Data Structures and Algorithms

eecs 281

Discussion 1: Week of Sep 7, 2011

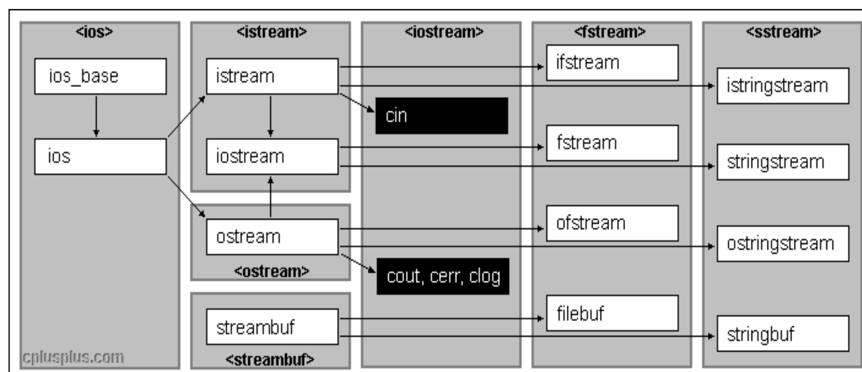
MichiganEngineering

Contents

- C++ Input/Output
- getopt
- Makefile
- gdb
- Unit Testing

MichiganEngineering

C++ Input/Output



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

MichiganEngineering

C++ Input/Output

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

MichiganEngineering

PA1

- 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
  `cin.get(), cin.unget(), getline(cin, var), cin >> var`

Read char by char

```
int main(int argc, char** argv)
{
      while(cin.get() != -1)
      {
            cin.unget();
            char c = cin.get();
            cout << c;
      }
      return 0;
}
```

Read line by line

```
int main(int argc, char** argv)
{
      string s;
      while(getline(cin,s))
      {
            cout << s << endl;
      }
      return 0;
}
```

Command Line

- Option: defines how the program should run
- Argument: input to the program
- $ ls –a *(-a is an option)*
- $ ls *.cpp *(*.cpp is an argument)*
- $ mysql –p –u username (username is required by option –u)

getopt

- *getopt* greatly simplifies command-line argument parsing.
- We pass it *argc* and *argv,* as well as *optstring*, which is a character array containing all available option characters.
  - After an option character, you can specify a : to indicate that the option has a required parameter. :: denotes that the argument is optional.
- *getopt* returns the option if it successfully found one.
- Otherwise, it might return : which indicates a missing parameter
- Or it might return ? which means one of the option characters was unknown
- Or it will return -1 if it is at the end of the option list

getopt – external variable soup

- *getopt* gets and sets some external variables on execution:
  - If you set ***opterr*** to something other than zero, then if there is an error, it will be output to stderr.
  - For unknown option characters/unknown arguments, *getopt* stores the value in ***optopt***.
  - ***optind*** contains the index of the next element of *argv* scheduled to be processed.
  - For options that accept arguments, ***optarg*** is set with the value of the argument.
- Windows users can get wingetopt from: http://note.sonots.com/Comp/CompLang/cpp/getopt.html
- Let's look at an example.

getopt

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

 int main (int argc, char **argv) {
   int aflag = 0; int bflag = 0;
   char *cvalue = NULL;
   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 -
```

Compiling: The old way

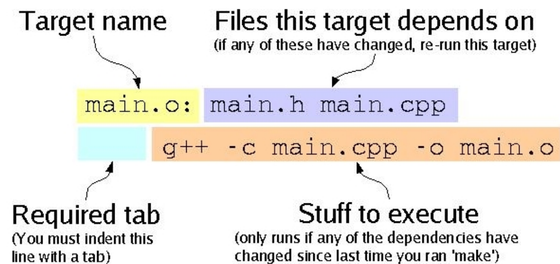
- `g++ main.cpp file1.cpp file2.cpp -o main`
  - Way too long to re-type over and over.
- Potential Problems:
  - `g++ main -o main.cpp`
  - `g++ main.cpp -o main.cpp`
    - What happens in the two above commands?

The Solution: Makefiles

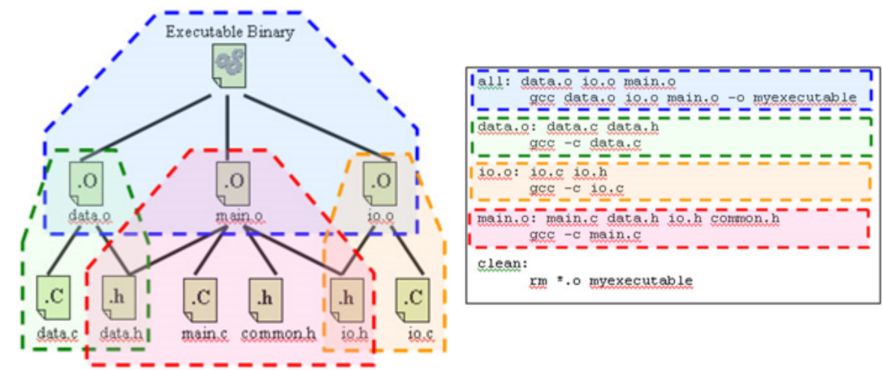
- The command becomes
  - `make`
  - Compare to
  - `g++ main.cpp file1.cpp file2.cpp -o main`

Structure of a Makefile

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



Dependencies



make

- make is a utility that can simplify the building process. As the size of a C++ project grows, it becomes critical to modularize the code for the sake of both the build time and for the sanity of the programmer (and the grader!).
- make is better than a shell script because it detects the files that have been changed and only recomplies and relinks when needed.
- make looks at a file called Makefile, created by you, to determine the dependency graph of your project.

MichiganEngineering

make

- Let's look at an example 3-file project.

classname.h

