Chapter 11
Introduction to Programming in C

The Course Thus Far...

We did digital logic
- Bits are bits
- Ultimately, to understand a simple processor

We did assembly language programming
- Programming the “raw metal” of the computer
- Ultimately, to understand C programming

Starting today: we’re doing C programming
- C is still common for systems programming
- You’ll need it for the operating systems class (CSE380)
- Ultimately, for a deeper understanding of any language (Java)

Why High-Level Languages?

Easier than assembly. Why?
- Less primitive constructs
- Variables
- Type checking

Portability
- Write program once, run it on the LC-3 or Intel’s x86

Disadvantages
- Slower and larger programs (in most cases)
- Can’t manipulate low-level hardware
  ➢ All operating systems have some assembly in them

Verdict: assembly coding is rare today
Our Challenge

All of you already know Java
- We're going to try to cover the basics quickly
- We'll spend more time on pointers & other C-specific nastiness

Created two decades apart
- C: 1970s - AT&T Bell Labs
- C++: 1980s - AT&T Bell Labs
- Java: 1990s - Sun Microsystems

Java and C/C++
- Syntactically similar (Java uses C syntax)
- C lacks many of Java's features
- Subtly different semantics

More C vs Java differences

C has a “preprocessor”
- A separate pre-pass over the code
- Performs replacements

Include vs Import
- Java has import java.io.*;
- C has: #include <stdio.h>
- #include is part of the preprocessor

Boolean type
- Java has an explicit boolean type
- C just uses an “int” as zero or non-zero
- C’s lack of boolean causes all sorts of trouble

More differences as we go along...

C is Similar To Java Without:

Objects
- No classes, objects, methods, or inheritance

Exceptions
- Check all error codes explicitly

Standard class library
- C has only a small standard library

Garbage collection
- C requires explicit memory allocate and free

Safety
- Java has strong type checking, checks array bounds
- In C, anything goes

Portability
- Source: C code is less portable (but better than assembly)
- Binary: C compiles to specific machine code

History of C and Unix

Unix is the most influential operating system

First developed in 1969 at AT&T Bell Labs
- By Ken Thompson and Dennis Ritchie
- Designed for “smaller” computers of the day
- Reject some of the complexity of MIT’s Multics

They found writing in assembly tedious
- Dennis Ritchie invented the C language in 1973
- Based on BCPL and B, needed to be efficient (24KB of memory)

Unix introduced to UC-Berkeley (Cal) in 1974
- Bill Joy was an early Unix hacker as a PhD student at Cal
- Much of the early internet consisted of Unix systems Mid-80s
- Good, solid TCP/IP for BSD in 1984

Linux - Free (re)implementation of Unix (libre and gratuit)
- Announced by Linus Torvalds in 1991

Much more in CSE380!
Aside: The Unix Command Line

Text-based approach to give commands
- Commonly used before graphical displays
- Many advantages even today

Examples
- mkdir cse240hw8  make a directory
- cd cse240hw8  change to the directory
- ls  list contents of directory
- cp /mnt/eniac/home1/c/cse240/project/hw/hw8/* .
  ➢ Copy files from one location to current dir ("")
- emacs foo.c &  run the command "emacs" with input "foo.c"
- gcc -o foo foo.c  compile foo.c (create program called "foo")

Unix eventually developed graphical UIs (GUIs)
- X-windows (long before Microsoft Windows)

Quotes on C/C++ vs Java

“C is to assembly language as Java is to C”
- Unknown

"With all due respect, saying Java is just a C++ subset is rather like saying that 'Pride and Prejudice' is just a subset of the Encyclopedia Britanica. While it is true that one is shorter than the other, and that both have the same syntax, there are rather overwhelming differences.”
- Sam Weber, on the ACM SIGSCE mailing list

“Java is C++ done right.”
- Unknown

What is C++?

C++ is an extension of C
- Also done at AT&T Bell Labs (1983)
- Backward compatible (good and bad)
- That is, all C programs are legal C++ programs

C++ adds many features to C
- Classes, objects, inheritance
- Templates for polymorphism
- A large, cumbersome class library (using templates)
- Exceptions (not actually implemented for a long time)
- More safety (though still unsafe)
- Operator and function overloading
- Kitchen sink

Thus, many people use it (to some extent)
- However, we’re focusing on only C, not C++

More quotes on C/C++

"The C programming language combines the power of assembly language with the ease-of-use of assembly language.”
- Unknown

"It is my impression that it's possible to write good programs in C++, but nobody does.”
- John Levine, moderator of comp.compilers

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it, it blows your whole leg off.”
- Bjarne Stroustrup, creator of C++
Program Execution: Compilation vs Interpretation
Different ways of executing high-level languages

Interpretation
- Interpreter: program that executes program statements
  - Directly interprets program (portable but slow)
  - Limited optimization
- Easy to debug, make changes, view intermediate results
- Languages: BASIC, LISP, Perl, Python, Matlab

Compilation
- Compiler: translates statements into machine language
  - Creates executable program (non-portable, but fast)
  - Performs optimization over multiple statements
- Harder to debug, change requires recompilation
- Languages: C, C++, Fortran, Pascal

Hybrid
- Java, has features of both interpreted and compiled languages

Compilation vs. Interpretation
Consider the following algorithm:
- Get W from the keyboard.
- X = W + W
- Y = X + X
- Z = Y + Y
- Print Z to screen.

