Git and GitHub ... then C
Agenda

Our computing environment
• Lecture 1 and Precepts 1 and 2: Linux and Bash
• Lecture 2: git and GitHub

A taste of C
• History of C
• Building and running C programs
• Characteristics of C
• Example program: charcount
Revision Control Systems

Problems often faced by programmers:

• Help! I’ve deleted my code! How do I get it back?
• How can I try out one way of writing this function, and go back if it doesn’t work?
• Help! I’ve introduced a subtle bug that I can’t find. How can I see what I’ve changed since the last working version?
• How do I work with source code on multiple computers?

• How do I work with others (e.g., a COS 217 partner) on the same program?
• What changes did my partner just make?
• If my partner and I make changes to different parts of a program, how do we merge those changes?

All of these problems are solved by revision control tools, e.g.: 

`git`
Repository vs. Working Copy

WORKING COPY

• Represents single version of the code
• Plain files (e.g., .c)
• Make a coherent set of modifications, then commit this version of code to the repository
• Best practice: write a meaningful commit message

REPOSITORY (or “repo”)

• Contains all checked-in versions of the code
• Specialized format, located in .git directory
• Can view commit history
• Can diff any versions
• Can check out any version, by default the most recent (known as HEAD)

‡ We’ll rarely use checkout except to throw away local changes (see slide 6)
Relevant xkcd

- Created main loop & timing control
- Enabled config file parsing
- Misc bug fixes
- Code additions/edits
- More code
- Here have code
- AAAAAAAA
- ADKFJSLKDFJSKLFJ
- My hands are typing words
- HAAAAAAAANDS

As a project drags on, my git commit messages get less and less informative.

https://xkcd.com/1296/
Local vs. Remote Repositories

**LOCAL REPOSITORY**
- Located in `.git` directory
- Only accessible from the current computer
- Commit early, commit often – you can only go back to versions you’ve committed
- Can **push** current state (i.e., complete checked-in history) to a remote repository

**REMOTE REPOSITORY**
- Located in the cloud E.g., github.com
- Can **clone** working copies on multiple machines
- Any clone can **pull** the current state

**Commands**
- `git push`
- `git clone`
- `git pull`
We distribute assignment code through a github.com repo

- But you can’t push to our repo!

Need to create your own (private!) repo for each assignment

- Two methods in git primer handout
- One clone on armlab, to test and submit
- If developing on your own machine, another clone there: be sure to commit and push "up" to github, then pull "down" onto armlab
Agenda

Our computing environment
  • Lecture 1 and Precepts 1 and 2: Linux and Bash
  • Lecture 2: git

A taste of C
  • History of C
  • Building and running C programs
  • Characteristics of C
  • Example program: charcount
The C Programming Language

Who? Dennis Ritchie

When? ~1972

Where? Bell Labs

Why? Build the Unix OS

Read more history: https://www.bell-labs.com/usr/dmr/www/chist.html
Java vs. C: History

This is what we're using


BCPL → B → C → K&R C → ANSI C89 ISO C90 → ISO/ANSI C99 → ISO C11 → ISO C18
Algol
Simula
LISP
Smalltalk

C++ → Java
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Build the Unix OS</td>
<td>Language of the Internet</td>
</tr>
<tr>
<td>Low-level; close to HW and OS</td>
<td>High-level; insulated from hardware and OS</td>
</tr>
<tr>
<td>Good for system-level programming</td>
<td>Good for application-level programming</td>
</tr>
<tr>
<td>Support structured programming</td>
<td>Support object-oriented programming</td>
</tr>
<tr>
<td>Unsafe: don’t get in the programmer’s way</td>
<td>Safe: can’t step “outside the sandbox”</td>
</tr>
<tr>
<td></td>
<td>Look like C!</td>
</tr>
</tbody>
</table>
Agenda

Our computing environment
• Lecture 1 and Precepts 1 and 2: Linux and Bash
• Lecture 2: git

