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**
- 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)

`git commit`

`git checkout`
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* to multiple machines
- Any clone can *pull* the current state

`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 to github, then pull on 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:
Java vs. C: History


BCPL > B > C > K&R C > ANSI C89
ISO C90 > ISO/ANSI C99 > ISO C11
> ISO C18

C++ > Java

Algon
Simula
LISP
Smalltalk

This is what we’re using
## C vs. Java: Design Goals

<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

Java compiler (machine lang code)
Running Java Programs

$ java MyProg

Java interpreter / "virtual machine" (machine lang code)
Building C Programs

$ gcc217 myprog.c –o myprog

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

$ ./myprog

HW (ArmLab)

OS (Linux)

myprog

(machine lang code)

data

myprog

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>
<thead>
<tr>
<th>Program</th>
<th>Code Type</th>
<th>Portable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyProgs.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>MyProgs.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
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>
<tbody>
<tr>
<td><strong>Hello.java:</strong></td>
<td><strong>public class Hello</strong></td>
<td><strong>hello.c:</strong></td>
</tr>
<tr>
<td></td>
<td><strong>{ public static void main</strong></td>
<td><strong>#include &lt;stdio.h&gt;</strong></td>
</tr>
<tr>
<td></td>
<td><strong>(String[] args)</strong></td>
<td><strong>int main(void)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>{ System.out.println(</strong></td>
<td><strong>{ printf(&quot;hello, world\n&quot;);</strong></td>
</tr>
<tr>
<td></td>
<td><strong>&quot;hello, world&quot;);</strong></td>
<td><strong>return 0;</strong></td>
</tr>
<tr>
<td></td>
<td><strong>}</strong></td>
<td><strong>}</strong></td>
</tr>
</tbody>
</table>

### Building

- `$ javac Hello.java`
- `$ gcc217 hello.c –o hello`

### Running

- `$ java Hello`
- `hello, world`
- `$`
## Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th><strong>Java</strong></th>
<th><strong>C</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Character type</strong></td>
<td>char // 16-bit Unicode</td>
<td>char /* 8 bits */</td>
</tr>
<tr>
<td><strong>Integral types</strong></td>
<td>byte // 8 bits</td>
<td>(unsigned, signed) char</td>
</tr>
<tr>
<td></td>
<td>short // 16 bits</td>
<td>(unsigned, signed) short</td>
</tr>
<tr>
<td></td>
<td>int // 32 bits</td>
<td>(unsigned, signed) int</td>
</tr>
<tr>
<td></td>
<td>long // 64 bits</td>
<td>(unsigned, signed) long</td>
</tr>
<tr>
<td><strong>Floating point types</strong></td>
<td>float // 32 bits</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>double // 64 bits</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long double</td>
</tr>
<tr>
<td><strong>Logical type</strong></td>
<td>boolean</td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/* use 0 and non-0 */</td>
</tr>
<tr>
<td><strong>Generic pointer type</strong></td>
<td>Object</td>
<td>void*</td>
</tr>
<tr>
<td><strong>Constants</strong></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 = new float [5][20];</td>
<td>int a[10];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>float b[5][20];</td>
</tr>
<tr>
<td>**Array bound</td>
<td>// run-time check</td>
<td>/* no run-time check */</td>
</tr>
<tr>
<td>checking**</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pointer type</strong></td>
<td>// Object reference is an implicit pointer</td>
<td>int *p;</td>
</tr>
<tr>
<td><strong>Record type</strong></td>
<td>class Mine { int x; float y; }</td>
<td>struct Mine { 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>
<tbody>
<tr>
<td>Strings</td>
<td>String s1 = &quot;Hello&quot;;</td>
<td>char *s1 = &quot;Hello&quot;;</td>
</tr>
<tr>
<td></td>
<td>String s2 = new String(&quot;hello&quot;);</td>
<td>char s2[6];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strcpy(s2, &quot;hello&quot;);</td>
</tr>
<tr>
<td>String concatenation</td>
<td>s1 + s2</td>
<td>#include &lt;string.h&gt;</td>
</tr>
<tr>
<td></td>
<td>s1 += s2</td>
<td>strcat(s1, s2);</td>
</tr>
<tr>
<td>Logical ops *</td>
<td>&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>Relational ops *</td>
<td>==, !=, &lt;, &gt;, &lt;=, &gt;</td>
<td>==, !=, &lt;, &gt;, &lt;=, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td>Arithmetic ops *</td>
<td>+, -, *, /, %, unary -</td>
<td>+, -, *, /, %, unary -</td>
</tr>
<tr>
<td>Bitwise ops</td>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;, &amp;, ^,</td>
<td>, ~</td>
</tr>
<tr>
<td>Assignment ops</td>
<td>=, +=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, &amp;=, ^=,</td>
<td>=</td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages*
<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>if stmt *</td>
<td>if (i &lt; 0)</td>
</tr>
<tr>
<td></td>
<td>statement1;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>statement2;</td>
</tr>
<tr>
<td>switch stmt *</td>
<td>switch (i)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>case 1:</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td>case 2:</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td>default:</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>goto stmt</td>
<td>// no equivalent</td>
</tr>
<tr>
<td></td>
<td>goto someLabel;</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;&quot; 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</td>
<td><code>continue someLabel;</code></td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>stmt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>break stmt</td>
<td><code>break;</code></td>
<td><code>break;</code></td>
</tr>
<tr>
<td>labeled break</td>
<td><code>break someLabel;</code></td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>stmt</td>
<td></td>
<td></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><code>return 5;</code> <code>return;</code></td>
<td><code>return 5;</code> <code>return;</code></td>
</tr>
<tr>
<td>Compound stmt (alias block) *</td>
<td><code>{ </code>statement1;<code> </code>statement2;<code> </code>}`</td>
<td><code>{ </code>statement1;<code> </code>statement2;<code> </code>}`</td>
</tr>
<tr>
<td>Exceptions</td>
<td><code>throw</code>, <code>try-catch-finally</code></td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>Comments</td>
<td>/* comment */ <code>// another kind</code></td>
<td>/* comment */</td>
</tr>
<tr>
<td>Method / function call</td>
<td><code>f(x, y, z);</code> <code>someObject.f(x, y, z);</code> <code>SomeClass.f(x, y, z);</code></td>
<td><code>f(x, y, z);</code></td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages*
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
Functionality:

- Read all characters from standard input stream
- Write to standard output stream the number of characters read
The charcount Program

The program:  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;
}
```

$ gcc217 charcount.c
$ ls
  ..  a.out
$ gcc217 charcount.c -o charcount
$ ls
  ..  a.out
  charcount
$
$ gcc charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D

What is this?
What is the effect?
What is printed?
Building and Running

$ gcc charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D
14
$

Includes visible characters plus two newlines
What is this?
What is the effect?
charcount Building and Running

$ ./charcount > someotherfile
Line 1
Line 2
^D
$ cat someotherfile
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...

#### 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 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`
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;
}
```

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...

```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() terminates program

Normal execution ⇒ 0 or EXIT_SUCCESS
Abnormal execution ⇒ EXIT_FAILURE
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 Wednesday!