COMP2004 Programming Practice 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Unix development tools

- make
  - Automatic code compilation
- gdb
  - Debugging
- shell
  - Simple text-manipulation programs
  - Bourne shell vs bash
  - Many other utilities

make

```
OBJS = main.o student.o course.o
CXXFLAGS = -Wall -g
LDFLAGS = -lm

all: prog
prog: $(OBJS)
   $(CXX) $(CXXFLAGS) $(LDFLAGS) \
                          -o prog $(OBJS)
clean:
   rm -f $(OBJS)
```

gdb

- Compiled with -g
- Crashed with
  Segmentation fault (core dumped)
- Run: gdb prog core
- Common commands
  - bt
    - Get a stack backtrace
    - Shows where the crash happened
  - p head
    - Print value of variable head

Simple shell

- Simple shell commands
  - cd, cp, mv, rm
  - echo - Prints parameters to stdout
  - cat - Prints files (or stdin) to stdout
- Redirection
  - echo "foo" > foo.txt
  - echo "bar" >> foo.txt
  - cat < hello.txt
- Redirection caveat
  - prog file.txt > file.txt

Shell / environment variables

- Assign:
  - variable_name="value of variable"
- Use:
  - echo "$variable_name"
- Shell variable -> environment variable:
  - export variable_name

If

- Exit status:
  - 0 is true, everything else is false
  - exit n command in shell

```
if cmp file1.txt file2.txt; then
   echo "files same"
elif diff -i file1.txt file2.txt; then
   echo "files same except case"
else
   echo "files different"
fi
```

Alternate if

- if ["$somevar" = "hello"]; then
- if ["$somevar" != "hello"]; then
- if ["$num" -lt 42]; then
- if ["$num" -ge 42]; then
- if [-r "$fname"]; then
- if ["$fname" -nt "file.txt"]; then
- and so on...

stdout -> shell variable

- Use backticks or $() in bash:

```
echo "f1.h f2.h f3.h" > file.lst
filelist=`cat file.lst`
cat $filelist
cat `cat file.lst`
cat $(cat file.lst)
```

- Shell arithmetic (expr in Bourne, $(()) in bash):
  - result=`expr 3 + 8 \* 2`
  - result=$((expr 3 + 8 * 2))

while

```
count=1
while [ $count -le 10 ]; do
      echo "$count"
      count=$(($count + 1))
done

while :; do
      if [ -e "file.txt" ]; then
            break
      fi
      sleep 1
done
```

Command line parameters

- The shell script sample:

```
#!/gnu/usr/bin/bash
while [ $# -ge 2 ]; do
      echo "$1" "$2"
      shift
done
```

```
bash$ sample a bc defgh
a bc
bc defgh
```

case

```
case "$1" in
      -d*)
            cmd="diff"
            ;;
      -c*)
            cmd="cmp"
            ;;
      *)
            exit 1
            ;;
esac
$cmd file1.txt file2.txt
```

for and pipes

```
#!/gnu/usr/bin/bash
for infile in tests/*.in; do
    name=`basename $infile .in`
    if prog < $infile | cmp - \
                    tests/$name.out; then
        echo "$name : passed"
    else
        echo "$name : failed"
    fi
done
```

Useful Unix utilities

- sed - stream editor
- awk - processing columnar data
- sort - sorts lines in a file
- uniq - removes duplicate lines
- head - output first n lines
- tail - output last n lines
- cut - extracts columns of characters
- join, paste - merges files by columns

Course survey

- Voluntary and anonymous
  - So don't write your name on it
- 15 minutes
  - I won't be here
  - Student representative collects and seals forms
- Comments are useful for improving all courses
- Unit of Study = Programming Practice
- Unit of Study Code = COMP2004

Advanced C++

- Generic code
  - Template functions
  - Template classes
- Exceptions
- Exception Safety
- String streams

Template Functions

```
template<typename T>
typename vector<T>::size_type
find(const vector<T> &v,
                    const T &value) {
    for (typename vector<T>::size_type
                i = 0; i < v.size(); ++i)
        if (v[i] == value) return i;
    return v.size();
}
vector<int> vi;
find(vi, 42);
```

Template Classes

```
template<typename T>
class Node {
    T value;
    Node<T> *next;
    ...
};
```

- Heavy reliance on operators
- All template code in .h files
  - Only time this is allowed

Exceptions

- Seperate error detection from handling
- Any object can be exception
- First catch type matched is used (including inheritance)
- `void func() throw (e1, e2)`
- `void func() throw ()`
- `void func()`
- Constructors can throw exception
- Copy constructors and destructors shouldn't

Exceptions

```
struct Error { int i;
    Error(int i) : i(i) {} };
struct OtherError { };

try {
    throw Error(42);
} catch (Error e) {
    cout << e.i << endl;
    throw OtherError();
} catch (...) {
    throw;
}
```

Exception Safety

- Each function must:
- Basic Guarantee
  - Resources not leaked, objects still usable
- Strong Guarantee
  - Program state is as before the call
- Nothrow
  - The function will never throw

Exception Unsafe Code

```
void some_function(string name) {
    Person *fred = new Person();

    fred->setName(name);

    delete fred;
}
```

Fixing with try/catch

```
void some_function(string name) {
    Person *fred = new Person(name);
    try {
        fred->setName(name);
    } catch (...) {
        delete fred;
        throw;
    }
    delete fred;
}
```

Fixing with auto_ptr

```
void some_function(string name) {
    Person *fredp = new Person(name);
    auto_ptr<Person> fred(fredp);

    fred->setName(name);
}
```

String streams

- Old - istrstream / ostrstream
- New - istringstream / ostringstream
- Allow input / output to / from strings using normal << and >>

New style

```
int main() {
        string s = "42 15";
        istringstream is(s);
        int i, j;
        is >> i >> j;

        ostringstream os;
        os << i << "." << j;
        s = os.str();
        cout << s << endl;
}
```