Problem Solving in C

Stepwise Refinement

• as covered in Chapter 6

...but can stop refining at a higher level of abstraction.

Same basic constructs

• **Sequential** -- C statements
• **Conditional** -- if-else, switch
• **Iterative** -- while, for, do-while
Problem 1: Calculating Pi

Calculate $\pi$ using its series expansion.
User inputs number of terms.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + \left(-1\right)^{n-1} \frac{4}{2n+1} + \cdots$$
Pi: 1st refinement

- Start
- Initialize
- Get Input
- Evaluate Series
- Output Results
- Stop

Initialize iteration count

Evaluate next term

count = count + 1

for loop

count < terms

F

T
Pi: 2nd refinement

1. Initialize iteration count
2. If count < terms, go to Evaluate next term.
3. If count is odd, subtract term; else add term.
4. If count = count+1, go back to Evaluate next term.
5. If count is odd, subtract term; else add term.
But, (Chaiken says) it's better to decide exactly what the variables (like count and terms) will mean!

count : variable that tracks which term we will calculate and add to the sum. The first term (4) will correspond to count==0. The second term (-4/3) will correspond to count==1, etc.

So, yes, for even count (0, 2, 4) we add a positive term; for odd count (1, 3, 5) we subtract a positive term.

terms: Number of terms to add up; input from user.

So: if count < terms, we add another term; if count==terms we stop before adding another term.
Pi: Code for Evaluate Terms

for (count=0; count < numOfTerms; count++) {
    if (count % 2) {
        /* odd term -- subtract */
        pi -= 4.0 / (2 * count + 1);
    }
    else {
        /* even term -- add */
        pi += 4.0 / (2 * count + 1);
    }
}

Note: Code in text is slightly different, but this code corresponds to equation.
#include <stdio.h>

main() {
    double pi = 0.0;
    int numOfTerms, count;

    printf("Number of terms (must be 1 or larger) : ");
    scanf("%d", &numOfTerms);

    for (count=0; count < numOfTerms; count++) {
        if (count % 2) {
            pi -= 4.0 / (2 * count + 1); /* odd term -- subtract */
        } else {
            pi += 4.0 / (2 * count + 1); /* even term -- add */
        }
    }

    printf("The approximate value of pi is \%f\n", pi);
}
C: A High-Level Language

Gives symbolic names to values
  • don’t need to know which register or memory location

Provides abstraction of underlying hardware
  • operations do not depend on instruction set
  • example: can write “a = b * c”, even though LC-3 doesn’t have a multiply instruction

Provides expressiveness
  • use meaningful symbols that convey meaning
  • simple expressions for common control patterns (if-then-else)

Enhances code readability

Safeguards against bugs
  • can enforce rules or conditions at compile-time or run-time
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

Symbol Table
• map between symbolic names and items
• like assembler, but more kinds of information
Functions in C

Declaration (also called prototype)

```c
int Factorial(int n);
```

- type of return value
- name of function
- types of all arguments

Function call -- used in expression

```c
a = x + Factorial(f + g);
```

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

State type, name, types of arguments

• must match function declaration
• give name to each argument (doesn't have to match declaration)

```c
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```
gives control back to calling function and returns value
Why Declaration?

Since function definition also includes return and argument types, why is declaration needed?

• Use might be seen before definition. Compiler needs to know return and arg types and number of arguments.

• Definition might be in a different file, written by a different programmer.
  • include a "header" file with function declarations only
  • compile separately, link together to make executable
Example

double ValueInDollars(double amount, double rate);

main()
{
  ...
  dollars = ValueInDollars(francs,
                           DOLLARS_PER_FRANC);
  printf("%f francs equals %f dollars.\n",
         francs, dollars);
  ...
}

double ValueInDollars(double amount, double rate)
{
  return amount * rate;
}
Implementing Functions: Overview

Activation record

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

Calling function

- push new activation record
- copy values into arguments
- call function
- get result from stack

Called function

- execute code
- put result in activation record
- pop activation record from stack
- return
Run-Time Stack

Recall that local variables are stored on the run-time stack in an activation record.

Frame pointer (R5) points to the beginning of a region of activation record that stores local variables for the current function.

When a new function is called, its activation record is pushed on the stack;

when it returns, its activation record is popped off of the stack.
Run-Time Stack

Memory

Before call

During call

After call
Activation Record

