

CSE 333 – Section 4: POSIX, C++ Intro

Welcome back to section! We're glad that you're here :)

POSIX and Files

POSIX has similar file I/O operations as the C stdio library, but unbuffered by default, including:

```
int open(char *name, int flags, mode_t mode);
```

- *name* is a string representing the name of the file. Can be relative or absolute.
- *flags* is an integer code describing the access. Some common flags are listed below:
 - ◆ `O_RDONLY` – Open the file in read-only mode.
 - ◆ `O_WRONLY` – Open the file in write-only mode.
 - ◆ `O_RDWR` – Open the file in read-write mode.
 - ◆ `O_APPEND` – Append new information to the end of the file.
- ★ Returns an integer which is the file descriptor. Returns `-1` if there is a failure.

```
int close(int fd);
```

- *fd* is the file descriptor (as returned by `open()`).
- ★ Returns `0` on success, `-1` on failure.

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

- *fd* is the file descriptor (as returned by `open()`).
- *buf* is the address of a memory area into which the data is read or written.
- *count* is the maximum amount of data to read from or write to the stream.
- ★ Returns the *actual* amount of data read from or written to the file.

POSIX and Errors

Unfortunately, errors are not handled as nicely for the user as they are in the C stdio library. So it is important to make sure your code handles errors gracefully. Note that:

- When an error occurs, the error number is stored in `errno` (defined in `<errno.h>`).
- You can use `perror()` to print out a message based on `errno`.
- Remember that `errno` is shared by all library functions and overwritten frequently, so you must read it *right* after an error to be sure of getting the right code.

POSIX functions have a variety of error codes to represent different errors. Some common error conditions:

- ◆ `EBADF` – *fd* is not a valid file descriptor or is not open for reading.
- ◆ `EFAULT` – *buf* is outside your accessible address space.
- ◆ `EINTR` – The call was interrupted by a signal before any data was read.
- ◆ `EAGAIN` – *fd* refers to a file other than a socket and has been marked nonblocking, and the read/write blocks.
- ◆ `EISDIR` – *fd* refers to a directory.

`EAGAIN` and `EINTR` are recoverable errors, unlike the rest.

POSIX and directories

POSIX calls can also be used to access directories. This is because in Linux, directories are nothing more than special files. An example workflow might be: open a directory, iterate through directory contents, close the directory.

```
DIR *opendir(const char* name);
```

→ *name is the directory to open. Accepts relative and absolute paths. Can end with '/', but is not necessary.*

★ Returns a pointer `DIR*` to the directory stream or `NULL` on error (with `errno` set).

```
int closedir(DIR *dirp);
```

→ *dirp is the directory stream to close.*

★ Returns 0 on success or -1 on error (with `errno` set).

```
struct dirent *readdir(DIR *dirp);
```

→ *dirp is the directory stream to process.*

★ Returns a pointer to a `dirent` structure representing the next directory entry in the directory stream or returns `NULL` on error or reaching the end of the directory stream.

On Linux, the `dirent` structure is defined as follows:

```
struct dirent {
    ino_t      d_ino;      /* inode number for the dir entry */
    off_t      d_off;      /* not necessarily an offset */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file (not what you think);
                             not supported by all file system
                             types */

    char       d_name[NAME_MAX+1]; /* directory entry name*/
};
```

Exercises:

- 1) Why might a POSIX standard be beneficial? From an application perspective? Versus using the C stdio library?
- 2) A common use of the POSIX I/O function is to **write** to a file; fill in the code skeleton below that writes all of the contents of a string `buf` to the file `333.txt`.

```
int fd = _____; // open 333.txt
int n = ....;
char *buf = ..... ; // Assume buf initialized with size n
int result;

_____ ; // initialize variable for loop

... // code that populates buf happens here

while ( _____ ) {
    result = write( _____, _____, _____ );
    if (result == -1) {
        if (errno != EINTR && errno != EAGAIN) {
            // a real error happened, return an error result
            _____ ; // cleanup
            perror("Write failed");
            return -1;
        }
        continue; // EINTR or EAGAIN happened, so loop and try again
    }
    _____ ; // update loop variable
}
_____ ; // cleanup
```

- 3) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?
- 4) Why is it important to remember to call the `close()` function once you have finished working on a file?

5) Given the name of a directory, write a C program that is analogous to `ls`, *i.e.* prints the names of the entries of the directory to `stdout`. Be sure to handle any errors!

Example usage: `./dirdump <path>` where `<path>` can be absolute or relative.

```
int main(int argc, char** argv) {
    /* 1. Check to make sure we have a valid command line arguments */

    /* 2. Open the directory, look at opendir() */

    /* 3. Read through/parse the directory and print out file names
       Look at readdir() and struct dirent */

    /* 4. Clean up */

}
```

References

References create *aliases* that we can bind to existing variables. References are not separate variables and cannot be reassigned after they are initialized. In C++, you define a reference using: **type& name = var**. The '&' is similar to the '*' in a pointer definition in that it modifies the type and the space can come before or after it.

