Getopt and Debugging
(and some C technicalities)
C technicalities

- **Precedence** is the order in which operators are applied
  - Example: $2+3*4$ means $2+(3*4)$ not $(2+3)*4$
  - Multiplication (*) is has **higher precedence** than addition (+)

- In C, some precedence relationships are non-intuitive
  - Member operator (.) is higher than dereference (*)
    - *ptr.foo means *(ptr.foo) not (*ptr).foo
    - This is partially why “->” is such a useful operator
  - Some unary operators (e.g., ++) are higher than dereference (*)
    - *ptr++ means *(ptr++) not (*ptr)++
    - Use the latter to apply the operator through a dereference

- **Full precedence list:**
• Array names are **aliases**, not pointers
  
  ```c
  int c[4]; // c is not (strictly speaking) a pointer
  int *d = c; // d is a pointer
  ```

  – Practically, they behave like constant pointers
  – Except that `&c == &c[0]` (which is not true of `d`)
    • And `sizeof(c)` is the size (in bytes) of the whole array

  ![Diagram](image)

• Zero-length arrays are (generally) not allowed
  
  ```c
  int a[0]; // compiler warning
  int b[];  // same as "int b[1];"
  ```
C technicalities

- Initializing arrays w/ pointer declaration
  - Generally results in a buffer overrun (compiler warning)
    
    ```c
    int *a = {1, 2, 3, 4};  // buffer overrun!
    ```
  - Special case for C strings:
    
    ```c
    char *s = "hello";  // ok, but read-only
    ```

- String "hello" is stored in a **read-only** section of static data
  - Regardless of whether `s` is local or global
- Pointer `s` is initialized to point to "hello"
- Read-only strings may be re-used by other portions of code
The type "void *" denotes a generic pointer
- No information about what it is pointing to
- Must cast it to a specific pointer type before using it
  - E.g., (int*)ptr
- This can be very dangerous if we're wrong
- Use it sparingly
  - E.g., return value of malloc() where we know the type
C technicalities

- `malloc()` can fail
  - Potential cause: memory leak fills up all available memory
  - If `malloc` fails, it will return `NULL`
  - This will cause a segfault when you try to use the pointer
  - You must check for this every time you call `malloc`
  - Find a graceful and informative way to fail
    - Printing a message and aborting the program is fine in this course

```c
double *temp_data = (double*)malloc(sizeof(double) * ndays);
if (temp_data == NULL) {
    fprintf(stderr, "ERROR: Cannot allocate storage for temperature data\n");
    exit(EXIT_FAILURE);
}
```

<code that uses `temp_data`>
C technicalities

• Memory is uninitialized by default
  - You should manually initialize values to useful defaults if you need to rely on them
  - One easy way to do this: `memset()`
    • Set all bytes in a region of memory to a given character
    • Often used to "zero out" (set to 0) a structure
  - You could also copy from another region with `memcpy()`
    • Inappropriate for strings because it does not append a null terminator
  - If on the heap, you can initialize and allocate with `calloc()`
    • Alternative to `malloc` that will zero out all allocated bytes
    • Slower than `malloc`!
C technicalities

- The C standard does not specify **everything** about how C should be compiled
  - E.g., integer type sizes
  - This allows compiler writers to optimize more highly for a particular architecture (e.g., struct field alignment)

- Printing a null string pointer is **undefined behavior**:

  **7.1.4 Use of library functions**

  Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow: **If an argument to a function has an invalid value (such as** a value outside the domain of the function, or a pointer outside the address space of the program, or **a null pointer**, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after promotion) not expected by a function with variable number of arguments, **the behavior is undefined**. If a function argument is

Thought exercise

- Write a program that takes command-line parameters according to the following usage text:

Usage: ./args [options] <filename>

Valid options:
- \-a        Print a 'A'
- \-b        Print a 'B'

Valid commands:
- ./args file.txt
- ./args -a file.txt
- ./args -a -b file.txt
- ./args -ab file.txt

Invalid commands:
- ./args
- ./args -a
- ./args -c file.txt

What could go wrong?

```c
int main (int argc, char **argv)
{
    // parse options
    for (int i = 0; i < argc; i++) {
        switch (argv[i][1]) {
            case 'a':    a_flag = true;    break;
            case 'b':    b_flag = true;    break;
            default:     report_err();     break;
        }
    }

    // get filename
    char *fn = argv[argc-1];
}
```
Thought exercise

- Write a program that takes command-line parameters according to the following usage text:

  Usage: ./args [options] <filename>

  Valid options:
  -a        Print an 'A'
  -b        Print a 'B'

  Valid commands:
  ./args file.txt
  ./args -a file.txt
  ./args -a -b file.txt
  ./args -ab file.txt

  Invalid commands:
  ./args
  ./args -a
  ./args -c file.txt

What if there's no filename at the end?
What if there are two filenames?
How to handle parameters (e.g., "-n 5")?
How to handle combined flags (e.g., "-ab")?
What if there is no argv[i][1]?

```c
int main (int argc, char **argv)
{
    // parse options
    for (int i = 0; i < argc; i++) {
        switch (argv[i][1]) {
            case 'a':    a_flag = true;    break;
            case 'b':    b_flag = true;    break;
            default:     report_err();     break;
        }
    }

    // get filename
    char *fn = argv[argc-1];
}
```
Getopt