If interpreting, how many arithmetic operations occur?

If compiling, we can analyze the entire program and possibly reduce the number of operations.
- Can we simplify the above algorithm to use a single arithmetic operation?

Compiling a C Program
Entire mechanism is usually called the “compiler”

Preprocessor
- Macro substitution
- Conditional compilation
- “Source-level” transformations
  - Output is still C

Compiler
- Generates object file
  - Machine instructions

Linker
- Combine object files (including libraries)
  - into executable image

Compiler
Source Code Analysis
- “Front end”
  - Parses programs to identify its pieces
    - Variables, expressions, statements, functions, etc.
  - Depends on language (not on target machine)

Code Generation
- “Back end”
  - Generates machine code from analyzed source
  - May optimize machine code to make it run more efficiently
  - Very dependent on target machine

Example Compiler: GCC
- The Free-Software Foundation’s compiler
- Many front ends: C, C++, Fortran, Java
- Many back ends: Intel x86, PowerPC, SPARC, MIPS, Itanium
A Simple C Program

```c
#include <stdio.h>
define STOP 0

void main()
{
    /* variable declarations */
    int counter; /* an integer to hold count values */
    int startPoint; /* starting point for countdown */

    /* prompt user for input */
    printf("Enter a positive number: ");
    scanf("%d", &startPoint); /* read into startPoint */

    /* count down and print count */
    for (counter=startPoint; counter >= STOP; counter--)
    {
        printf("%d\n", counter);
    }
}
```

Preprocessor Directives

```c
#include <stdio.h>
#define STOP 0
```

- Before compiling, copy contents of header file (stdio.h) into source code.
- Header files typically contain descriptions of functions and variables needed by the program.
  - no restrictions -- could be any C source code

```c
#define STOP 0
```

- Before compiling, replace all instances of the string "STOP" with the string "0"
- Called a macro
- Used for values that won’t change during execution, but might change if the program is reused. (Must recompile.)

Comments

- Begins with /* and ends with */
  - Can span multiple lines
  - Comments are not recognized within a string
    - example: "my/"don't print this*/string"
      would be printed as: my/"don't print this*/string

- Begins with // and ends with “end of line”
  - Single-line comment
  - Much like ";" in LC-3 assembly
  - Introduced in C++, later back-ported to C

main Function

Every C program must have a function called main()

- Starting point for every program
- Similar to Java’s main method
  - public static void main(String[] args)

The code for the function lives within brackets:
```c
void main()
{
    /* code goes here */
}
```
Variable Declarations

Variables are used as names for data items

Each variable has a type, tells the compiler:

- How the data is to be interpreted
- How much space it needs, etc.

```c
int counter;
int startPoint;
```

C has similar primitive types as Java

- int, char, long, float, double
- More later

More About Output

Can print arbitrary expressions, not just variables

```c
printf("%d\n", startPoint - counter);
```

Print multiple expressions with a single statement

```c
printf("%d %d\n", counter,
        startPoint - counter);
```

Different formatting options:

- %d decimal integer
- %x hexadecimal integer
- %c ASCII character
- %f floating-point number

Input and Output

Variety of I/O functions in C Standard Library

- Must include <stdio.h> to use them

```c
printf("%d\n", counter);
```

- String contains characters to print and formatting directions for variables
- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)

```c
scanf("%d", &startPoint);
```

- String contains formatting directions for looking at input
- This call says to read a decimal integer and assign it to the variable startPoint (Don't worry about the & yet)

Examples

This code:

```c
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

produces this output:

```
43 is a prime number.
43 plus 59 in decimal is 102.
43 plus 59 in hex is 66.
43 plus 59 as a character is f.
```
Examples of Input

Many of the same formatting characters are available for user input

```c
scanf("%c", &nextChar);
```
- reads a single character and stores it in nextChar

```c
scanf("%f", &radius);
```
- reads a floating point number and stores it in radius