A taste of C
• History of C
• Building and running C programs
• Characteristics of C
• Example program: charcount
Building Java Programs

$ javac MyProg.java

Windows (ArmLab)

MyProg.java
(Java code)

javac

Java compiler
(machine lang code)

OS
(Linux)

MyProg.class
(bytecode)
Running Java Programs

$ java MyProg

Java interpreter / “virtual machine” (machine lang code)

HW (ArmLab)

OS (Linux)

MyProg.class (bytecode)

data

data
Building C Programs

$ gcc217 myprog.c –o myprog

C “Compiler driver” (machine lang code)
Running C Programs

$ ./myprog

myprog
(machine lang code)

HW (ArmLab)

OS (Linux)

myprog

data

data
Agenda

Our computing environment
• Lecture 1 and Precepts 1 and 2: Linux and Bash
• Lecture 2: git

A taste of C
• History of C
• Building and running C programs
• Characteristics of C
• Example program: charcount
## Java vs. C: Portability

### Table:

<table>
<thead>
<tr>
<th>Program</th>
<th>Code Type</th>
<th>Portable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyProg.java</td>
<td>Java source code</td>
<td>Yes</td>
</tr>
<tr>
<td>myprog.c</td>
<td>C source code</td>
<td>Mostly</td>
</tr>
<tr>
<td>MyProg.class</td>
<td>Bytecode</td>
<td>Yes</td>
</tr>
<tr>
<td>myprog</td>
<td>Machine lang code</td>
<td>No</td>
</tr>
</tbody>
</table>

### Conclusion:
Java programs are more portable

(For example, COS 217 has used many architectures over the years, and every time we've switched, all our programs have had to be recompiled!)
Java vs. C: Safety & Efficiency

Java

• null reference checking
• Automatic array-bounds checking
• Automatic memory management (garbage collection)
• Other safety features

C

• NULL pointer checking,
• Manual bounds checking
• Manual memory management

Conclusion 1: Java is often safer than C

Conclusion 2: Java is often slower than C
Occasional questions in class, graded on participation not correctness.

- Using an app on your phone or the web client

- Setup is "iClicker Cloud", integrated with our course's Canvas.

- Register, select Princeton University, and find course "COS 217 – Fall 2023"
Q: Can you answer this iClicker question today?

• A. Yes

• B. No, but I’ve been practicing my mental electrotelekinesis and the response is being registered anyway

• C. I’m not here, but someone is iClicking for me (don’t do this – it’s a violation of our course policies!)
Q: Which corresponds to the C programming language?

A. 

B. 

C.
Next 7 slides show C language details by way of Java comparisons.

For now, use as a comparative language overview reference to start the simple "syntax mapping" stage of learning C, so that you're well prepared to dive into the less rote aspects in the coming weeks.
### Java vs. C: Details

<table>
<thead>
<tr>
<th>Overall Program Structure</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
</table>
| Hello.java:               | public class Hello  
  {  public static void main  
      (String[] args)  
      {  System.out.println(  
            "hello, world");  
      }  
  }  |
| hello.c:                  | #include <stdio.h>  
  int main(void)  
  {  printf("hello, world\n");  
      return 0;  
  }  |

<table>
<thead>
<tr>
<th>Building</th>
<th>$ javac Hello.java</th>
<th>$ gcc217 hello.c –o hello</th>
</tr>
</thead>
</table>
| Running                   | $ java Hello  
  hello, world  
  $ | $ ./hello  
  hello, world  
  $ |
## Java vs. C: Details