```c
int NoName(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>a</td>
<td>int</td>
<td>4</td>
<td>NoName</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
<td>5</td>
<td>NoName</td>
</tr>
<tr>
<td>w</td>
<td>int</td>
<td>0</td>
<td>NoName</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>-1</td>
<td>NoName</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>-2</td>
<td>NoName</td>
</tr>
</tbody>
</table>

**bookkeeping**
- dynamic link
- return address
- return value

**locals**
- y
- x
- w

**args**
- a
- b

R5
Arg VALUES are COPIED to Fun PARAMETERS

main()
{
    printf("x%x", Factorial(3));
}

.ORIG x3000
Brnzp main

Three .FILL #3
main LD R0, Three ;R0: LC3 Fun Arg Reg
    JSR Factorial
    JSR PrintHex
    HALT
Arg VALUES are COPIED to Fun PARAMETERS

```c
int Factorial(int n)
{
    if( n == 0 ) return 1;
    else return( Factorial( n - 1 )*n );
}

HALT
```

```assembly
Factorial	ADD R6,R6,#-2 ;Make stack frame
            STR R7,R6,#0 ;0(R6) saves R7
            STR R0,R6,#1

; Mem[R6+1] IS the Fun PARAMETER Variable!
            BRnp FactRec ;smart!
            AND R0,R0,#0
            ADD R0,R0,#1
            Brnzp FactRet
```
Arg VALUES are COPIED to Fun PARAMETERS

```c
int Factorial(int n)
{
    .... return( Factorial( n - 1 )*n );
}
```

Mem[R6+1] IS the Fun PARAMETER Variable!

```asm
BRnp FactRec ;smart!

... ... Brnzp FactRet
```

```asm
FactRec ADD R0,R0,#-1 ;calc n-1, smart!
JSR Factorial ;ARG is n-1.
; now, R0 holds (n-1)! (FACT!)
```

```asm
LDR R1,R6,#1 ;copy n into R1
JSR Multiply ;please write for LC3
```

```asm
FactRet LDR R7,R6,#0 ; and forget abt n.
ADD R6,R6,#2 ; pop stack
RET
```
Arg VALUES are COPIED to Fun PARAMETERS

main() {
    char PATTERN[31], SUBJECT[31];
    Brnzp Main
    PATTERN .BLKW #31
    SUBJECT .BLKW #31
    PATA .FILL PATTERN
    SUBA .FILL SUBJECT
    /* Inputting is OMITTED */
    if(FUN( PATTERN, SUBJECT )) printf("Yes");
    Main LD R0,PATA
    LD R1,SUBA
    JSR FUN
    Brz NoMatch
int FUN(char *pat, char *sub) {
    if( *pat == '\0' && *sub == '\0' ) ...  
    FUN  ADD   R6,R6,#-3 ;alloc. stack frame
    STR   R7,R6,#0  ;save ret. address
    STR   R0,R6,#1  ;1(R6) used for pat PARA
    STR   R1,R6,#2  ;2(R6) used for sub PARA
    LDR   R0,R6,#1  ;Load pat value:a POINTER
    LDR   R0,R0,#0  ;Load value IT POINTS TO
    BRnp  IfIsFalse  ;C doesn't bother
    LDR   R0,R6,#2  ;Load sub (pointer!)
    LDR   R0,R0,#0  ;Load (char) it points to
    BRnp  IfIsFalse
    ;Generate a 1 in R0
    BRnzp FunEpilogue
Function Definition

State type, name, types of arguments

- must match function declaration
- give name to each argument (doesn't have to match declaration)

```c
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
gives control back to calling function and returns value
```
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

Dynamic link

• caller’s frame pointer
• used to pop this activation record from stack
Example Function Call

```c
int Volta(int q, int r)
{
    int k;
    int m;
    ...
    return k;
}
```

```c
int Watt(int a)
{
    int w;
    ...
    w = Volta(w,10);
    ...
    return w;
}
```
Calling the Function

\[ w = \text{Volta}(w, 10); \]

; push second arg
AND  R0, R0, #0
ADD  R0, R0, #10
ADD  R6, R6, #-1
STR  R0, R6, #0

; push first argument
LDR  R0, R5, #0
ADD  R6, R6, #-1
STR  R0, R6, #0

; call subroutine
JSR  Volta

Note: Caller needs to know number and type of arguments, doesn't know about local variables.
Starting the Callee Function

