A Taste of C
Agenda

Getting started with C
- History of C
- Building and running C programs
- Characteristics of C

Three Simple C Programs
- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details
- For initial cram and/or later reference
## The C Programming Language

<table>
<thead>
<tr>
<th><strong>Who?</strong></th>
<th>Dennis Ritchie</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When?</strong></td>
<td>~1972</td>
</tr>
<tr>
<td><strong>Where?</strong></td>
<td>Bell Labs</td>
</tr>
<tr>
<td><strong>Why?</strong></td>
<td>Build the Unix OS</td>
</tr>
</tbody>
</table>
Java vs. C: History

- **1960**: BCPL
- **1970**: B
- **1972**: C
- **1978**: K&R C
- **1989**: ANSI C89, ISO C90
- **1999**: ISO C99, ANSI C99
- **2011**: ISO C11
- **2018**: ISO C18

This is what we’re using

- **1960**: Algol
- **1970**: Simula
- **1970**: LISP
- **1972**: Smalltalk
- **1978**: C++
- **1989**: Java
- **(many evolutions)**
## 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>
Getting started with C
  • History of C
  • Building and running C programs
  • Characteristics of C

Three Simple C Programs
  • charcount (loops, standard input)
    • 4-stage build process
  • upper (character data, ctype library)
    • portability concerns
  • upper1 (switch statements, enums, functions)
    • DFA program design

Java versus C Details
  • For initial cram and/or later reference
Building Java Programs

$ javac MyProg.java

Java compiler
(machine lang code)

HW (ArmLab)

OS (Linux)

MyProg.java
(Java code)

javac

MyProg.class
(bytecode)
Running Java Programs

$ java MyProg

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

HW (ArmLab)

OS (Linux)

MyProg.class (bytecode)

data

java

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)

data

myprog

data
Agenda

Getting started with C
  • History of C
  • Building and running C programs
  • **Characteristics of C**

Three Simple C Programs
  • charcount (loops, standard input)
    • 4-stage build process
  • upper (character data, ctype library)
    • portability concerns
  • upper1 (switch statements, enums, functions)
    • DFA program design

Java versus C Details
  • For initial cram and/or later reference
## Java vs. C: Portability

<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

Example: since I’ve been here, we’ve used three architectures (x86, x86_64, and AArch64) and all our programs ... class samples, assignment reference implementations, grading infrastructure, etc. had to be recompiled with each change!
Java vs. C: Safety & Efficiency

Java

- Automatic array-bounds checking,
- NULL pointer checking,
- Automatic memory management (garbage collection)
- Other safety features

C

- Manual bounds checking
- NULL pointer 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.
Goals of the rest of this Lecture

Help you learn about:
- The basics of C
- Deterministic finite-state automata (DFA)
- Expectations for programming assignments

Why?
- Help you get started with Assignment 1
  - Required readings…
  - + coverage of programming environment in precepts…
  - + minimal coverage of C in this lecture…
  - = enough info to start Assignment 1
- DFAs are useful in many contexts
  - Theoretical problem characteristics + modeling
  - Practical system/program design (e.g. A1)
Agenda

Getting started with C
  • History of C
  • Building and running C programs
  • Characteristics of C

Three Simple C Programs
  • charcount (loops, standard input)
    • 4-stage build process
  • upper (character data, ctype library)
    • portability concerns
  • upper1 (switch statements, enums, functions)
    • DFA program design

Java versus C Details
  • For initial cram and/or later reference
The “charcount” Program

Functionality:
• Read all characters from standard input stream
• Write to standard output stream the number of characters read
Q: What is the output of `charcount` on this input?

A. 10
B. 12
C. 13
D. 14
E. 15
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;
}
```
Running “charcount”

Run-time trace, referencing the original C code…

ccharcount.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;
}
```

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

```
#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…

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;
}
```

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() terminates program

Normal execution ⇒ 0 or EXIT_SUCCESS
Abnormal execution ⇒ EXIT_FAILURE
"charcount" Building and Running

```
$ gcc217 charcount.c
$ ls
  ..  a.out
$ gcc217 charcount.c -o charcount
$ ls
  ..  a.out  charcount
$  
```
“charcount” Building and Running

```
$ gcc217 charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D
14
$
```

What is this?
What is the effect?
building and running

$ cat somefile
Line 1
Line 2
$ ./charcount < somefile
14
$

What is this?
What is the effect?
```
$. /charcount > someotherfile
Line 1
Line 2
^D
$. cat someotherfile
14
$. 
```

