(fp)) != EOF && ch != ‘\n’
      count++;
   return count;
}
```

This gets translated to:

```
while ( (int) (ch = (char) fgetc(fp)) !=EOF && ch != ‘\n’
```

What’s the problem and solution?

Common Errors

```
int x;
char st[31];

printf("Enter an integer: ");
scanf("%d", &x);
printf("Enter a line of text: ");
fgets(st, 31, stdin)
```

What will fgets read? Why?

Solution

```
void dump_line( FILE * fp )
{
  int ch;

  while( (ch = fgetc(fp)) != EOF && ch != '\n' )
    /* null body */;
}

int x;
char st[31];

printf("Enter an integer: ");
scanf("%d", &x);
dump_line(stdin);
printf("Enter a line of text: ");
fgets(st, 31, stdin);
```

Detecting and Reporting Errors

```
FILE * f;
f = fopen(“myfile”, "r");
if(f == NULL) {
    exit(fprintf(stderr, “Open file failed\n”));
}
```

Detecting and Reporting Errors

```
FILE * f;
f = fopen(“myfile”, "r");
if(f == NULL) {
    exit(fprintf(stderr, “Open file failed\n”));
}
```

- The C library detects errors and provides support for reporting error conditions and messages.

Errno variable

- The `errno.h` header file defines the error conditions detected by the C Library
- Library functions designed to report informative error messages in the customary format about the failure of a library call:

```
char * strerror(int errno);
void perror(const char *message);
```

errno

- errno contains the system error number. The programmer can change the value of errno.
- Initial value of errno at program startup is zero.
- Any library functions might modify errno, by setting it to nonzero values when they encounter errors (listed for each function).
- Functions do not change errno when they succeed; thus, the value of errno after a successful call is not necessarily zero.
- Do not use errno to determine whether a function failed. Each function documents how you can check that it failed. If the function failed, you can examine errno.

Error Codes

- All the error codes have symbolic names (reserved names) and have distinct positive values; They are defined in `errno.h‘.
- Examples:
  - EPERM : Operation not permitted; only the owner of the file (or other resource) or processes with special privileges can perform the operationr.
  - EINVAL: Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.

Common error values

```
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO          5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
```

130 error numbers in Linux

Error Messages - perror

```
include <stdio.h>
void perror(const char *message);
```

- Prints the string message and the error message corresponding to the value or errno to the stream stderr.
- If message is either a null pointer or an empty string, perror just prints the error message corresponding to errno, adding a trailing newline.
- Otherwise perror prefixes its output with the string message.

Example

```
int fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) {
    perror("open");
    exit(1);
}
```

```
open: Permission denied
```

Error Messages - strerror

```
include <string.h>
char * strerror (int errnum);
```

- Maps the error code specified by the errnum argument to a descriptive error message string and returns a pointer to this string.
- The value errnum normally comes from the variable errno.
- You should not modify the string returned by strerror.
- Subsequent calls to strerror, might overwrite the error string.

Error Messages (contd.)

- Many programs that do not read input from the terminal are designed to exit if any system call fails. By convention, the error message from such a program should start with the program's name.
- How do we obtain the program’s name?

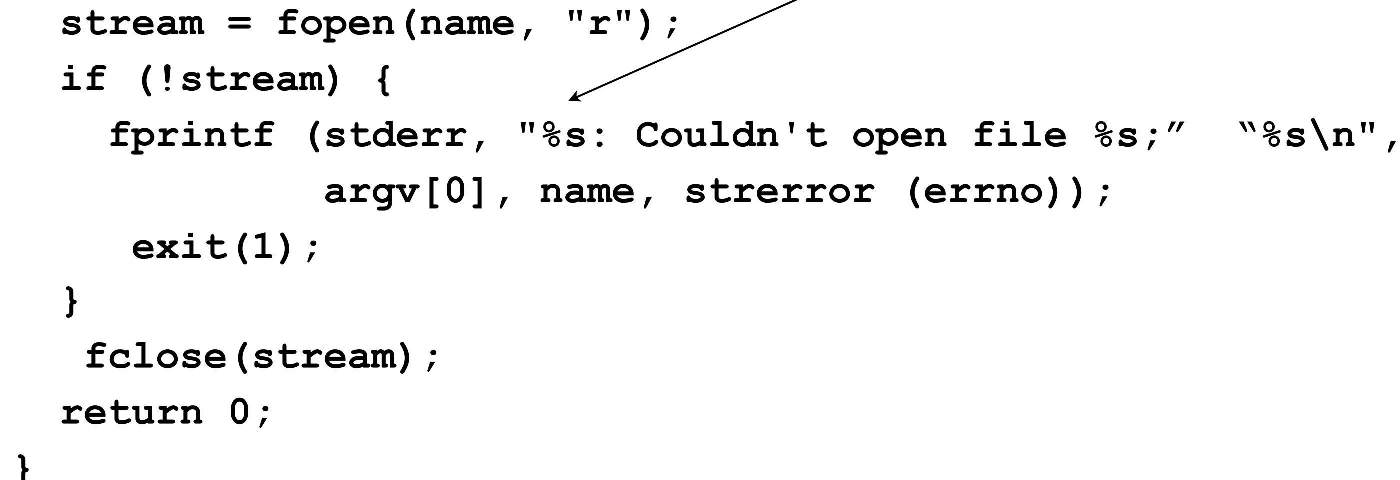
Error Message - Example

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


int main(int argc, char *argv[]) {
  FILE *stream;
  char name[]=“myfile”;


  stream = fopen(name, "r");
  if (!stream) {
    fprintf (stderr, "%s: Couldn't open file %s;”  “%s\n",
             argv[0], name, strerror (errno));
     exit(1);
  }
   fclose(stream);
  return 0;
}
```

argv[0] is the program name



Error messages with varying-length argument lists

```
#include <stdarg.h>
void va_start(va_list ap, arg_name);
```

- initializes processing of a varying-length argument list.
- **`arg_name`**, is the name of the parameter to the calling function after which the varying part of the parameter list begins (the parameter immediately before the ,...). The results of the **`va_start`** macro are unpredictable if the argument values are not appropriate.

```
void va_end(va_list ap);
```

- ends varying-length argument list processing

Varying-length Argument List

```
#include <stdarg.h>
(arg_type) va_arg(va_list ap, arg_type);
```

- returns the value of the next argument in a varying-length argument list.
- **`va_list`** must be initialized by a previous use of the **`va_start`** macro, and a corresponding **`va_end`** should be called after finishing processing the arguments.
- **`arg_type`** is the type of the argument that is expected.
- the returned results are unpredictable if the argument values are not appropriate.
- no way to test whether a particular argument is the last one in the list. Attempting to access arguments after the last one in the list produces unpredictable results.

Simplified version of cp

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

#define PERMS 0666  /* RW for owner, group and others */
void error(char *, ...);
int main(int argc, char *argv[]) {
  int f1, f2, n;
  char buf[BUFSIZ];
  if(argc != 3) error("Usage: cp from to");
  if((f1 = open(argv[1], O_RDONLY, 0)) == -1) error("my_cp: can't open %s", argv[1]);
  if((f2 = creat(argv[2], PERMS)) == -1) error("my_cp: can't create %s, mode%3o", argv[2], PERMS);
  while((n = read(f1, buf, BUFSIZ)) > 0)
    if(write(f2, buf, n) != n) error("cp: write error on file %s", argv[2]);;
  return 0;
}
```