```c
scanf("%d %d", &length, &width);
```
- reads two decimal integers (separated by whitespace), stores the first one in length and the second in width

Must use ampersand (&) for variables being modified
(Explained in Chapter 16.)

Compiling and Linking

Various compilers available
- `cc`, `gcc`
  - includes preprocessor, compiler, and linker

Lots and lots of options!
- level of optimization, debugging
- preprocessor, linker options
- intermediate files -- object (.o), assembler (.s), preprocessor (.i), etc.

Remaining Chapters

A more detailed look at many C features
- Variables and declarations
- Operators
- Control Structures
- Functions
- Pointers and Data Structures
- I/O

Emphasis on how C is converted to assembly language

Also see “C Reference” in Appendix D
Basic C Elements

Variables
- Named, typed data items

Operators
- Predefined actions performed on data items
- Combined with variables to form expressions, statements

Statements and Functions
- Group together operations

Data Types
C has several basic data types

- `int` integer (at least 16 bits, commonly 32 bits)
- `long` integer (at least 32 bits)
- `float` floating point (at least 32 bits)
- `double` floating point (commonly 64 bits)
- `char` character (at least 8 bits)

Exact size can vary, depending on processor
- `int` is supposed to be "natural" integer size:
  - for LC-3, that's 16 bits -- 32 bits for most modern processors

Signed vs unsigned:
- Default is 2's complement signed integers
- Use "unsigned" keyword for unsigned numbers

Examples

Legal

```c
i
wordsPerSecond
words_per_second
_green
aReally_longName_moreThan31chars
aReally_longName_moreThan31characters
```

Illegal

```c
10s digit
'ten's digit
done?
done
double
double
```
**Literals**

**Integer**

- `123 /* decimal */`
- `-123`
- `0x123 /* hexadecimal */`

**Floating point**

- `6.023`
- `6.023e23 /* 6.023 x 10^{23} */`
- `5E12 /* 5.0 x 10^{12} */`

**Character**

- `'c'`
- `'\n' /* newline */`
- `'\xA' /* ASCII 10 (0xA) */`

---

**Scope: Global and Local**

Where is the variable accessible?

- **Global**: accessed anywhere in program
- **Local**: only accessible in a particular region

Compiler infers scope from where variable is declared

- Programmer doesn't have to explicitly state

Variable is local to the block in which it is declared

- Block defined by open and closed braces `{ }`
- Can access variable declared in any "containing" block

Global variable is declared outside all blocks

---

**Example**

```c
#include <stdio.h>

int itsGlobal = 0;

main()
{
    int itsLocal = 1; /* local to main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    { int itsLocal = 2; /* local to this block */
      itsGlobal = 4; /* change global variable */
      printf("Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```

---

**Expression**

Any combination of variables, constants, operators, and function calls

- Every expression has a type, derived from the types of its components (according to C typing rules)

Examples:

```
counter >= STOP
x + sqrt(y)
x & z + 3 || 9 - w-- % 6
```
Statement
Expresses a complete unit of work
  • Executed in sequential order

Simple statement ends with semicolon
  \[ z = x * y; /* assign product to z */ \]
  \[ y = y + 1; /* after multiplication */ \]
  \[ ; /* null statement */ \]

Compound statement formed with braces
  • Syntactically equivalent to a simple statement
  \[ \{ z = x * y; y = y + 1; \} \]

Operators
Three things to know about each operator
(1) Function
  • What does it do?
(2) Precedence
  • In which order are operators combined?
    • Example:
      "a + b * c" is the same as "(a + b) * c" because multiplication (*) has a higher precedence than addition (+)
(3) Associativity
  • In which order are operators of the same precedence combined?
    • Example:
      "a - b - c" is the same as "(a - b) - c" because add/sub associate left-to-right

Assignment Operator
Changes the value of a variable
  \[ x = x + 4; \]
  1. Evaluate right-hand side.
  2. Set value of left-hand side variable to result.

Assignment Operator
All expressions evaluate to a value, even ones with the assignment operator

For assignment, the result is the value assigned
  • Usually (but not always) the value of the right-hand side
    ➢ Type conversion might make assigned value different than computed value

Assignment associates right to left.
  \[ y = x = 3; \]
  \[ y \] gets the value 3, because \( x = 3 \) evaluates to the value 3
  \[ y = (x = 3); \]
### Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>x % y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>x + y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>x - y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

All associate left to right

* / % have higher precedence than + -

**Example**
- \(2 + 3 \times 4\) versus \((2 + 3) \times 4\)

### Arithmetic Expressions

If mixed types, smaller type is "promoted" to larger

\(x + 4.3\)

if \(x\) is int, converted to double and result is double

**Integer division -- fraction is dropped**

\(x / 3\)

if \(x\) is int and \(x=5\), result is 1 (not 1.666666...)