What is this? What is the effect?
“charcount” Build Process in Detail

Question:
• Exactly what happens when you issue the command
  gcc217 charcount.c -o charcount

Answer: Four steps
• Preprocess
• Compile
• Assemble
• Link
“charcount” Build Process in Detail

The starting point

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;
}
```

- C language
- Missing declarations of `getchar()` and `printf()`
- Missing definitions of `getchar()` and `printf()`
Preprocessing “charcount”

Command to preprocess:
• gcc217 -E charcount.c > charcount.i

Preprocessor functionality
• Removes comments
• Handles preprocessor directives
Preprocessing "charcount"

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;
}
```

Preprocessor removes comment (this is A1!)
Preprocessing “charcount”

charcount.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;
}
```

Preprocessor replaces
#include <stdio.h>
with contents of
/usr/include/stdio.h

Preprocessor replaces
EOF with -1
The result

```
... int getchar();
int printf(char *fmt, ...);
...

int main(void)
{
  int c;
  int charCount = 0;
  c = getchar();
  while (c != -1)
  {
    charCount++;
    c = getchar();
  }
  printf("%d
", charCount);
  return 0;
}
```

- C language
- Missing comments
- Missing preprocessor directives
- Contains code from stdio.h: declarations of `getchar()` and `printf()`
- Missing definitions of `getchar()` and `printf()`
- Contains value for EOF
Compiling “charcount”

Command to compile:
• gcc217 –S charcount.i

Compiler functionality
• Translate from C to assembly language
• Use function declarations to check calls of getchar() and printf()
Compiling “charcount”

charcount.i

...  
int getchar();  
int printf(char *fmt, ...);  
...  
int main(void)  
{  
  int c;  
  int charCount = 0;  
  c = getchar();  
  while (c != -1)  
  {  
    charCount++;  
    c = getchar();  
  }  
  printf("%d\n", charCount);  
  return 0;  
}
Compiling “charcount”

charcount.i

... 
int getchar();
int printf(char *fmt, ...);
...
int main(void)
{
    int c;
    int charCount = 0;
    c = getchar();
    while (c != -1)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}

• Definition of main() function
• Compiler checks calls of getchar() and printf() when encountered
• Compiler translates to assembly language
Compiling “charcount”

The result: charcount.s

```
.section .rodata
.LC0:
.string "%d\n"

.section .text
.global main
main:
    stp x29, x30, [sp, -32]!
    add x29, sp, 0
    str wzr, [x29,24]
    bl getchar
    str w0, [x29,28]
    b .L2
.L3:
    ldr w0, [x29,24]
    add w0, w0, 1
    str w0, [x29,24]
    bl getchar
    str w0, [x29,28]
.L2:
    ldr w0, [x29,28]
    cmn w0, #1
    bne .L3
    adrp x0, .LC0
    add x0, x0, :lo12:.LC0
    ldr w1, [x29,24]
    bl printf
    mov w0, 0
    ldp x29, x30, [sp], 32
    ret
```

- Assembly language
- Missing definitions of getchar() and printf()
Assembling “charcount”

Command to assemble:
• gcc217 –c charcount.s

Assembler functionality
• Translate from assembly language to machine language
Assembling “charcount”

The result:

charcount.o

- Machine language
- Missing definitions of getchar() and printf()
Linking “charcount”

Command to link:
• `gcc217 charcount.o -o charcount`

Linker functionality
• Resolve references within the code
• Fetch machine language code from the standard C library (/usr/lib/libc.a) to make the program complete
Linking “charcount”

The result:

charcount

- Machine language
- Contains definitions of getchar() and printf()

Complete! Executable!
iClicker Question

Q: There are other ways to char count – which is best?

A. ```
for (c=getchar(); c!=EOF; c=getchar())
    charCount++;
```  

B. ```
while ((c=getchar()) != EOF)
    charCount++;
```  

C. ```
for (;;)
    { c = getchar();
        if (c == EOF)
            break;
        charCount++;
    }
```  

