C Programming Tips
Andrew Cortese
Overview:

1. Error Handling
2. Wrapper/Helper Functions
3. Return Values and Conditionals
4. Structures as “classes”
5. Makefiles (With step-by-step example)
Tip #1 - Error Checking

- Many programmers neglect error checking during I/O and memory function calls.
- This is just asking for segmentation faults and other strange program behavior.
- You should perform checks of return values, pointers, lengths, etc whenever possible.
Example: Malloc

Code without error checking:

```c
int main(void)
{
    int length;
    char* string;
    string = (char*) malloc(length);
    return 0;
}
```
Example: Malloc

Improved code:

```c
int main(void)
{
    int length;
    char* string;

    string = (char*) malloc(length);
    if(string == NULL)
    {
        fprintf(stderr, "malloc error");
        exit(1); 
    }

    return 0;
}
```
An Even Better Solution: Wrapper

Create a wrapper function that calls malloc for you, so you can call that function rather than repeating yourself with the extra code.

The wrapper includes the error check and the exit code so you don’t have to type it each time.
Malloc Wrapper

```c
void* checkedMalloc(size_t numBytes)
{
    void * buffer = malloc(numBytes);
    if(buffer == NULL)
    {
        fprintf(stderr, "malloc error");
    }

    return buffer;
}
```

You can now call this in place of malloc and won’t need to do the error check each time.
Other Error Checks: Input

When you call fprintf, fread, gets, and other data input functions, make sure you check the return value. A return value of 0 means no bytes were read, and a return value less than 0 means there was an error.
Other Error Checks: Null Check

Before you dereference (use) a pointer, make sure it isn’t null. Example:

```c
structurePointer = functionCall();
if(structurePointer != NULL) {
    someVariable = structurePointer->memberName;
} else {
    fprintf(stderr, "someVariable is null after functionCall");
    exit(1);
}
```

This will prevent a segfault and instead stop the program and print a useful message.
Other Error Checks: File Access

When you call fopen or fclose, check the return value to be sure it was successful:

```c
FILE* file;
file = fopen("myFilename.txt", "r");
if(fp == NULL) {
    fprintf(stderr, "Could Not Open File");
    exit(1);
}
```
Tip #2: Wrappers / Helper Functions

- We already looked at one wrapper function in action with malloc. Try to apply that concept as often as possible.

- Any time you find yourself repeating code or doing lots of copy/pasting, ask yourself if you could encapsulate the behavior in a function.
Helper Example: Expand An Array

Say you were using an array of integers and found yourself adding values beyond the original length. Each time you added another, you’d have to create a new array with length 1+oldLength, copy everything over, and then add the new value in the new position.
Helper Example: Expand An Array

Instead of copy/pasting this code a dozen times, writing a helper function could help.

By encapsulating the code in a function, we don’t have to repeat ourselves. Debugging also becomes easier, because we only need to look for errors in one spot.
Expanding An Array:

```c
int* intArrayInsert(int* intArray, int valueToInsert) {
    int* newArray;
    int length;
    int i;
    if(intArray == NULL) {
        length = 0;
    } else {
        length = sizeof(intArray) / sizeof(int);
    }
    newArray = (int*) malloc((length+1)*sizeof(int));
    if(newArray == NULL) {
        fprintf(stderr, "malloc error");
        exit(1);
    }
    for(i=0; i<length; i++) {
        newArray[i] = intArray[i];
    }
    newArray[length] = valueToInsert;
    return newArray;
}
```
Tip #3: Beware of Return Values

Any time you’re writing an if statement, a while statement, or other conditional check, I recommend checking for explicit values.

This includes comparisons, equality checks, and return-value investigations. This also means being aware of expected return values.
Example: String Compare

Which of these will execute `someFunction` when `str1` and `str2` are the same?

```c
//example A
if(strcmp(str1, str2))
{
    someFunction();
}

//example B
if(strcmp(str1, str2) == 0)
{
    someFunction();
}
```
Example: String Compare

Which of these will execute `someFunction` when `str1` and `str2` are the same?

```c
//example A
if(strcmp(str1, str2))
{
    someFunction();
}

//example B
if(strcmp(str1, str2) == 0)
{
    someFunction();
}
```

Answer: Example B
Example: Were Any Bytes Read?

