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?
   - **More explicit control since read and write functions are system calls and you can directly access system resources.**
   - **POSIX calls are unbuffered so you can implement your own buffer strategy on top of read()/write().**
   - **There is no standard higher level API for network and other I/O devices**

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`.

```
// **NOTE: This is one way to solve this exercise.
// There exist other correct solutions to this exercise.

int fd = open("333.txt", O_WRONLY);   // open 333.txt
int n = ....;
char *buf = ....... ; // Assume buf initialized with size n
int result;

char *ptr = buf;    // initialize variable for loop

...    // code that populates buf happens here

while (ptr < buf + n) {

     result = write(fd, ptr, buf + n - ptr);

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

3) Why is it important to store the return value from the `write()` function? Why don't we check for a return value of 0 like we do for `read()`?

write() may not actually write all the bytes specified in count.

The 0 case for reading was EOF, but writing adds length to your file and we know exactly how much we are trying to write.

4) Why is it important to remember to call the `close()` function once you have finished working on a file?

In order to free resources i.e. other processes can acquire locks on those files.

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 valid command line arguments */
  if (argc != 2) {
    fprintf(stderr, "Usage: ./dirdump <path>\n");
    return EXIT_FAILURE;
  }

  /* 2. Open the directory, look at opendir() */
  DIR* dirp = opendir(argv[1]);
  if (dirp == NULL) {
    fprintf(stderr, "Could not open directory\n");
    return EXIT_FAILURE;
  }

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

  entry = readdir(dirp);
  while (entry != NULL) {
    printf("%s\n", entry->d_name);
    entry = readdir(dirp);
  }

  /* 4. Clean up */
  closedir(dirp);
  return EXIT_SUCCESS;
}
```

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
```

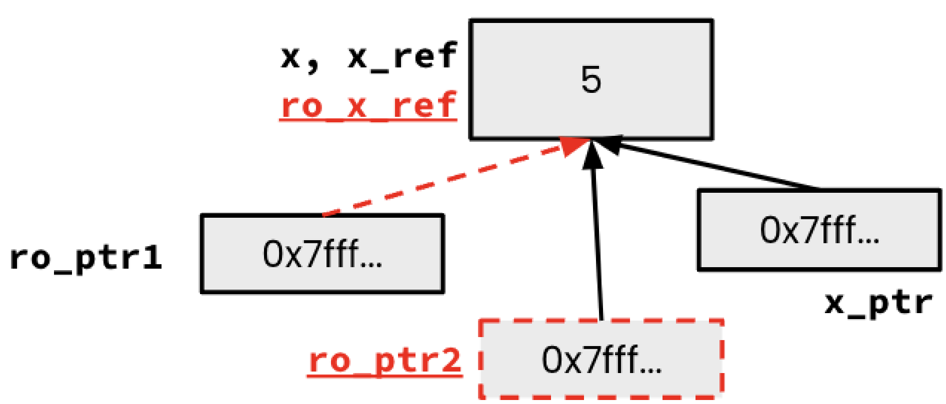
Class objects can be declared const too - a const class object can only call member functions that have been declared as const, which are not allowed to modify the object instance it is being called on.

<u>Exercises</u>:

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`.

- When you don't want to deal with pointer semantics, use references
- When you don't want to copy stuff over (doesn't create a copy, especially for parameters and/or return values), use references
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last

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);                         // No issues
bar(ro_x_ref);                      // Error - ro_x_ref is const
foo(x_ref);                         // No issues
```

d) How about this code?

```
ro_ptr1 = (int*) 0xDEADBEEF;        // No issues
x_ptr = &ro_x_ref;                  // Error - ro_x_ref is const
ro_ptr2 = ro_ptr2 + 2;              // Error - ro_ptr2 is const
*ro_ptr1 = *ro_ptr1 + 1;            // Error - (*ro_ptr1) is const
```

e) In a function `const int f(const int a);` are the `const` declarations useful to the client? How about the programmer? What about this function needs to change to make `const` matter?