**Modulo -- result is remainder**

\(x \% 3\)

if \(x\) is int and \(x=5\), result is 2

### Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>x &lt;&lt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>x &gt;&gt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>x &amp; y</td>
<td>11</td>
<td>l-to-r</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>x ^ y</td>
<td>12</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

Operate on variables bit-by-bit
- Like LC-3 AND and NOT instructions

Shift operations are logical (not arithmetic)
- Operate on values -- neither operand is changed
- \(x = y << 1\) same as \(x = y+y\)

### Logical Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>!x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>logical AND</td>
<td>x &amp; &amp; y</td>
<td>14</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>logical OR</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Treats entire variable (or value) as
- TRUE (non-zero), or
- FALSE (zero).

**Result is 1 (TRUE) or 0 (FALSE)**

\(x = 15; y = 0; \text{printf(\"%d\", \(x | | y\});}\)

**Bit-wise vs Logical**
- \(1 \& 8 = 0\) (000001 AND 001000 = 000000)
- \(1 \& \& 8 = 1\) (True \& True = True)
### Relational Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>x &gt; y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>x &gt;= y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>x &lt; y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>x &lt;= y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>x == y</td>
<td>10</td>
<td>l-to-r</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>x != y</td>
<td>10</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

Result is 1 (TRUE) or 0 (FALSE)

### Assignment vs Equality

Don't confuse equality (==) with assignment (=)

```c
int x = 9;
int y = 10;
if (x == y) {
    printf("not executed\n");
}
```

```c
if (x = y) {
    printf("%d %d", x, y);
}
```

Result: “10 10" is printed. Why?

Compiler will not stop you! (What happens in Java?)

### Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>postincrement</td>
<td>x++</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>x--</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
</tbody>
</table>

**Pre:** Increment/decrement variable before using its value

**Post:** Increment/decrement variable after using its value

### Using ++ and --

```c
x = 4;
y = x++;
```

Results: x = 5, y = 4
(because x is incremented after assignment)

```c
x = 4;
y = ++x;
```

Results: x = 5, y = 5
(because x is incremented before assignment)

Please, don’t combine ++ and =. Really. Just don’t!
### Special Operators: `+=, *=, etc.`

Arithmetic and bitwise operators can be combined with assignment operator

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x += y;</code></td>
<td><code>x = x + y;</code></td>
</tr>
<tr>
<td><code>x -= y;</code></td>
<td><code>x = x - y;</code></td>
</tr>
<tr>
<td><code>x *= y;</code></td>
<td><code>x = x * y;</code></td>
</tr>
<tr>
<td><code>x /= y;</code></td>
<td><code>x = x / y;</code></td>
</tr>
<tr>
<td><code>x %= y;</code></td>
<td><code>x = x % y;</code></td>
</tr>
<tr>
<td><code>x &amp;= y;</code></td>
<td><code>x = x &amp; y;</code></td>
</tr>
<tr>
<td>`x</td>
<td>= y;`</td>
</tr>
<tr>
<td><code>x ^= y;</code></td>
<td><code>x = x ^ y;</code></td>
</tr>
<tr>
<td><code>x &lt;&lt;= y;</code></td>
<td><code>x = x &lt;&lt; y;</code></td>
</tr>
<tr>
<td><code>x &gt;&gt;= y;</code></td>
<td><code>x = x &gt;&gt; y;</code></td>
</tr>
</tbody>
</table>

All have same precedence and associativity as `=` and associate right-to-left.

### Practice with Precedence

Assume `a=1, b=2, c=3, d=4`

```plaintext
x = a * b + c * d / 2;  /* x = 8 */
```

same as:

```plaintext
x = (a * b) + ((c * d) / 2);
```

For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.

Note: Assignment operator has lowest precedence, so all the arithmetic operations on the right-hand side are evaluated first.

### Special Operator: Conditional

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?:</code></td>
<td>conditional</td>
<td><code>x?y:z</code></td>
<td>16</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

- `x ? y : z`
  - If `x` is non-zero, result is `y`
  - If `x` is zero, result is `z`

Seems useful, but I don’t use it
- A normal “if” is almost always more clear
- You don’t need to use every language feature
- Really, don’t use it (you don’t have to show how clever you are)

### Practice with Operators

In preparation for our dis-assembler (HW#8):

```c
int opcode(int ir) {
    /* code to extract bits 15 through 12 of ir */
}
```

```c
int get_field(int bits, int hi_bit, int lo_bit) {
    /* code to extract hi_bit through lo_bit of bits */
}
```

For example, body of `opcode` function is now just

```c
int ir = 0b1010;
```

What about a “signed-extended” version?

```c
int ir = 0b1010;
```
Practice with Operators (Solution 1)

```c
int opcode(int ir) {
    ir = ir >> 12;
    ir = ir & 0xf;
    return ir;
}
```

OR

```c
int opcode(int ir) {
    ir = ir & 0xf000;
    ir = ir >> 12;
    return ir;
}
```