Which of these will execute `someFunction()` when bytes were successfully read?

//example A
if(fread(ptr, size, num, file))
{
    someFunction();
}

//example B
if(fread(ptr, size, num, file) == 0)
{
    someFunction();
}
Example: Were Any Bytes Read?

Which of these will execute `someFunction()` when bytes were successfully read?

//example A
if(fread(ptr, size, num, file))
{
    someFunction();
}

//example B
if(fread(ptr, size, num, file) == 0)
{
    someFunction();
}

Answer: Example A
Explanation

- The reason this is important is that sometimes zero indicates a positive statement, as in “the strings are the same.” And other times, it indicates a negative statement, like “no bytes were read.”
- However, conditional statements (if, while) always evaluate true if they see a non-zero value.
Tip #4: Treat Structures Like Classes

Structures in C can be treated very similarly to the way you deal with classes in Java. It takes some extra work, and lacks certain features like inheritance, polymorphism and built-in constructors, but object-oriented programming can be somewhat achieved.
Source Files For Structures

I usually create two files per structure when doing “structure-oriented” programming.

1. A header file with the structure definition, the typedef, and “method” prototypes.

2. A C file for the “method” (function) definitions
Example: “Employee”

Let’s sketch a simple Employee “class” using structs. An employee has:

1. A name (string)
2. An id (int)
3. A “constructor” function for allocation
4. Getter/setter functions for 1 and 2
5. Utility functions (isEqual(), print(), etc)
Employee Structure:

```c
struct employee
{
    char* name;
    int id;
};
typedef struct employee Employee;
typedef struct employee* EmployeePtr;
```

Pretty straightforward so far. Notice the two typedefs: one for a value and one for a pointer. This would all go in employee.h
Prototypes

Next we add function prototypes for our “methods.” These also go in employee.h

```c
EmployeePtr createEmployee(char* name, int id);
char* getName(EmployeePtr e);
int getID(EmployeePtr e);
void setName(EmployeePtr e, char* newName);
void setID(EmployeePtr e, int newID);
int isEqual(EmployeePtr e1, EmployeePtr e2);
void print(EmployeePtr e);
```

Based on the names, you should be able to tell what each one does.
“Methods” (helper functions)

The actual definitions for the functions will go in their own file, called employeeFunctions.c.

We’ll want to #include employee.h in employeeFunctions.c so we have access to the structure definition.
Here is the first part of employeeFunctions.c

```c
#include "employee.h"

// "constructor" function
EmployeePtr createEmployee(char* name, int id)
{
    // allocate for the employee (error check left out for conciseness)
    EmployeePtr e = (EmployeePtr) malloc(sizeof(struct employee));

    // allocate for the name (error check left out for conciseness)
    e->name = (char*) malloc(strlen(name));

    // set the values:
    strcpy(e->name, name);
    e->id = id;

    return e;
}

// other functions would be defined below...
```
employeeFunctions.c

- The getters and setters would access the fields (they should encapsulate any null-checks or mallocs needed).
- The isEqual function would compare two employees by comparing each field.
- The print function would print info about an employee
Using the Employee “class”

To use this “class” in a program, we need to:
1. #include employee.h
2. Make sure employee.h and employeeFunctions.c are in the directory and listed in the makefile
3. Do **NOT** #include employeeFunctions.c (this could cause compilation errors)
Tip #5: Makefiles

Makefiles, as you know, are essentially just scripts you can use to compile your programs.

They are useful because you can issue one command, “make”, to build your whole project with all the necessary files and dependencies.
Simple Makefile Example:

```
 prog:  main.o funct.o
    gcc main.o funct.o -o prog
main.o:  main.c
    gcc -c main.c
funct.o: funct.c
    gcc -c funct.c
```

Important Note

Each gcc command line above starts with the tab character.