The `const` return and parameter both don't affect the client at all, since they work with copies of the parameter/return value. This enforces the programmer not to modify `a` at all. If `f` used references for the parameter/return, then it would matter to both the client and the programmer.

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?
`const MultChoice m1(1,'A');`	N
`MultChoice m2(2,'B');`	N
`cout << m1.get_resp();`	Y
`cout << m2.get_q();`	N

Code Snippets	Error?
`const MultChoice m1(1,'A');`	N
`MultChoice m2(2,'B');`	N
`m1.Compare(m2);`	N
`m2.Compare(m1);`	Y

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

Many possibilities. Importantly, make get_resp() const and make the parameter to Compare() const. Stylistically, it makes sense to add a setter method and default constructor. Could also optionally disable copy constructor and assignment operator.

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 a valid command line arguments */
  if (argc != 2) {
    fprintf(stderr, "Usage: ./filedump <filename>\n");
    return EXIT_FAILURE;
  }
  /* 2. Open the file, use O_RDONLY flag */
  int fd = open(argv[1], O_RDONLY);
  if (fd == -1) {
    fprintf(stderr, "Could not open file for reading\n");
    return EXIT_FAILURE;
  }
  /* 3. Read from the file and write it to standard out.*/
  char buf[SIZE];
  ssize_t len;
  do {
    len = read(fd, buf, SIZE);
    if (len == -1) {
      if (errno != EINTR && errno != EAGAIN) {
        close(fd);
        perror(NULL);
        return EXIT_FAILURE;
      }
      continue;
    }
    size_t total = 0;
    ssize_t wlen;
    while (total < len) {
      wlen = write(1, buf + total, len - total);
      if (wlen == -1) {
        if (errno != EINTR && errno != EAGAIN) {
          close(fd);
          perror(NULL);
          return EXIT_FAILURE;
        }
        continue;
      }
      total += wlen;
    }
  } while (len != 0);
  /*4. Clean up */
  close(fd);
  return EXIT_SUCCESS;
}
```

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

Code Snippets	Error?
`int z = 5;`	N
`const int* x = &z;`	N
`int* y = &z;`	N
`x = y;`	N
`*x = *y;`	Y

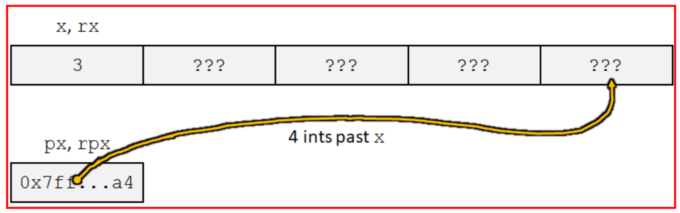
Code Snippets	Error?
`int z = 5;`	N
`int* const w = &z;`	N
`const int* const v = &z;`	N
`*v = *w;`	Y
`*w = *v;`	N

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;  //   x: 3
  cout << "  rx: " <<   rx << endl;  //  rx: 3
  cout << " *px: " <<  *px << endl;  // *px: ??? (garbage)
  cout << "  &x: " <<   &x << endl;  //  &x: 0x7ff...94
  cout << " rpx: " <<  rpx << endl;  // rpx: 0x7ff...a4
  cout << "*rpx: " << *rpx << endl;  // *rpx = *px: ??? (garbage)
  return 0;
}
```



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:  f2
  PrintThing(foo);

  cout << "(2) ";
  __???__(&foo);   // mystery 2:  f3
  PrintThing(foo);

  cout << "(3) ";
  __???__(foo);    // mystery 3:  f1, f2, f4, or f5
  PrintThing(foo);

  return 0;
}
```

Program Output:	**Possible Functions:**

`(0) Thing:  5, true`
`(1) Thing:  6, false`
`(2) Thing:  3, true`
`(3) Thing:  3, true`
	`void f1(Thing t);`
	`void f2(Thing& t);`
	`void f3(Thing* t);`
	`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.