Const

Const makes a variable *unchangeable* after initialization, and is enforced at compile time.

```
const int x = 5;           // Can't assign to x
const int* x_ptr = &x;    // Can assign to x_ptr, but not *x_ptr
int* const y_ptr = &y;    // Can assign to *y_ptr, but not y_ptr
const int* const z_ptr = &z; // Can't assign to *z_ptr or z_ptr
```

Class objects can be declared const too - a const class object can only call member functions that have been declared as const, which are not allowed to modify the object instance it is being called on.

Exercises:

6) Consider the following functions and variable declarations.

- a) Draw a memory diagram for the variables declared in main. It might be helpful to distinguish variables that are constant in your memory diagram.

```
int main(int argc, char** argv) {
    int x = 5;
    int& x_ref = x;
    int* x_ptr = &x;
    const int& ro_x_ref = x;
    const int* ro_ptr1 = &x;
    int* const ro_ptr2 = &x;
    // ...
}
```

- b) When would you prefer `void Func(int &arg);` to `void Func(int *arg);`? Expand on this distinction for other types besides `int`.

c) If we have functions `void Foo(const int& arg);` and `void Bar(int& arg);`, what does the compiler think about the following lines of code:

```
Bar(x_ref);  
Bar(ro_x_ref);  
Foo(x_ref);
```

d) How about this code?

```
ro_ptr1 = (int*) 0xDEADBEEF;  
x_ptr = &ro_x_ref;  
ro_ptr2 = ro_ptr2 + 2;  
*ro_ptr1 = *ro_ptr1 + 1;
```

7) Refer to the following *poorly-written* class declaration.

```
class MultChoice {
public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?

private:
    int q_; // question number
    char resp_; // response: 'A','B','C','D', or 'E'
}; // class MultChoice
```

a) Indicate (Y/N) which *lines* of the snippets of code below (if any) would cause compiler errors:

Code Snippets	Error?
<pre>const MultChoice m1(1, 'A'); MultChoice m2(2, 'B'); cout << m1.get_resp(); cout << m2.get_q();</pre>	

Code Snippets	Error?
<pre>const MultChoice m1(1, 'A'); MultChoice m2(2, 'B'); m1.Compare(m2); m2.Compare(m1);</pre>	

b) What would you change about the class declaration to make it better? Feel free to mark directly on the class declaration above.

Bonus: Given the name of a file as a command-line argument, write a C program that is analogous to `cat`, *i.e.* one that prints the contents of the file to `stdout`. Handle any errors! Example usage: `./filedump <path>` where `<path>` can be absolute or relative.

```
int main(int argc, char** argv) {
    /* 1. Check to make sure we have valid command line arguments */

    /* 2. Open the file, use O_RDONLY flag */

    /* 3. Read from the file and write it to standard out. Try doing
       this without using printf() and instead have write() pipe to
       Stdout (take a look at STDOUT_FILENO). It might be helpful
       to initialize a buffer variable (of size 1024 bytes should be
       fine) to pass in to read() and write(). */

    /*4. Clean up */

}
```


Bonus: Which of the following lines will result in a compiler error?

Code Snippets	Error?	Code Snippets	Error?
<pre>int z = 5; const int* x = &z; int* y = &z; x = y; *x = *y;</pre>		<pre>int z = 5; int* const w = &z; const int* const v = &z; *v = *w; *w = *v;</pre>	

Bonus: What does the following program print out? Hint: box-and-arrow diagram!

```
int main(int argc, char** argv) {
    int x = 1;          // assume &x = 0x7ff...94
    int& rx = x;
    int* px = &x;
    int*& rpx = px;

    rx = 2;
    *rpx = 3;
    px += 4;
    cout << "    x: " << x << endl;
    cout << "   rx: " << rx << endl;
    cout << "  *px: " << *px << endl;
    cout << "   &x: " << &x << endl;
    cout << "  rpx: " << rpx << endl;
    cout << "*rpx: " << *rpx << endl;

    return EXIT_SUCCESS;
}
```

Bonus:

Consider the following C++ code, which has ??? in the place of 3 function names in `main`:

```

struct Thing {
    int a;
    bool b;
};

void PrintThing(const Thing& t) {
    cout << boolalpha << "Thing: " << t.a << ", " << t.b << endl;
}

int main() {
    Thing foo = {5, true};
    cout << "(0) ";
    PrintThing(foo);

    cout << "(1) ";
    ???(foo); // mystery 1
    PrintThing(foo);

    cout << "(2) ";
    ???(&foo); // mystery 2
    PrintThing(foo);

    cout << "(3) ";
    ???(foo); // mystery 3
    PrintThing(foo);

    return 0;
}

```

Program Output:	Possible Functions:
(0) Thing: 5, true	void f1 (Thing t);
(1) Thing: 6, false	void f2 (Thing& t);
(2) Thing: 3, true	void f3 (Thing* t);
(3) Thing: 3, true	void f4 (const Thing& t);
	void f5 (const Thing t);

List *all* of the possible functions (**f1** - **f5**) that could have been called at each of the three mystery points in the program that would compile cleanly (no errors) and could have produced the results shown. There is at least one possibility at each point; there might be more.

- Hint: look at parameter lists and types in the function declarations and in the calls.