The labels on the left are called targets.
To the right of the target are its dependencies.
On the next line is a tab and a build command.
Makefile Tips

● Make sure your default target (the first target) is the desired name of the executable.
● Don’t add extra spaces or newline characters.
● Beware of indentation: build commands need to be indented with 1 tab (no spaces).
● The end of the makefile should have an extra newline.
Makefile Step-By-Step Example

Say we have a program with the following files:

- main.c
- employeeFunctions.c
- employee.h
Assume:

1. We need to access the structure and functions for the employee “class” in main.c
2. employee.h contains the structure definition for the structure, **and** the function prototypes
3. employeeFunctions.c contains the function definitions.
Makefile Example - Dependencies

The default target will be myProg, and its dependencies will be:

1. main.o (the object file for main.c)

2. employeeFunctions.o (object file for employeeFunctions.c)
Makefile:

The makefile so far:

```
myProg: main.o employeeFunctions.o
```

We have our dependencies for the default target. Now we need its build command.
Makefile:

With the build command added:

```
myProg: main.o employeeFunctions.o
gcc main.o employeeFunctions.o -o myProg
```

Notes

- The gcc line is indented with one tab.
- The -o indicates we’re generating an executable from object files.
Makefile:

Our first entry tells make how to create our default target (myProg) from its dependencies (main.o and employeeFunctions.o) by using the specified build command.

Now we need to tell it how to create those individual dependencies.
Makefile: main.o dependency

main.o depends on its source file (main.c) and any files whose code it needs access to (employee.h).

Remember that the function prototypes are in employee.h, so we will #include that in main.c but we won’t #include employeeFunctions.c
Main.o dependencies

Here is the target and dependencies for main.o

```
main.o: main.c employee.h
```

We’ve listed the files that main.o depends on, now let’s add the build command.
Main.o with build command:

```
main.o: main.c employee.h
gcc -c main.c
```

Notes:
- The gcc line is indented with 1 tab.
- The -c indicates that we’re generating an object file from a c source file.
Makefile So Far

```
myProg: main.o employeeFunctions.o
       gcc main.o employeeFunctions.o -o myProg

main.o: main.c employee.h
       gcc -c main.c
```

Note the use of -o for building the default target (since it’s an executable), and the use of -c for building main.o (since it’s an object file).
Next Target

We still need to tell make how to create employeeFunctions.o

This file depends on its source file (employeeFunctions.c) and also depends on the employee.h file, since it needs access to the structure definition.
We have the *target* and *dependencies*. And just like before, we need a *build command*. 
With the build command:

```c
employeeFunctions.o: employeeFunctions.c employee.h
gcc -c employeeFunctions.c
```

Again, note that the gcc line is indented with one tab character (no spaces). Also note that we use -c since we’re generating an object file from a source file.
The makefile so far

```makefile
myProg: main.o employeeFunctions.o
gcc main.o employeeFunctions.o -o myProg

main.o: main.c employee.h
gcc -c main.c

employeeFunctions.o: employeeFunctions.c employee.h
gcc -c employeeFunctions.c
```

We now have rules to build each target from its corresponding dependencies using the specified build command.
Clean target

Last (and yes, least), is the clean target. This gives an easy way to clean the project so you can do a fresh build.

If you’re ever having strange compile/linking errors, it can be useful to try a `make clean` command and then re-issue the `make` to rebuild.
The makefile with “clean”

```
myProg: main.o employeeFunctions.o
    gcc main.o employeeFunctions.o -o myProg

main.o: main.c employee.h
    gcc -c main.c

employeeFunctions.o: employeeFunctions.c employee.h
    gcc -c employeeFunctions.c

clean:
    rm -f *.o core
```

The `rm` line is also indented with a tab. It gives the command to force-delete all object files and core files.
The Employee Program

I will post the entire Employee program, including makefile, on Blackboard. On ITSUnix, you can download it to your folder with the following command:

```
wget http://www.cs.albany.edu/~acortese/CSI402/EmployeeProgram.zip
```

And unzip with:

```
unzip EmployeeProgram.zip
```