<table>
<thead>
<tr>
<th>Character type</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>char</td>
<td>char /* 8 bits */</td>
</tr>
<tr>
<td>Integral types</td>
<td>byte</td>
<td>(unsigned, signed) char</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>(unsigned, signed) short</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>(unsigned, signed) int</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>(unsigned, signed) long</td>
</tr>
<tr>
<td>Floating point types</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long double</td>
</tr>
<tr>
<td>Logical type</td>
<td>boolean</td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/* use 0 and non-0 */</td>
</tr>
<tr>
<td>Generic pointer type</td>
<td>Object</td>
<td>void*</td>
</tr>
<tr>
<td>Constants</td>
<td>final int MAX = 1000;</td>
<td>#define MAX 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const int MAX = 1000;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enum {MAX = 1000};</td>
</tr>
</tbody>
</table>
### Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrays</strong></td>
<td>int [] a = new int [10]; float [][] b =</td>
<td>int a[10]; float b[5][20];</td>
</tr>
<tr>
<td></td>
<td>new float [5][20];</td>
<td></td>
</tr>
<tr>
<td><strong>Array bound</strong></td>
<td>// run-time check</td>
<td>/* no run-time check */</td>
</tr>
<tr>
<td><strong>checking</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pointer type</strong></td>
<td>// Object reference is an // implicit</td>
<td>int *p;</td>
</tr>
<tr>
<td></td>
<td>pointer</td>
<td></td>
</tr>
<tr>
<td><strong>Record type</strong></td>
<td>class Mine</td>
<td>struct Mine</td>
</tr>
<tr>
<td></td>
<td>{ int x; float y; }</td>
<td>{ int x; float y; }</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>};</td>
</tr>
</tbody>
</table>
# Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
</table>
| **Strings**         | String s1 = "Hello";
                      | String s2 = new String("hello");                                    | char *s1 = "Hello";
                      |                                                            | char s2[6];                                                  |
|                     |                                                            | strcpy(s2, "hello");                                               |
| **String concatenation** | s1 + s2     | #include <string.h>                                                |
|                     | s1 += s2     | strcat(s1, s2);                                                    |
| **Logical ops**     | &&, ||, !     | &&, ||, !                                                          |
| **Relational ops**  | ==, !=, <, >, <=, => | ==, !=, <, >, <=, =>                                               |
| **Arithmetic ops**  | +, -, *, /, %, unary - | +, -, *, /, %, unary -                                             |
| **Bitwise ops**     | <<, >>, >>>, &, ^, | <<, >>, &, ^, |                                                   |
|                     | | ~                                                     | ~                                                                 |
| **Assignment ops**  | =, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |= | =, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |= |

* Essentially the same in the two languages*
### Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if stmt</strong></td>
<td><code>if (i &lt; 0) statement1;</code> <code>else statement2;</code></td>
<td><code>if (i &lt; 0) statement1;</code> <code>else statement2;</code></td>
</tr>
<tr>
<td><strong>switch stmt</strong></td>
<td><code>switch (i) {  case 1:    ...    break;    case 2:    ...    break;    default:    ...  }</code></td>
<td><code>switch (i) {  case 1:    ...    break;    case 2:    ...    break;    default:    ...  }</code></td>
</tr>
<tr>
<td><strong>goto stmt</strong></td>
<td>// no equivalent</td>
<td><code>goto someLabel;</code></td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages
## Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>for stmt</td>
<td><code>for (int i=0; i&lt;10; i++)</code></td>
<td><code>int i;</code></td>
</tr>
<tr>
<td></td>
<td><code>statement;</code></td>
<td><code>for (i=0; i&lt;10; i++)</code></td>
</tr>
<tr>
<td></td>
<td><code>statement;</code></td>
<td><code>statement;</code></td>
</tr>
<tr>
<td>while stmt *</td>
<td><code>while (i &lt; 0)</code></td>
<td><code>while (i &lt; 0)</code></td>
</tr>
<tr>
<td></td>
<td><code>statement;</code></td>
<td><code>statement;</code></td>
</tr>
<tr>
<td>do-while stmt *</td>
<td><code>do</code></td>
<td><code>do</code></td>
</tr>
<tr>
<td></td>
<td><code>statement;</code></td>
<td><code>statement;</code></td>
</tr>
<tr>
<td></td>
<td><code>while (i &lt; 0)</code></td>
<td><code>while (i &lt; 0);</code></td>
</tr>
<tr>
<td>continue stmt *</td>
<td><code>continue;</code></td>
<td><code>continue;</code></td>
</tr>
<tr>
<td>labeled continue stmt</td>
<td><code>continue someLabel;</code></td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>break stmt *</td>
<td><code>break;</code></td>
<td><code>break;</code></td>
</tr>
<tr>
<td>labeled break stmt</td>
<td><code>break someLabel;</code></td>
<td>/* no equivalent */</td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages*
## Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>return stmt *</td>
<td>return 5; return;</td>
<td>return 5; return;</td>
</tr>
<tr>
<td>Compound stmt (alias block) *</td>
<td>{ statement1; statement2; }</td>
<td>{ statement1; statement2; }</td>
</tr>
<tr>
<td>Exceptions</td>
<td>throw, try-catch-finally</td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>Comments</td>
<td>/* comment <em>/ /</em> comment */</td>
<td></td>
</tr>
<tr>
<td>Method / function call</td>
<td>f(x, y, z); someObject.f(x, y, z); SomeClass.f(x, y, z);</td>
<td>f(x, y, z);</td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages*
Agenda