; leave space for return value
ADD R6, R6, #−1

; push return address
ADD R6, R6, #−1
STR R7, R6, #0

; push dyn link (caller’s frame ptr)
ADD R6, R6, #−1
STR R5, R6, #0

; set new frame pointer
ADD R5, R6, #−1

; allocate space for locals
ADD R6, R6, #−2
Ending the Callee Function

```assembly
return k;

; copy k into return value
LDR  R0, R5, #0
STR  R0, R5, #3

; pop local variables
ADD  R6, R5, #1

; pop dynamic link (into R5)
LDR  R5, R6, #0
ADD  R6, R6, #1

; pop return addr (into R7)
LDR  R7, R6, #0
ADD  R6, R6, #1

; return control to caller
RET
```
Resuming the Caller Function

\[ w = \text{Volta}(w, 10); \]

```
JSR Volta

; load return value (top of stack)
LDR R0, R6, #0
; perform assignment
STR R0, R5, #0
; pop return value
ADD R6, R6, #1
; pop arguments
ADD R6, R6, #2
```
Summary of LC-3 Function Call Implementation

1. **Caller** pushes arguments (last to first).
2. **Caller** invokes subroutine (JSR).
3. **Callee** allocates return value, pushes R7 and R5.
4. **Callee** allocates space for local variables.
5. **Callee** executes function code.
6. **Callee** stores result into return value slot.
7. **Callee** pops local vars, pops R5, pops R7.
8. **Callee** returns (JMP R7).
9. **Caller** loads return value and pops arguments.
10. **Caller** resumes computation…
A Simple C Program

#include <stdio.h>
#define STOP 0

/* Function: main */
/* Description: counts down from user input to STOP */
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);
    }
}
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.
Symbol Table

Like assembler, compiler needs to know information associated with identifiers

- in assembler, all identifiers were labels and information is address

Compiler keeps more information

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>int</td>
<td>0</td>
<td>main</td>
</tr>
<tr>
<td>hours</td>
<td>int</td>
<td>-3</td>
<td>main</td>
</tr>
<tr>
<td>minutes</td>
<td>int</td>
<td>-4</td>
<td>main</td>
</tr>
<tr>
<td>rate</td>
<td>int</td>
<td>-1</td>
<td>main</td>
</tr>
<tr>
<td>seconds</td>
<td>int</td>
<td>-5</td>
<td>main</td>
</tr>
<tr>
<td>time</td>
<td>int</td>
<td>-2</td>
<td>main</td>
</tr>
</tbody>
</table>
Symbol Table

Like assembler, compiler needs to know information associated with identifiers

• in assembler, all identifiers were labels and information is address

In C: LOCAL lifetime versus GLOBAL (static) lifetime.

• LOCAL lifetime variables live in the activation records which live int the STACK.

• Symbol table tracks their OFFSET from frame pointer, so offset from the stack pointer can be known.

• \{ int A; char *B; ... ADD R6,R6,#-[the appropriate number] \\
• GLOBAL (static) lifetime variables live in non-stack memory forever

• Symbol table tracks their absolute memory address

• In C, declare OUTSIDE a fun. body to make a global livetimer.

• int DEBUG = 0; DEBUG .FILL #0

• int InitMeLater; InitMeLater .BLKW #1
Example: Code Generation

; main

; initialize variables

    AND R0, R0, #0
    ADD R0, R0, #5 ; inLocal = 5
    STR R0, R5, #0 ; (offset = 0)

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

; first statement:

; outLocalA = inLocal++ & ~inGlobal;

    LDR R0, R5, #0 ; get inLocal
    ADD R1, R0, #1 ; increment
    STR R1, R5, #0 ; store

    LDR R1, R4, #0 ; get inGlobal
    NOT R1, R1 ; ~inGlobal
    AND R2, R0, R1 ; inLocal & ~inGlobal
    STR R2, R5, #-1 ; store in outLocalA
        ; (offset = -1)
Example (continued)

; next statement:
; outLocalB = (inLocal + inGlobal) 
; - (inLocal - inGlobal);

LD R0, R5, #0 ; inLocal
LDR R1, R4, #0 ; inGlobal
ADD R0, R0, R1 ; R0 is sum
LDR R2, R5, #0 ; inLocal
LDR R3, R5, #0 ; inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3 ; R2 is difference
NOT R2, R2 ; negate
ADD R2, R2, #1
ADD R0, R0, R2 ; R0 = R0 - R2
STR R0, R5, #-2 ; outLocalB (offset = -2)