```
void writeClassName( void );
```

classname.cpp

```
#include <iostream>

using namespace std;

void writeClassName( void ) {
      cout << "Hello EECS 281!" << endl;
}
```

hello.cpp

```
#include <iostream>
#include "classname.h"

using namespace std;

int main( void ) {
        cout << "Hello world!" << endl;
        writeClassName();
}
```

MichiganEngineering

make - Makefile

- Here is our simple Makefile:

```
# top-level dependency
helloWorld281: hello.o classname.o
        g++ hello.o classname.o -o helloWorld281

# hello module
hello.o: hello.cpp classname.h
        g++ -c hello.cpp

# the classname module
classname.o: classname.cpp classname.h
        g++ -c classname.cpp
```

- We define a *dependency graph* in Makefile. To the left of the colon is a label often denoting a file. To the right are dependencies. For example, helloWorld281 depends on hello.o and classname.o.
- The indented lines are, in a nutshell, the commands executed to create the object denoted by the label. Compiled in order. *These lines must be indented.*
- # denotes comments

MichiganEngineering

make – macros

- It's possible to define and use macros in your Makefile:

```
OBJECTS = hello.o classname.o
CPPFLAGS = -c -Wall

# 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
```

- Note above we've also defined a "clean" operation. This is an action, not a g++ operation. To use, type *make clean* at the command prompt.

MichiganEngineering

make - sub-makefiles

- *You can also embed other makefiles inside Makefile. This allows you to modularize your build process even further. To specify a make file with a different name than Makefile, use the -f option:*

```
make -f MyOtherMakeFile
```

make

- *make is a powerful and complex utility, and we're just barely scratching the surface. For more information, do a man make or check out one of the many tutorials on the web.*
- *http://www.eng.hawaii.edu/Tutor/Make/*

Makefiles Summary

- Basic syntax
  - *Target: sources*
  - *[tab] system command*
- To run a makefile, simply use:
  - *make -f makefilename*
- BUT, if you simply name your makefile "Makefile", then you only have to type:
  make

Makefiles Summary

- Suppose we run the following command to compile our program:
  - g++ main.cpp hello.cpp factorial.cpp -o hello
- Then we can do the same thing in our makefile by just doing this:

```
all:
    g++ main.cpp hello.cpp factorial.cpp -o hello
```

- Remember tabs, of course
- All is default target for makefiles. That's why it works here.

Makefiles Summary

- Dependencies are important!
- Here is an example makefile for the same source code:

```
all: hello
hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello
main.o: main.cpp
	g++ -c main.cpp
factorial.o: factorial.cpp
	g++ -c factorial.cpp
hello.o: hello.cpp
	g++ -c hello.cpp
clean:
	rm -rf *o hello
```

- What advantages does this code have?

MichiganEngineering

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

MichiganEngineering

`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>`

MichiganEngineering

`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 <u>not</u> 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

MichiganEngineering

gdb Live Example

- See live example…

MichiganEngineering

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.*

MichiganEngineering

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 ); // OK
assert( myFunc( 123 ) ); // OK if myFunc returns true
assert( 5 == 2 + 2 ); // Will break into code!!!!
```

MichiganEngineering

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_;
};
```

MichiganEngineering

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 == number ) {
                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() );
}

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.