Our computing environment
- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git

A taste of C
- History of C
- Building and running C programs
- Characteristics of C
- Example program: charcount
The charcount Program

Functionality:
• Read all characters from standard input stream
• Write to standard output stream the number of characters read
The program:

```c
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void) {
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF) {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```
Building and Running

```
$ gcc 217 charcount.c
$ ls
  ..  a.out
$ gcc 217 charcount.c -o charcount
$ ls
  ..  a.out
  charcount
$
```
charcount Building and Running

$ gcc -o charcount charcount.c
$ ./charcount
Line 1
Line 2
^D

What is this?
What is the effect?
What is printed?
$ gcc charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D
14
$

Includes visible characters plus two newlines
$ cat somefile
Line 1
Line 2
$ ./charcount < somefile
14
$
What is this?
What is the effect?
Running `charcount`

Run-time trace, referencing the original C code...

```
#include <stdio.h>

/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
  int c;
  int charCount = 0;
  c = getchar();
  while (c != EOF)
  {
    charCount++;
    c = getchar();
  }
  printf("%d\n", charCount);
  return 0;
}
```

Execution begins at `main()` function
• No classes in the C language.
Running charcount

Run-time trace, referencing the original C code...

```
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

We allocate space for `c` and `charCount` in the stack section of memory.

Why `int` instead of `char`?
Running charcount

Run-time trace, referencing the original C code...

charcount.c

```c
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF)
    {
    charCount++;
    c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

getchar() tries to read char from stdin
- Success ⇒ returns that char value (within an int)
- Failure ⇒ returns EOF

EOF is a special value, distinct from all possible chars
Running charcount

Run-time trace, referencing the original C code...

```c
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

Assuming \( c \neq EOF \), we increment `charCount`
Running charcount

Run-time trace, referencing the original C code...

charcount.c

```c
#include <stdio.h>
/* Write to stdout the number of
chars in stdin. Return 0. */
int main(void)
{
  int c;
  int charCount = 0;
  c = getchar();
  while (c != EOF)
  {
    charCount++;
    c = getchar();
  }
  printf("%d\n", charCount);
  return 0;
}
```

We call getchar() again and recheck loop condition
Running charcount

Run-time trace, referencing the original C code...

charcount.c

```c
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

- Eventually `getchar()` returns EOF
- Loop condition fails
- We call `printf()` to write final `charCount`
Running charcount

Run-time trace, referencing the original C code...

charcount.c

```c
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != EOF)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

- return statement returns to calling function
- return from main() returns to _start, terminates program

Normal execution ⇒ 0 or EXIT_SUCCESS
Abnormal execution ⇒ EXIT_FAILURE

#include <stdlib.h> to use these constants
Coming up next ...

More character processing, structured exactly how we'll want you to design your Assignment 1 solution!

Read the A1 specs soon: you'll be ready to start after Lecture 3!