Practice with Operators (Solution 2)

```c
int get_field(int bits, int hi_bit, int lo_bit) {
    int inv_mask = ~0 << (hi_bit+1)
    int mask = ~inv_mask;
    bits = bits & mask; // Mask high bits
    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}
```

OR

```c
int get_field(int bits, int hi_bit, int lo_bit) {
    bits = ~(~0 << (hi_bit+1)) & bits; // Mask high bits
    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}
```

Sign Extended Version

```c
int get_sext_field(int bits, int hi_bit, int lo_bit) {
    int most_significant_bit = bits & (1 << hi_bit);
    if (most_significant_bit != 0) {
        bits = (~0 << hi_bit) | bits; // One extend
    } else {
        bits = ~(~0 << (hi_bit+1)) & bits; // Zero extend
    }

    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}
```

Chapter 13
Control Structures
Control Structures

Conditional
• Making decision about which code to execute, based on evaluated expression
  • if
  • if-else
  • switch

Iteration
• Executing code multiple times, ending based on evaluated expression
  • while
  • for
  • do-while

Example If Statements
if (x <= 10)
  y = x * x + 5;
if (x <= 10) {
  y = x * x + 5;
  z = (2 * y) / 3;
}
if (x <= 10)
  y = x * x + 5;
  z = (2 * y) / 3;

If

```
if (condition)
  action;
```

Condition is a C expression, which evaluates to TRUE (non-zero) or FALSE (zero). Action is a C statement, which may be simple or compound (a block).

More If Examples
```
if (0 <= age && age <= 11) {
  kids = kids + 1;
}
if (month == 4 || month == 6 || month == 9 || month == 11) {
  printf("The month has 30 days.\n");
}
if (x = 2) {
  y = 5;
}
```

Common C error, assignment (=) Versus equality (==)
This is a common programming error (= instead of ==), not caught by compiler because it’s syntactically correct.
Generating Code for If Statement

```assembly
if (x == 2) {
    y = 5;
}
```

```
LDR R0, R6, #0  ; load x into R0
ADD R0, R0, #-2  ; subtract 2
BRnp NOT_TRUE  ; if non-zero, x is not 2
AND R1, R1, #0  ; store 5 to y
ADD R1, R1, #5
STR R1, R6, #1
```

```
DONE  ...  ; next statement
```

Generating Code for If-Else

```assembly
if (x) {
    y++;  
    z--; 
}
else {
    y--;  
    z++; 
}
```

```
LDR R0, R6, #0       ; load x into R0
ADD R0, R0, -2       ; subtract 2
BRz ...   STR R1, R6, #1       ; incr z
ADD R1, R1, #1       ; add 1 to z
STR R1, R6, #2
DONE  ...  ; next statement
```

If-else

```assembly
if (condition)  
    action_if;
else
    action_else;
```

```
condition

T  
F  

action_if  
action_else
```

Else allows choice between two mutually exclusive actions without re-testing condition.

Generating Code for If Else

```assembly
if (x != 10) {
    y = 5;
}
```

```
LDR R0, R6, #0  ; load x into R0
ADD R0, R0, #2  ; add 2 to x
AND R1, R1, #0  ; store 5 to y
ADD R1, R1, #5
STR R1, R6, #1
```

```
NOT_TRUE ...             ; next statement
```

Matching Else with If

Else is always associated with closest unassociated if

```assembly
if (x != 10)  
    if (y > 3)    z = z * 2;
else    z = z / 2;
```

```
else
    z = z * 2;
```

is NOT the same as...

```assembly
if (x != 10)  
    if (y > 3)    z = z / 2;
else
    z = z * 2;
```

is the same as...

```assembly
if (x != 10)  
    if (y > 3)    z = z / 2;
else
    z = z * 2;
```

Solution: always use braces (avoids the problem entirely)
Chaining If’s and Else’s

```c
if (month == 4 || month == 6 || month == 9 || month == 11) {
    printf("Month has 30 days.\n");
} else if (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12) {
    printf("Month has 31 days.\n");
} else if (month == 2) {
    printf("Month has 28 or 29 days.\n");
} else {
    printf("Don’t know that month.\n");
}
```

While

```c
while (test) {
    loop_body;
}
```

Executes loop body as long as test evaluates to TRUE (non-zero)

Note: Test is evaluated before executing loop body

Generating Code for While

```assembly
x = 0;
while (x < 10) {
    printf("%d ", x);
    x = x + 1;
}
```

Infinite Loops

The following loop will never terminate:

```c
x = 0;
while (x < 10) {
    printf("%d ", x);
}
```

Loop body does not change condition...
- ...so test is never false

Common programming error that can be difficult to find
For

```c
for (init; end-test; re-init) {
  statement
}
```

