DATA STRUCTURES AND ALGORITHMS

Discussion: October 10, 2007

The University of Michigan

Agenda

- Questions on anything
- Makefiles
- GDB
- Bit operations

Questions on anything?

- Questions?

Makefiles revisited

Basic syntax

Target: dependencies

[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 revisited

- 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 revisited

- Why is this so basic compared to before (a.k.a. I hate you, GSI)?

Makefiles revisited

- 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?

Makefiles revisited

- Let's look at an example

Makefiles revisited

- You can use comments with #
- Again, we can have macros:

```
CC=g++

CFLAGS=-c -Wall

LDFLAGS=

SOURCES=main.cpp hello.cpp factorial.cpp

OBJECTS=$(SOURCES:.cpp=.o)

EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)

    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:

    $(CC) $(CFLAGS) $< -o $@
```

Makefiles revisited

- If you understand the last example, you can modify it by changing only two lines, no matter what files you have in your project!
- Here it is again:

```
CC=g++

CFLAGS=-c -Wall

LDFLAGS=

SOURCES=main.cpp hello.cpp factorial.cpp

OBJECTS=$(SOURCES:.cpp=.o)

EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)

    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:

    $(CC) $(CFLAGS) $< -o $@
```

Makefiles revisited

- Questions on makefiles?

GDB revisited

- Let's look at an example

```
kirbyb@myprompt> gdb main
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb)
```

- Right now, gdb is just waiting

GDB revisited

- Let's say we try to run a program that we've got

(gdb) run

Starting program: /home/cec/s/a/agg1/.www-docs/tutorial/main

Creating Node, 1 are in existence right now

Creating Node, 2 are in existence right now

The fully created list is:

2

1

Now removing elements:

Creating Node, 3 are in existence right now

Destroying Node, 2 are in existence right now

2

1

(continued on next page)

GDB revisited

```
Program received signal SIGSEGV, Segmentation fault.
Node<int>::next (this=0x0) at main.cc:28
28          Node<T>* next () const { return next_; }
(gdb)
```

- Oops, we've got a segfault. What do we do now?
- Well, what do we know about the error at this point?
  - (this)
- What do we still want to know?

GDB revisited

- It'd be useful to go backwards, and see what values were at places *before* the error

```
(gdb) backtrace
#0  Node<int>::next (this=0x0) at main.cc:28
#1  0x2a16c in LinkedList<int>::remove (this=0x40160,
    item_to_remove=@0xffbef014) at main.cc:77
#2  0x1ad10 in main (argc=1, argv=0xffbef0a4) at main.cc:111
(gdb)
```

- Ah, now we know a bit more about how we got to the error...
- But, how do we figure out what we're trying to remove?

GDB revisited

- We can actually take a look at memory addresses!

```
(gdb) x 0xffbef014
0xffbef014:  0x00000001
(gdb)
```

- This tells us exactly what's in that address!
- And how did we know the address was 0xffbef014?

GDB revisited

- OK, we can look at memory values, but what about breakpoints again?

```
(gdb) break LinkedList<int>::remove
Breakpoint 1 at 0x29fa0: file main.cc, line 52.
(gdb)
```

- Here's the classic way to make a breakpoint.
  - When we do a run with GDB, it will stop here
- And conditional breakpoints:

```
(gdb) condition 1 item_to_remove==1
(gdb)
```

- This means “only stop at breakpoint 1 if item_to_remove == 1”

GDB revisited

- Stepping is also useful.
  - Simply type “step” to go to the next line, after a breakpoint
- And, finally, you can quit gdb by typing “quit”
- Use google as a reference for other gdb commands!

Bit operations

- Does anyone know what bit operations are?

Bit operations

- Bits are fundamental units in computers.
- A lot of bit operations simply come down to logic!
- Recall truth tables:

F ^ F = F

T ^ F = F

F ^ T = F

T ^ T = T

- Other examples

Bit operations

- Well, we can represent T as 1 and F as 0, and C++ (and C) has equivalent functionality!

0 & 0 = 0

1 & 0 = 0

0 & 1 = 0

1 & 1 = 1

Bit operations

- Bitwise operator meanings:

  &: binary bitwise AND

  ^: binary bitwise exclusive OR (XOR)

  | : binary bitwise inclusive OR

  ~ : unary bitwise complement (NOT)

- What are the values in the following truth table?

  0 ^ 0 = ?

  1 ^ 0 = ?

  0 ^ 1 = ?

  1 ^ 1 = ?

Bit operations

- Remember not to mix these operators up with standard logical operators.
- What is the following pseudo-code-segment doing?

```
if((x==y) & (z!=x))
```

Bit operations

- Does anyone know what bit shifting is?
- How is it useful?

Bit operations

- What is the value of x after the following code is executed?

```
Int x = 5;

int y = 2;

x = y << 5;
```

- A useful property of bit shifting! Powers of 2 are cool!
- Similarly, you can shift in the other direction with: >>

Questions?

Any questions?