- There's a better way: `getopt()` and `getopt_long()`
  - The latter enables longer options (e.g., "--help")
    - Useful (and mostly standard now), but we won't require it in this course
  - Basic idea: call `getopt()` repeatedly
    - It will return each of the flags individually even if they are grouped or out of order
    - Returns -1 when done
  - Need to pass an `optstring` (list of valid flags as a string)
    - E.g., "abc" indicates that "-a", "-b", and "-c" are valid (any any any combinations)
    - Use a colon to indicate a flag that takes a parameter (e.g., "n:" to allow "-n 4")

- Global variables
  - `optarg`: pointer to string parameter for flags that take them
  - `optind`: index of next flag (use to check for extra arguments at the end!)
#include <getopt.h>

int main (int argc, char **argv)
{
    // parse options
    int opt;
    while ((opt = getopt(argc, argv, "ab")) != -1) {
        switch (opt) {
        case 'a':   a_flag = true;     break;
        case 'b':   b_flag = true;     break;
        default:    report_err();      break;
        }
    }

    // check for and get filename
    if (optind != argc-1) {
        report_err();
        return 1;
    }

    char *fn = argv[optind];
}
Software testing

• **Test-Driven Development**: write the tests first!
  – Popular software engineering technique
  – Describe the behavior of correct code
    • Write a series of test cases to test individual features
    • Make sure you consider edge/corner cases!
    • Save these tests in a test suite that is easy to run
  – THEN write the code
    • Now you have some indication of when you're "done"
    • Write more tests as you go if new cases arise

Project tip: don't rely on the provided test suite—devise your own tests!
“It’s 11pm and I just wrote 500 lines of code!”
  – “All the functions are there.”
  – “I’m done now, right?

“I should probably run some tests”
  – “Just to be sure…”

“@#$%, it’s not working!”
  – “But it looks like it should work…”
A **software defect** is an error in code that produces incorrect or undesired behavior

- Colloquially called “bugs”
- Many types: syntax, logic, integration, concurrency
- Many causes: typos, incorrect code, design flaws, ambiguous spec

**Fundamental issue**: mismatches between user’s expectations and machine’s behavior

- Proximate cause (symptom) vs. root cause (defect)
- **Debugging** is the process of starting from the former and working towards discovering the latter
- Basically: the process of continually asking “why is this happening?”
- One of the most important practical skills in programming
9 Rules of Debugging

1) Understand the system
2) Make it fail
3) Quit guessing and look
4) Divide and conquer
5) Change one thing at a time
6) Keep an audit trail
7) Check the obvious
8) Get a fresh view
9) If you didn’t fix it, it ain’t fixed

Author: David Agans

Recommended book

www.debuggingrules.com
Debugging

● The nature of C makes it possible to explore the kinds of things we want to explore in CS 261
  – However, the power comes at a cost: it is easier to make a mistake!
● Debugging in C will be harder than it was in Java
  – The failure point (e.g., segfault location) is usually not where the bug is!
● Main question: Where is the earliest point at which the program diverges from your expectations?
  – Use debug output or a debugger tool to help
● Other useful questions:
  – What data type(s) are you dealing with?
  – Which memory regions are involved?
  – What is the size and lifetime of the variables?
Debuggers

- A **debugger** (e.g., **gdb**) is a program that allows you to examine another program while it is running
  - Execute the program step-by-step
  - Examine the contents of memory at any point
  - Add **breakpoints** and **watchpoints**
  - Reverse execution to find the root cause

- Debuggers are more useful with extra information from the compiler
  - In gcc, compile with the “-g” option to enable this
  - It’s also useful to disable optimization (“-00”)
Valgrind

- **Valgrind** is a **tool framework** for memory analysis
  - Most useful tool (and the default) is **memcheck**, which searches for memory leaks, uninitialized variables, and other memory problems
  - We use memcheck to check for memory leaks on projects
  - You can use it to help find memory bugs
  - To run: *valgrind <exe-name> <exe-options>*
GDB quick reference

gdb ./program - launch GDB on program (include “--tui” for “graphical” interface)
run <args> - begin/restart execution
start <args> - begin/restart execution and pause at main
break <func> - set a breakpoint ("pause here") at the beginning of a function
break <file>:<line> - set a breakpoint at a specific line of code
watch <loc> - pause when a specific variable or memory location changes
continue - resume execution (until a breakpoint, watchpoint, or segfault)
next - run one line of code then pause (skips over function calls)
step - run one line then pause (descends into functions)
print <expr> - print current value of a variable or expression
print /x <expr> - print current value of a variable or expression in hex
ptype <expr> - print the type of a variable or expression
backtrace - print stack trace (list of active functions on the stack)
                 (up and down to cycle through function call sites)
quit - exit GDB

most of these can be abbreviated to the first letter (e.g., ‘p’ for ‘print’)
(see also CS:APP 3.10.2 and Fig. 3.39)