**Example For Loops**

```c
/* -- what is the output of this loop? -- */
for (i = 0; i <= 10; i++) {
  printf("%d ", i);
}

/* -- what does this one output? -- */
letter = 'a';
for (c = 0; c < 26; c++) {
  printf("%c ", letter+c);
}

/* -- what does this loop do? -- */
numberOfOnes = 0;
for (bitNum = 0; bitNum < 16; bitNum++) {
  if (inputValue & (1 << bitNum)) {
    numberOfOnes++;
  }
}
```

Generating Code for For

```c
for (i = 0; i < 10; i++) {
  printf("%d ", i);
}
```

```c
; init
AND R0, R0, #0
STR R0, R6, #0 ; i = 0
; test
LOOP LDR R0, R6, #0 ; load i
ADD R0, R0, #-10
BRzp DONE ; loop body
LDR R0, R6, #0 ; load i
...<printf>
...
; re-init
ADD R0, R0, #1 ; incr i
STR R0, R6, #0
BR LOOP ; test again
DONE ; next statement
```

**Nested Loops**

Loop body can (of course) be another loop

```c
/* print a multiplication table */
for (mp1 = 0; mp1 < 10; mp1++) {
  for (mp2 = 0; mp2 < 10; mp2++) {
    printf("%d\t", mp1*mp2);
  }
  printf("\n");
}
```
Another Nested Loop
Here, test for the inner loop depends on counter variable of outer loop

```java
for (outer = 1; outer <= input; outer++) {
    for (inner = 0; inner < outer; inner++) {
        sum += inner;
    }
}
```

Do-While

do
  loop_body;
while (test);

Executes loop body as long as test evaluates to TRUE (non-zero).

Note: Test is evaluated after executing loop body

For vs. While

In general:

For loop is preferred for counter-based loops
  • Explicit counter variable
  • Easy to see how counter is modified each loop

While loop is preferred for sentinel-based loops
  • Test checks for sentinel value.

Either kind of loop can be expressed as other, so really a matter of style and readability

Break and Continue

```java
break;
```
  • used only in switch statement or iteration statement
  • passes control out of the “nearest” (loop or switch) statement containing it to the statement immediately following
  • usually used to exit a loop before terminating condition occurs (or to exit switch statement when case is done)

```java
continue;
```
  • used only in iteration statement
  • terminates the execution of the loop body for this iteration
  • loop expression is evaluated to see whether another iteration should be performed
  • if for loop, also executes the re-initializer
Example
What does the following loop do?
for (i = 0; i <= 20; i++) {
    if (i%2 == 0) {
        continue;
    }
    printf("%d ", i);
}

What would be an easier way to write this?

What happens if break instead of continue?

Switch
switch (expression) {
    case const1:
        action1;
        break;
    case const2:
        action2;
        break;
    default:
        action3;
}

Alternative to long if-else chain.
If break is not used, then case "falls through" to the next.

More About Switch
Case expressions must be constant
    case i:   /* illegal if i is a variable */
If no break, then next case is also executed
switch (a) {
    case 1:
        printf("A");
    case 2:
        printf("B");
    default:
        printf("C");
}
Enumerations

Keyword `enum` declares a new type
- `enum colors { RED, GREEN, BLUE, GREEN, YELLOW, MAUVE };`
- RED is now 0, GREEN is 1, etc.
- Gives meaning to constants, groups constants

```cpp
def house_color = get_color();
switch (house_color) {
  case RED:
    /* code here */
    break;
    /* more here... */
}
```

Enums are just ints, but can provide more type checking
- Warning on assignment (example: house_color = 85;)
- Warning on "partial" switch statement
- C++ adds even more checking support

---

Example: Searching for Substring

Have user type in a line of text (ending with linefeed) and print the number of occurrences of "the"

Reading characters one at a time
- Use the `getchar()` function -- returns a single character

Don't need to store input string; look for substring as characters are being typed
- Similar to state machine:
  - based on characters seen, move toward success state or move back to start state
- Switch statement is a good match to state machine

---

Substring: State machine to flow chart

- Reading characters one at a time
- Process: `read char`
- States: `match`, `state of matching`
- Transition: match, no match
- Action: `increment count`
- Example: Searching for Substring

---

Subtring: Code (Part 1)

```cpp
#include <stdio.h>
enum state { NO_MATCH, ONE_MATCH, TWO_MATCHES };
main()
{
  char key; /* input character from user */
  int match = NO_MATCH; /* state of matching */
  int count = 0; /* number of substring matches */
  /* Read character until newline is typed */
  key = getchar();
  while (key != '\n') {
    /* Action depends on number of matches so far */
    switch (match) {
    case NO_MATCH:
      break;
      case ONE_MATCH:
        break;
        case TWO_MATCHES:
          break;
    default:
      break;
    }
    key = getchar();
  }
  printf("Number of matches = %d\n", count);
}
```
C and the Right Shift Operator (>>)

