



Data Structures and Algorithms

Discussion 1





Contents

- Makefiles
- Valgrind
- GDB
- Unit Testing
- C++ Input/Output
- getopt
- CAEN/Remote access



Compiling: The old way

- `g++ main.cpp file1.cpp file2.cpp -o main`
 - Way too long to re-type over and over.
- Potential Problems:
 - `g++ main -o main.cpp`
 - `g++ main.cpp -o main.cpp`
 - What happens in the two above commands?



The Solution: Makefiles

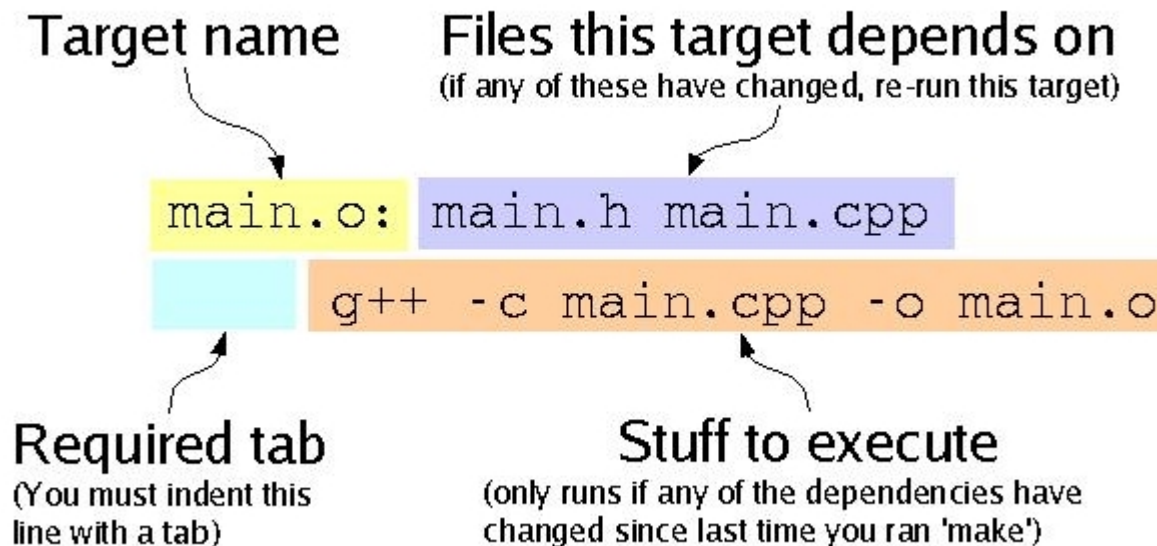
- The command becomes

- ☐ `make`
- ☐ Compare to
- ☐ `g++ main.cpp file1.cpp file2.cpp -o main`

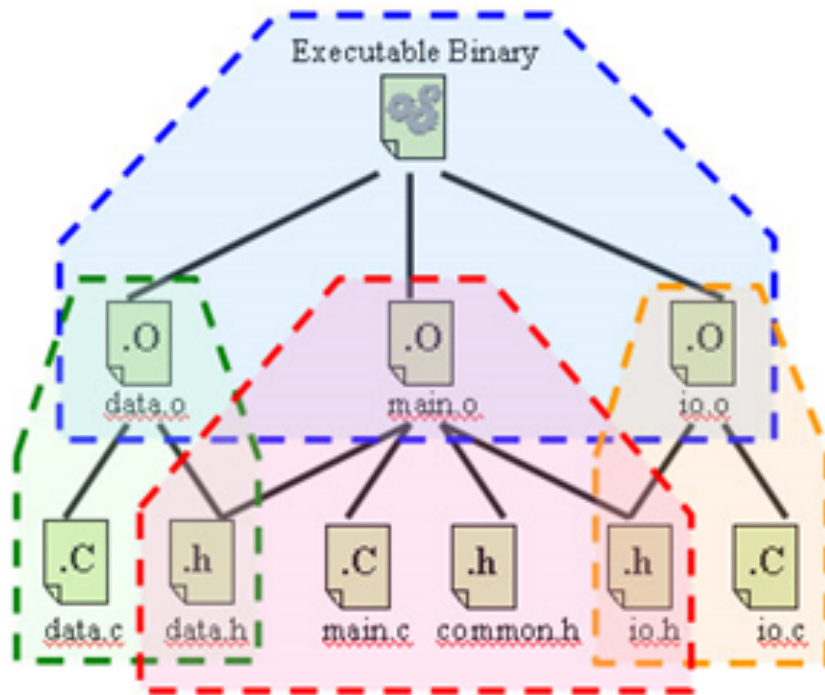


Structure of a Makefile

- The file must be called `Makefile`
- It consists of rules as follows:



Dependencies



```
all: data.o io.o main.o
    gcc data.o io.o main.o -o myexecutable

data.o: data.c data.h
    gcc -c data.c

io.o: io.c io.h
    gcc -c io.c

main.o: main.c data.h io.h common.h
    gcc -c main.c

clean:
    rm *.o myexecutable
```

all and clean

- `all: proj1`
 - Tells make that the main target is the file `proj1`.
- `clean:`
 - `rm exec *.o`
 - Allows make `clean` to clean up generated files (object files and executable)



Macros in Makefile

```
OBJECTS = hello.o classname.o
CPPFLAGS = -c -Wall -Wextra -Wvla -pedantic -O3

# top-level dependency
helloWorld281: $(OBJECTS)
    g++ $(OBJECTS) -o helloWorld281

# hello module
hello.o: hello.cpp classname.h
    g++ $(CPPFLAGS) hello.cpp

# the classname module
classname.o: classname.cpp classname.h
    g++ $(CPPFLAGS) classname.cpp

# clean the project
clean:
    rm -rf *.o helloWorld281
```



Why Object Files?

- Object files (.o) are pre-compiled – much faster to compile to executables.
- If you change one part of a large project, only need to recompile one file from source.
- Much faster overall.



Valgrind



Valgrind

- Detects *undefined behavior*
 - On CAEN (or your local machine), your program may...
 - produce correct output