D. ```
c = getchar();
while (c!=EOF)
    { charCount++;
        c =
        getchar();
    }
```
Example 2: “upper”

Functionality
- Read all chars from stdin
- Convert each lower-case alphabetic char to upper case
  - Leave other kinds of chars alone
- Write result to stdout

stdin
Does this work?
It seems to work.

upper

stdout
DOES THIS WORK?
IT SEEMS TO WORK.
# ASCII

## American Standard Code for Information Interchange

![Partial map]

### Note:
Lower-case and upper-case letters are 32 apart
#include <stdio.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if ((c >= 97) && (c <= 122))
            c -= 32;
        putchar(c);
    }
    return 0;
}
Character Literals

Examples

'a'  the a character
     97 on ASCII systems

'\n'  newline
     10 on ASCII systems

'\t'  horizontal tab
     9 on ASCII systems

'\"' backslash
     92 on ASCII systems

'\'' single quote
     39 on ASCII systems

'\0'  the null character (alias NUL)
     0 on all systems
#include <stdio.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if ((c >= 'a') && (c <= 'z'))
            c += 'A' - 'a';
        putchar(c);
    }
    return 0;
}
ctype.h Functions

$ man islower
NAME
  isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit – character classification routines

SYNOPSIS
  #include <ctype.h>
  int isalnum(int c);
  int isalpha(int c);
  int isascii(int c);
  int isblank(int c);
  int iscntrl(int c);
  int isdigit(int c);
  int isgraph(int c);
  int islower(int c);
  int isprint(int c);
  int ispunct(int c);
  int isspace(int c);
  int isupper(int c);
  int isxdigit(int c);
  These functions check whether c falls into various character classes
$ man toupper

NAME
toupper, tolower - convert letter to upper or lower case

SYNOPSIS
#include <ctype.h>
int toupper(int c);
int tolower(int c);

DESCRIPTION
toupper() converts the letter c to upper case, if possible.
tolower() converts the letter c to lower case, if possible.

If c is not an unsigned char value, or EOF, the behavior of these functions is undefined.

RETURN VALUE
The value returned is that of the converted letter or c if the conversion was not possible.
```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int c;
    while (((c = getchar()) != EOF))
    {
        if (islower(c))
        {
            c = toupper(c);
            putchar(c);
        }
        return 0;
    }
}
```
Q: Is the `if` statement really necessary?

A. Gee, I don’t know. Let me check the man page (again)!

```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  int c;
  while ((c = getchar()) != EOF)
  {
    if (islower(c))
    {
      c = toupper(c);
      putchar(c);
    }
    putchar(c);
  }
  return 0;
}
```
$ man toupper

NAME

toupper, tolower - convert letter to upper or lower case

SYNOPSIS

#include <ctype.h>
int toupper(int c);
int tolower(int c);

DESCRIPTION

toupper() converts the letter c to upper case, if possible.
tolower() converts the letter c to lower case, if possible.

If c is not an unsigned char value, or EOF, the behavior of
these functions is undefined.

RETURN VALUE

The value returned is that of the converted letter
or c if the conversion was not possible.
iClicker Question

Q: Is the if statement really necessary?

A. Yes, necessary for correctness.

B. Not necessary, but I’d leave it in.

C. Not necessary, and I’d get rid of it.

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if (islower(c))
        {
            c = toupper(c);
            putchar(c);
        }
        putchar(c);
    }
    return 0;
}
```
Agenda

Getting started with C
- History of C
- Building and running C programs
- Characteristics of C

Three Simple C Programs
- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details
- For initial cram and/or later reference
Example 3: “upper1”

Functionality
- Read all chars from stdin
- Capitalize the first letter of each word
  - “cos 217 rocks” ⇒ “Cos 217 Rocks”
- Write result to stdout

stdin

```
cos 217 rocks
Does this work?
It seems to work.
```

upper1

```
Cos 217 Rocks
 Does This Work?
It Seems To Work.
```
“upper1” Challenge

Problem

• Must remember where you are
• Capitalize “c” in “cos”, but not “o” in “cos” or “c” in “rocks”

Solution

• Maintain some extra information
• “In a word” vs “not in a word”
Deterministic Finite Automaton (DFA)

- States, one of which denotes the start
- Transitions labeled by chars or categories
- Optionally, actions on transitions

isalpha
(print uppercase equiv)

!isalpha
(print)
```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    int state = 0;
    while ((c = getchar()) != EOF)
    {
        switch (state)
        {
            case 0:
                if (isalpha(c))
                {
                    putchar(toupper(c)); state = 1; }
                else
                {
                    putchar(c); state = 0; }
                break;
            case 1:
                if (isalpha(c))
                {
                    putchar(c); state = 1; }
                else
                {
                    putchar(c); state = 0; }
                break;
        }
    }
    return 0;
}
```

That’s a B. What’s wrong?
Problem:
- The program works, but…
- States should have names