Does right shift sign extend or not?
  • Answer: Yes and No

Unsigned values: zero extend
  • unsigned int x = ~0;
  • Then, (x >> 10) will have 10 leading zeros

Signed values:
  • “Right shifting a signed quantity will fill with
    sign bits (‘arithmetic shift’) on some machines and
    with 0-bits (‘logical shift’) on others.” - Kernighan and Ritchie
  • In practice, it does sign extend
    ➢ int x = ~0; /* signed */
    ➢ Then, (x >> 10) will still be all 1s
Function
Smaller, simpler, subcomponent of program
Provides abstraction
- Hide low-level details
- Give high-level structure to program, easier to understand overall program flow
- Enables separable, independent development

C functions
- Zero or multiple arguments (or parameters) passed in
- Single result returned (optional)
- Return value is always a particular type

In other languages, called procedures, subroutines, ...

Example of High-Level Structure
void main()
{
    setup_board(); /* place pieces on board */
    determine_sides(); /* choose black/white */

    /* Play game */
    while (no_outcome_yet()){
        whites_turn();
        blacks_turn();
    }
}

Structure of program is evident, even without knowing implementation.

Functions in C
Definition

int factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}

Function call -- used in expression
a = x + factorial(f + g);

- 1. evaluate arguments
- 2. execute function
- 3. use return value in expression

Implementing Functions and Variables in LC-3
We’ve talked about...
- Variables
  - Local
  - Global
- Functions
  - Parameter passing
  - Return values

What does the assembly code look like for these idioms?

Important notes
- Different compilers for different ISAs do things differently
- As long as a compiler is consistent
- We’re straying from the book’s version to simplify things
  - Leaving out the R5 “frame pointer”
Allocating Space for Variables

Global data section
- All global variables stored here (actually all static variables)
- R4 points to beginning

Run-time stack
- Used for local variables
- R6 points to top of stack
- New frame for each block (goes away when block exited)

Offset = distance from beginning of storage area
- Global: LDR R1, R4, #4
- Local: LDR R2, R6, #3

Compiler keeps more information

Symbol Table

- In assembler, all identifiers were labels
- In compiler, identifiers are variables

Symbol Table Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>double</td>
<td>0</td>
<td>main</td>
</tr>
<tr>
<td>hours</td>
<td>int</td>
<td>1</td>
<td>main</td>
</tr>
<tr>
<td>minutes</td>
<td>int</td>
<td>2</td>
<td>main</td>
</tr>
<tr>
<td>seconds</td>
<td>int</td>
<td>3</td>
<td>main</td>
</tr>
</tbody>
</table>

Local Variable Storage

Local variables stored in activation record (stack frame)

Symbol table “offset” gives the distance from the base of the frame
- A new frame is pushed on the run-time stack each time block is entered
- R6 is the stack pointer – holds address of current top of run-time stack
- Because stack grows downward, stack pointer is the smallest address of the frame, and variable offsets are >= 0.

Symbol Table Example

```c
int main() {
    int seconds;
    int minutes;
    int hours;
    double amount;
    ...
}
```
Example: Compiling to LC-3

```c
#include <stdio.h>

int inGlobal;

main()
{
    int inLocal;
    int outLocalA;
    int outLocalB;

    /* initialize */
    inLocal = 5;
    inGlobal = 3;

    /* perform calculations */
    outLocalA = inLocal & ~inGlobal;
    outLocalB = (inLocal + inGlobal) + outLocalA;

    /* print results */
    printf("The results are: outLocalA = %d, outLocalB = %d\n", 
           outLocalA, outLocalB);
}
```

Example: Code Generation

```c
; main
; inLocal = 5
    AND R0, R0, #0
    ADD R0, R0, #5 ; inLocal = 5
    STR R0, R6, #2 ; (offset = 2)

; inGlobal = 3
    AND R0, R0, #0
    ADD R0, R0, #3 ; inGlobal = 3
    STR R0, R4, #0 ; (offset = 0)
```

Example (continued)

; first statement:
; outLocalA = inLocal & ~inGlobal;

```c
LDR R0, R6, #2 ; get inLocal (offset = 2)
LDR R1, R4, #0 ; get inGlobal
NOT R1, R1 ; ~inGlobal
AND R2, R0, R1 ; inLocal & ~inGlobal
STR R2, R6, #1 ; store in outLocalA
    (offset = 1)
```

Example (continued)

```c
; outLocalB = (inLocal + inGlobal) + outLocalA;

LDR R0, R6, #2 ; inLocal
LDR R1, R4, #0 ; inGlobal
ADD R0, R0, R1 ; R0 is sum
ADD R1, R4, #3 ; inGlobal
ADD R0, R0, R1 ; R0 is sum
ADD R2, R0, R1 ; R2 is sum
STR R2, R6, #0 ; outLocalB (offset = 0)
```
Implementing Functions