 - produce wrong output
 - crash with segfault
 - Will crash on autograder



Detected behavior

- Out-of-bounds reads (“invalid read of size...”)
- Out-of-bounds writes (“invalid write of size...”)
- And lastly, memory leaks
 - You should need to manage very little memory manually in 281
- Plugins galore
 - `callgrind`: profile program speed (like `gprof`)
 - `massif`: profile memory usage
 - `helgrind`: check for data races in multithreaded programs — useful for EECS 482
 - and more...



Example

```
int main()
{
    vector<int> foo = {1, 2, 3};
    for (int i = 0; i <= 3; i++) {
        cout << foo[i] << endl;
    }
}
```



```

$ valgrind ./my-program-executable --any-flags --to --program
==32230== ...
1
2
3
==32230== Invalid read of size 4
==32230==    at 0x1000016CC: main ( main.cpp:8)
==32230==    Address 0x100014c8c is 0 bytes after a block of size 12
    alloc'd
==32230==    at 0x66BB: malloc (in /usr/local/...)
==32230==    by 0x4
==32230==    by 0x1
    allocator<int> >:
==32230==    by 0x1
==32230==
0
==32230== ...

```

- Program tried to read out-of-bounds or uninitialized memory.
- size 4: 4 bytes, size of an int.
- Happened on `main.cpp`, line 8
 - Make sure to make with debugging symbols!
- Incorrect output 0 — undefined behavior can do anything!

- Always `valgrind` code before submitting to the autograder
 - If you have undefined behavior, it *will* crash
- Once you know what lines are causing problems, examine further with `gdb`



GDB




GNU Debugger (gdb)

- Text-based debugging tool
- Useful for solving segmentation faults
 - Program received signal SIGSEGV, Segmentation fault.
- Or memory issues, e.g., index out of bounds
- When compiling, add **-g** to Makefile
 - Adds debugging symbols to binary



gdb Usage

- To start gdb, type `gdb <exe_name>`

```
[ 70 ] hw2 -: gdb hw2
GNU gdb Fedora (6.8-29.fc10)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb) 
```