Solution:
- Define your own named constants
  - `enum Statetype {NORMAL, INWORD};`
  - Define an enumeration type
  - `enum Statetype state;`
  - Define a variable of that type
```c
#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};

int main(void)
{
    int c;
    enum Statetype state = NORMAL;
    while ((c = getchar()) != EOF)
    {
        switch (state)
        {
        case NORMAL:
            if (isalpha(c))
            {
                putchar(toupper(c)); state = INWORD;
            }
            else
            {
                putchar(c); state = NORMAL;
            }
            break;
        case INWORD:
            if (isalpha(c))
            {
                putchar(c); state = INWORD;
            }
            else
            {
                putchar(c); state = NORMAL;
            }
            break;
        }
    }
    return 0;
}
```

That's a B+. What's wrong?
Problem:
- The program works, but…
- Deeply nested statements
- No modularity

Solution:
- Handle each state in a separate function
`#include <stdio.h>`
`#include <ctype.h>`

```c
enum Statetype {NORMAL, INWORD};

enum Statetype handleNormalState(int c)
{
    enum Statetype state;
    if (isalpha(c))
    {
        putchar(toupper(c));
        state = INWORD;
    }
    else
    {
        putchar(c);
        state = NORMAL;
    }
    return state;
}

enum Statetype handleInwordState(int c)
{
    enum Statetype state;
    if (!isalpha(c))
    {
        putchar(c);
        state = NORMAL;
    }
    else
    {
        putchar(c);
        state = INWORD;
    }
    return state;
}

int main(void)
{
    int c;
    enum Statetype state = NORMAL;
    while ((c = getchar()) != EOF)
    {
        switch (state)
        {
            case NORMAL:
            {
                state = handleNormalState(c);
                break;
            }
            case INWORD:
            {
                state = handleInwordState(c);
                break;
            }
        }
    }
    return 0;
}
```

That’s an A-.
What’s wrong?
Problem:
  • The program works, but…
  • No comments

Solution:
  • Add (at least) function-level comments
Function Comments

Function comment should describe

*what the function does* (from the caller’s viewpoint)

- Input to the function
  - Parameters, input streams
- Output from the function
  - Return value, output streams, (call-by-reference parameters)

Function comment should not describe

*how the function works*
**Bad** main() function comment

Read a character from stdin. Depending upon the current DFA state, pass the character to an appropriate state-handling function. The value returned by the state-handling function is the next DFA state. Repeat until end-of-file.

Describes **how the function works**

**Good** main() function comment

Read text from stdin. Convert the first character of each "word" to uppercase, where a word is a sequence of characters. Write the result to stdout. Return 0.

Describes **what the function does**
(from caller’s viewpoint)
#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};
/* Implement the NORMAL state of the DFA. c is the current DFA character. Write c or its uppercase equivalent to stdout, as specified by the DFA. Return the next state. */

enum Statetype handleNormalState(int c)
{
    enum Statetype state;
    if (isalpha(c))
    {
        putchar(toupper(c));
        state = INWORD;
    }
    else
    {
        putchar(c);
        state = NORMAL;
    }
    return state;
}
/* Implement the INWORD state of the DFA. c is the current DFA character. Write c to stdout, as specified by the DFA. Return the next state. */

enum Statetype handleInwordState(int c)
{
    enum Statetype state;
    if (!isalpha(c))
    {
        putchar(c);
        state = NORMAL;
    }
    else
    {
        putchar(c);
        state = INWORD;
    }
    return state;
}
int main(void)
{
    int c;
    /* Use a DFA approach. state indicates the DFA state. */
    enum Statetype state = NORMAL;
    while ((c = getchar()) != EOF)
    {
        switch (state)
        {
        case NORMAL:
            state = handleNormalState(c);
            break;
        case INWORD:
            state = handleInwordState(c);
            break;
        }
    }
    return 0;
}
Review of Example 3

Deterministic finite-state automaton
- Two or more states
- Transitions between states
  - Next state is a function of current state and current character
  - Actions can occur during transitions

Expectations for COS 217 assignments
- Readable
  - Meaningful names for variables, constants, and literals
  - Reasonable max nesting depth
- Modular
  - Multiple functions, each with 1 well-defined job
- Function-level comments
  - Should describe what function does
- See K&P book for style guidelines specification
Agenda

Getting started with C
  • History of C
  • Building and running C programs
  • Characteristics of C

Three Simple C Programs
  • charcount (loops, standard input)
    • 4-stage build process
  • upper (character data, ctype library)
    • portability concerns
  • upper1 (switch statements, enums, functions)
    • DFA program design