Activation record

- Information about each function, including arguments and local variables
- Also stored on run-time stack

Calling function

- Copy args into stack or regs
- Call function

Called function

- Allocate activation record
- Save registers
- Execute code
- Put result in AR or reg
- Pop activation record
- Return

Run-Time Stack for Functions

Main

Memory

R6

func

Memory

R6

main

Before call

During call

After call

Activation Record

int func(int a, int b)
{
    int w, x, y;
    ...
    return y;
}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>int</td>
<td>7</td>
<td>func</td>
</tr>
<tr>
<td>a</td>
<td>int</td>
<td>6</td>
<td>func</td>
</tr>
<tr>
<td>&quot;ret. value&quot;</td>
<td>int</td>
<td>5</td>
<td>func</td>
</tr>
<tr>
<td>w</td>
<td>int</td>
<td>2</td>
<td>func</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>1</td>
<td>func</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>0</td>
<td>func</td>
</tr>
</tbody>
</table>

Activation Record Bookkeeping

Return value

- Space for value returned by function
- Allocated even if function does not return a value

Return address

- Save pointer to next instruction in calling function
- Convenient location to store R7
  - in case another function (JSR) is called

Save registers

- Save all other registers used (but not R6, and often not R4)
Function Call Example

```c
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    return val;
}

int max(int a, int b)
{
    int result;
    result = a;
    if (b > a) {
        result = b;
    }
    return result;
}
```

Max Function

```assembly
MAX     ADD R6, R6, #-7   ; allocate frame
        STR R7, R6, #3    ; save R7 (link register)
        STR R1, R6, #2    ; save R1
        STR R0, R6, #1    ; save R0
        LDR R0, R6, #5    ; load "a" into R0
        STR R0, R6, #0    ; store "a" into R0
        LDR R1, R6, #6    ; load "b" into R1
        NOT R1, R1        ; calculate -b
        ADD R1, R1, R0    ; compare
        BRP AFTER
        LDR R0, R6, #5    ; load "a" into R0
        STR R0, R6, #0    ; store "a" into R0
AFTER   LDR R0, R6, #0    ; load "result" into return value
        STR R0, R6, #4    ; store "result" into return value
        LDR R0, R6, #1    ; restore R0
        LDR R1, R6, #2    ; restore R1
        LDR R7, R6, #3    ; restore R7 (link register)
        ADD R6, R6, #7    ; pop stack
        RET
```

Main Function (1 of 2)

```assembly
MAIN     ADD R6, R6, #-4   ; allocate frame
        ADD R0, R0, #10   ; x = 10
        STR R0, R6, #2   ; y = 11
        ADD R0, R0, #11   ; R0 = x + 10
        STR R0, R6, #1   ; load y into R0
        STR R0, R6, #-1  ; 2nd argument
        LDR R0, R6, #1    ; load x into R1
        ADD R1, R1, R0    ; R1 = x + 10
        STR R1, R6, #-2  ; 1st argument
        JSR MAX           ; call max function
```

Main Function (2 of 2)

```assembly
        ; previous code here
        JSR MAX           ; call max function
        LDR R0, R6, #-3   ; read return value of max
        STR R0, R6, #0    ; put value into local "val"
        LDR R0, R6, #0    ; load "val"
        STR R0, R6, #3    ; put "val" into main's return value
        ADD R6, R6, #4    ; pop stack
        RET
```
Summary of LC-3 Function Call Implementation

1. **Caller** places arguments on stack (last to first)
2. **Caller** invokes subroutine (JSR)
3. **Callee** allocates frame
4. **Callee** saves R7 and other registers
5. **Callee** executes function code
6. **Callee** stores result into return value slot
7. **Callee** restores registers
8. **Callee** deallocates frame (local vars, other registers)
9. **Callee** returns (RET or JMP R7)
10. **Caller** loads return value
11. **Caller** resumes computation...

Callee versus Caller Saved Registers

**Callee saved registers**
- In our examples, the callee saved and restored registers
- Saves/restores any registers it modifies

**What if a you wants R7 to be preserved across a call?**
- Before call: caller saves it on the stack
- After call: caller restores it from the stack

**Caller saved registers**
- R7 is an example of a caller saved register
- Value assumed destroyed across calls
- Only needs to save R7 when it’s in use

**Which is better? Callee or Caller saved registers?**
- Neither: many ISA calling conventions specify some of each

Compilers are Smart(er)

In our examples, variables always stored in memory
- Read from stack, written to stack

Compiler will perform code optimizations
- Keeps many variables in registers
- Avoids many save/restores of registers
- Why?

Passing parameter values in registers
- First few parameters in registers
- Return value in register
- Like in your homework projects
- Again, why?