- At this point, gdb is waiting. To start the program, type `run <command line>`

`gdb` Helpful Commands

- `(r)un`: start the executable
- `(b)reak`: sets points where `gdb` will halt
- `where`: prints function line where seg. fault occurred
- `(b)ack(t)race`: prints the full chain of function calls
- `(s)tep`: executes current line of the program, enters function calls
- `(n)ext`: like `step` but does not enter functions
- `(c)ontinue`: continue to the next breakpoint
- `(p)rint <var>`: prints the value of `<var>`
- `watch <var>`: watch a certain variable
- `(l)ist <line_num>` : list source code near `<line_num>`
- `kill`: terminate the executable
- `(q)uit`: quit `gdb`



Unit Testing



Unit testing

- Unit testing is a methodology by which you test the software of your program.
- Tests are *unit* tests because they usually operate on a small, specific part of the system.
- They are organized in classes.
- We can implement an extremely simple version of unit testing with `assert`.



Unit testing

- `assert` is the most important macro in C++ (at least in terms of debugging).
- We assert that something is true. If, when the program counter reaches an assert and the expression turns out to be false, it triggers a breakpoint, and we break into the code.
- It's incredibly useful and convenient. Use it!
 - `assert(3 == 3); // OK`
 - `assert(false == false && "Message Here"); // OK`
 - `assert(myFunc(123)); // OK if myFunc returns true`
 - `assert(5 == 2 + 2); // Will break in code!!!!`



Unit Testing

PowerRaiser.h

```
class PowerRaiser {  
public:  
    PowerRaiser( unsigned int base );  
  
    unsigned int getBase() const;  
    unsigned int raise( unsigned int power ) const;  
  
private:  
    unsigned int base_;  
};
```


Unit Testing

PowerRaiser.cpp

```
PowerRaiser::PowerRaiser( unsigned int base ) :  
    base_( base )  
{  
}  
  
unsigned int PowerRaiser::getBase() const {  
    return base_;  
}  
  
unsigned int PowerRaiser::raise( unsigned int power ) const {  
    if ( 0 == power ) {  
        return 1;  
    } else {  
        return base_ * raise( power - 1 );  
    }  
}
```

Unit Testing

PowerRaiserTest.h

```
class PowerRaiserTest {  
public:  
    void runAllTests();  
    void testGetBase();  
    void testGetPower();  
    ...  
};
```

Unit Testing

PowerRaiserTest.cpp

```
void PowerRaiserTest::testGetBase() {  
    PowerRaiser p( 10 );  
    assert( 10 == p.getBase() && "Failed testGetBase" );  
}
```

```
void PowerRaiserTest::testGetPower() {  
    PowerRaiser p( 3 );  
    assert( 1 == p.raise( 0 ) );  
    assert( 3 == p.raise( 1 ) );  
    assert( 9 == p.raise( 2 ) );  
    assert( 4782969 == p.raise( 12 ) );  
    assert( 15625 == PowerRaiser( 5 ).raise( 6 ) );  
    assert( 1000000 == PowerRaiser( 10 ).raise( 6 ) );  
    ...  
}
```

```
void PowerRaiserTest::testAll() {  
    testGetBase();  
    testGetPower();  
    ...  
}
```