Java versus C Details
  • For initial cram and/or later reference
## 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></td>
<td></td>
</tr>
<tr>
<td><code>public class Hello</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>public static void main</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>String[] args</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`System.out.println(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;hello, world&quot;)`</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Building**

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

**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 // 16-bit Unicode</td>
<td>char /* 8 bits */</td>
</tr>
<tr>
<td>Integral types</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>Floating point types</td>
<td>float // 32 bits</td>
<td>float double</td>
</tr>
<tr>
<td></td>
<td>double // 64 bits</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>const int MAX = 1000;</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]; float b[5][20];</td>
</tr>
<tr>
<td><strong>Array bound checking</strong></td>
<td>// run-time check</td>
<td>/* no run-time check */</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>
</tbody>
</table>
# Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strings</strong></td>
<td>String s1 = &quot;Hello&quot;; String s2 = new String(&quot;hello&quot;);</td>
<td>char *s1 = &quot;Hello&quot;; char s2[6]; strcpy(s2, &quot;hello&quot;);</td>
</tr>
<tr>
<td><strong>String concatenation</strong></td>
<td>s1 + s2</td>
<td>#include &lt;string.h&gt; strcat(s1, s2);</td>
</tr>
<tr>
<td></td>
<td>s1 += s2</td>
<td></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;=</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
# Java vs. C: Details

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>if stmt *</td>
<td><code>if (i &lt; 0)</code></td>
<td><code>if (i &lt; 0)</code></td>
</tr>
<tr>
<td></td>
<td><code>statement1;</code></td>
<td><code>statement1;</code></td>
</tr>
<tr>
<td></td>
<td><code>else</code></td>
<td><code>else</code></td>
</tr>
<tr>
<td></td>
<td><code>statement2;</code></td>
<td><code>statement2;</code></td>
</tr>
<tr>
<td>switch stmt *</td>
<td><code>switch (i)</code></td>
<td><code>switch (i)</code></td>
</tr>
<tr>
<td></td>
<td><code>{ case 1:</code></td>
<td><code>{ case 1:</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>break;</code></td>
<td><code>break;</code></td>
</tr>
<tr>
<td></td>
<td><code>case 2:</code></td>
<td><code>case 2:</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>break;</code></td>
<td><code>break;</code></td>
</tr>
<tr>
<td></td>
<td><code>default:</code></td>
<td><code>default:</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td>goto stmt</td>
<td><code>// no equivalent</code></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>Statement Type</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
</table>
| for stmt       | `for (int i=0; i<10; i++)
statement;` | `int i;
for (i=0; i<10; i++)
statement;` |
| while stmt *   | `while (i < 0)
statement;`            | `while (i < 0)
statement;`            |
| do-while stmt *| `do
statement;
while (i < 0)` | `do
statement;
while (i < 0);` |
| continue stmt *| `continue;`                       | `continue;`                       |
| labeled continue stmt | `continue someLabel;` | /* no equivalent */ |
| break stmt *   | `break;`                              | `break;`                              |
| labeled break stmt | `break someLabel;`                  | /* no equivalent */                  |

* 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;</td>
<td>return 5;</td>
</tr>
<tr>
<td></td>
<td>return;</td>
<td>return;</td>
</tr>
<tr>
<td>Compound stmt</td>
<td><code>{ statement1; statement2; }</code></td>
<td><code>{ statement1; statement2; }</code></td>
</tr>
<tr>
<td>(aka: block) *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>throw, try-catch-finally</td>
<td>/* no equivalent */</td>
</tr>
<tr>
<td>Comments</td>
<td>/* comment */</td>
<td>/* comment */</td>
</tr>
<tr>
<td></td>
<td>// another kind</td>
<td></td>
</tr>
<tr>
<td>Method / function call</td>
<td>f(x, y, z);</td>
<td>f(x, y, z);</td>
</tr>
<tr>
<td></td>
<td>someObject.f(x, y, z);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SomeClass.f(x, y, z);</td>
<td></td>
</tr>
</tbody>
</table>

* Essentially the same in the two languages*
Example C Program

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    const double KMETERS_PER_MILE = 1.609;
    int miles;
    double kMeters;

    printf("miles: ");
    if (scanf("%d", &miles) != 1)
    {
        fprintf(stderr, "Error: Expected a number.\n");
        exit(EXIT_FAILURE);
    }

    kMeters = (double)miles * KMETERS_PER_MILE;
    printf("%d miles is %f kilometers.\n", miles, kMeters);
    return 0;
}
```