Unit Testing

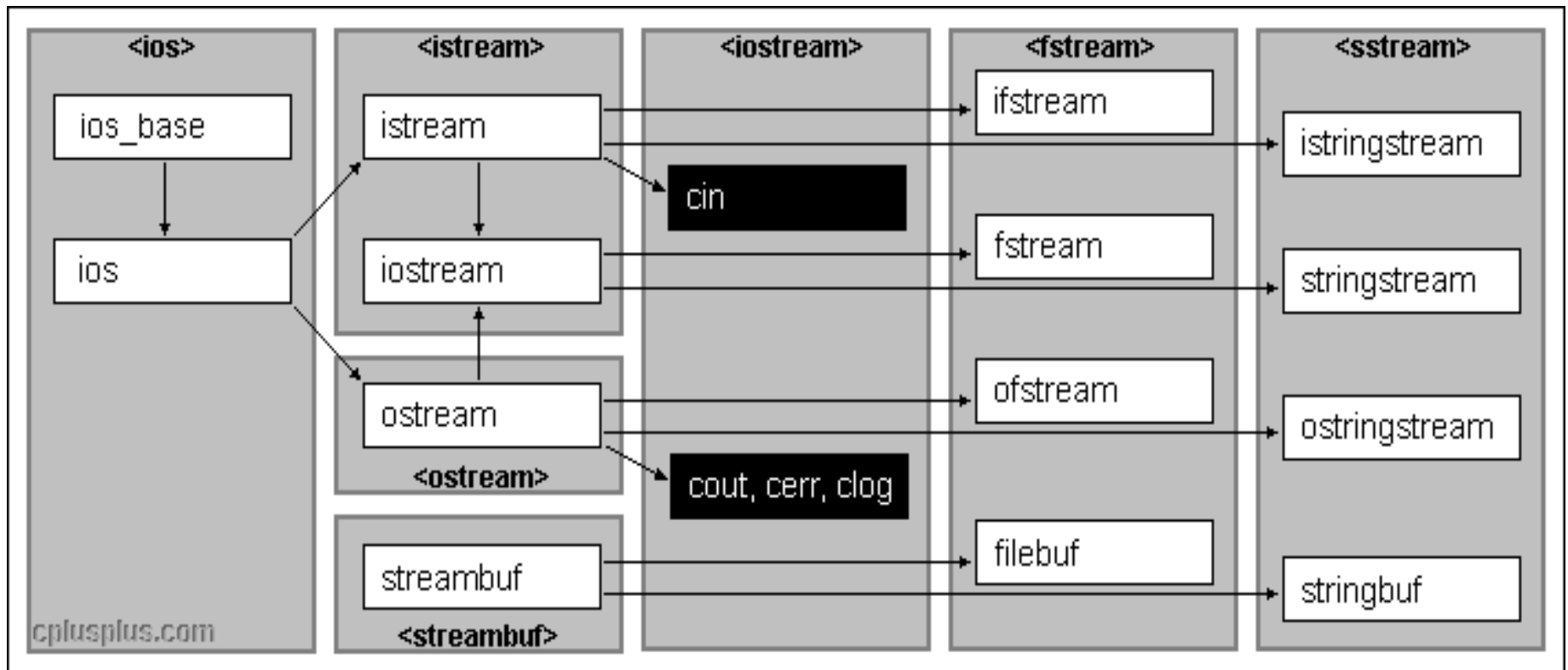
- Each time we make a change to the code base, we run all unit tests to make sure that all of the functionality is still there.
- If an error occurs, it signals a bug. We can figure out where it is with our tests, identify it immediately, and correct it.
- Or, if the bug cannot be resolved, we can **revert** our code (using SVN or CVS, for example) to the prior state.
- Thus, code repositories play a big part in unit testing.



C++ Input/Output



C++ Input/Output



<http://www.cplusplus.com/reference/>

C++ Input/Output

- `cin`: read from stdin
- `cout`: write to stdout
- `cerr`: write to stderr
- `ifstream`: open files with read permission
- `ofstream`: open files with write permission
- `fstream`: open files with read & write permission



C++ stdin

- Input/output is redirected from files
- `./path281 < input.txt > output.txt`
- `<` and `input.txt` are not command line args, do not appear in `char** argv`
- You can use
 - `getline(cin, var)`
 - `cin >> var`



Read char by char

```
int main(int argc, char** argv)
{
    char c;
    while (cin >> c) {
        cout << c;
    }
}
```



Read line by line

```
int main(int argc, char** argv)
{
    string s;
    while (getline(cin, s))
    {
        cout << s << endl;
    }
}
```



Printing line by line

```
int main(int argc, char** argv)
{
    for(int i = 0; i < 100; ++i)
        cout << i << endl;
}
```

- Inefficient
- Using `endl` instead of `'\n'` causes us to flush the buffer after every number.

Using Internal Buffers

```
#include <sstream>
int main(int argc, char** argv)
{
    ostringstream ss;
    for(int i = 0; i < 100; ++i)
        ss << i << '\n';
    cout << ss.str();
}
```

- `ostringstream` has the same syntax as `cout`



I/O Tips

- 1. When I/O becomes a bottleneck, avoid reading/writing character-by-character or word-by-word.
- 2. While C++ streams are slower than stdlibc I/O, this can be changed by setting `std::ios_base::sync_with_stdio` to `false` (and sometimes by setting it to `true`).



getopt



Getopt Overview

- Motivation: simplifies command line parsing
 - `./exec -a mergesort -V 1.0`
 - `ls -ltr`
- Example in next slide for the skeleton of getopt
- To read more about getopt and how to use it, check out: <http://www.ibm.com/developerworks/aix/library/au-unix-getopt.html>



getopt

```
#include <unistd.h>
#include <stdio.h>

int main (int argc, char **argv) {
    int aflag = 0; int bflag = 0;
    char *cvalue = nullptr;
    int index, c;
    opterr = 0; // extern var

    while ((c = getopt (argc, argv, "abc:")) != -1) {
        switch (c) {
            case 'a':
                aflag = 1; break;
            case 'b':
                bflag = 1; break;
            case 'c':
                cvalue = optarg; break;
            case '?':
                if (isprint (optopt))
                    fprintf (stderr, "Unknown option `-%c'.\n" , optopt);
                else
                    fprintf (stderr, "Unknown option character `\\x%x'.\n" , optopt);
                return 1;
            default:
                abort ();
        }
    }

    printf ("aflag = %d, bflag = %d, cvalue = %s\n" ,
           aflag, bflag, cvalue);

    for (index = optind; index < argc; index++)
        printf ("Non-option argument %s\n" , argv[index]);
    return 0;
}
```



getopt

```
% testopt
aflag = 0, bflag = 0, cvalue = (null)

% testopt -a -b
aflag = 1, bflag = 1, cvalue = (null)

% testopt -ab
aflag = 1, bflag = 1, cvalue = (null)

% testopt -c foo
aflag = 0, bflag = 0, cvalue = foo

% testopt -cfoo
aflag = 0, bflag = 0, cvalue = foo

% testopt arg1
aflag = 0, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -a arg1
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -c foo arg1
aflag = 0, bflag = 0, cvalue = foo
Non-option argument arg1

% testopt -a -- -b
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -b
```

```
% testopt -a -
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -
```

getopt_long Overview

- Useful for parsing *multi-character* command line arguments
 - Ex: `./my.exe --help` // displays help
- Same principle as getopt
- Syntax:

```
getopt_long(int argc, char **argv,           // comes directly from int
            const char *short_options,       // same as getopt's options
            const struct option *long_options, // see next slide
            int *index_ptr)                  // index of option in
                                            // long_options
```

http://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Option-Example.html



getopt_long Long Options

```
struct option
{
    const char *name; // full name of option
    int has_arg;      // can take following values:
                      // no_argument
                      // required_argument
                      // optional_argument

    int *flag;        // option can set a flag
    int val;           // (short) value of option
                      // can be used for getopt
};
```

http://www.gnu.org/software/libc/manual/html_node/Getopt.html



CAEN Remote Access



Connect to CAEN

- Remote CAEN Virtual sites
- Remote CAEN VNC
 - <http://www.engin.umich.edu/caen/connect/vnc.html>
- SFTP Transfer from your local machine
 - <http://caen.engin.umich.edu/connect/sftp>



Pure SSH

- Unix/Linux/Mac

- Open terminal
- Run `ssh username@login.engin.umich.edu`

- Windows

- Download SSH client (such as PuTTY)
- Specify hostname as `username@login.engin.umich.edu`



Compatibility with CAEN

- Your code **must** compile and run correctly with `g++` as it is on the CAEN machines.
- You still might want to develop on your own computer.
 - If you do, you must find a way to test on CAEN `g++` as well.



Ways To Test on CAEN

- Develop on your machine and copy to CAEN when you finish.
 - Leave plenty of time to debug, because small errors could become system-crashers on a different system.
- Develop directly on CAEN using either SSH or VNC or CAEN computers.
- Edit remote files with a local program.