Design and Debug: Essential Concepts
Numerical Conversions

CS 16: Solving Problems with Computers
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- We are grading your midterms this week!
  - So far, so good…
- A reminder about Labs
  - Please make sure to SIGN IN (or you will be counted as absent)
  - Please make sure to COLLECT YOUR HW in lab
- Next lab is a required pair programming assignment
  - You are required to work with a partner for Lab 5 (next week)
  - More on that later…

Programming and **submit.cs**: The Devil is in the Details…

Change Tests: 1_general -- Your program's output did not match the expected.

	Correct Output		Your Output
1	Enter number of cents (or zero to quit):	1	Enter number of cents (or zero to quit) :
2	96 cents can be given as 3 quarters, 2 dimes, 1 penny.	2	96 cents can be given as 3 quarters, 2 dimes, 1 penny.
3	Enter number of cents (or zero to quit):	3	Enter number of cents (or zero to quit) :
...	<<Remaining diff not shown>>	...	<<Remaining diff not shown>>

Change Tests: 2_single -- Your program's output did not match the expected.

	Correct Output		Your Output
1	Enter number of cents (or zero to quit):	1	Enter number of cents (or zero to quit) :
2	25 cents can be given as 1 quarter.	2	25 cents can be given as 1 quarter.
3	Enter number of cents (or zero to quit):	3	Enter number of cents (or zero to quit) :
...	<<Remaining diff not shown>>	...	<<Remaining diff not shown>>

Change Tests: 3_multiple -- Your program's output did not match the expected.

	Correct Output		Your Output
1	Enter number of cents (or zero to quit):	1	Enter number of cents (or zero to quit) :
2	50 cents can be given as 2 quarters.	2	50 cents can be given as 2 quarters.
3	Enter number of cents (or zero to quit):	3	Enter number of cents (or zero to quit) :
...	<<Remaining diff not shown>>	...	<<Remaining diff not shown>>

Lecture Outline

- Design and Debug Tips
  - Designing and Debugging Loops
  - The Mighty TRACE
  - Designing and Debugging Functions
- Numerical Conversions
  - The Positional Notation
  - Going from Binary to Decimal (and Octal, and Hexadecimal)
  - Going from Decimal to… anything…

Designing Loops

What do I need to know?

- What am I **doing** inside the loop?
- What are my **initializing** statements?
- What are the **conditions for ending** the loop?

Exit on Flag Condition

- Loops can be ended when a particular flag condition exists
  - Applies to while and do-while loops
  - ***Flag***: A variable that changes value to indicate that some event has taken place
  - Examples of exit on a flag condition for input
    - List ended with a sentinel value
    - Running out of input

Exit on Flag Example

- Consider this loop to identify a student with a grade of 90 or better and think of how it's logically limited.

```
int n = 1;                  //student ID number
grade = compute_grade(n); // compute_grade() is a function
while (grade < 90)
{
    grade = compute_grade(n);
    cout <<"Student number "<< n <<" has a score of " << grade << endl;
    n++;
}
```

The Problem

```
int n = 1;                  //student ID number
grade = compute_grade(n);   // compute_grade() is a function
while (grade < 90)
{
   grade = compute_grade(n);
   cout <<"Student number "<< n <<" has a score of " << grade << endl;
   n++;
}
```

- The loop on the previous slide might not stop at the end of the list of students if ***no*** student has a grade of 90 or higher!
- It is a good idea to use **a second flag** to ensure that there are still students to consider
- The code on the following slide shows a better solution

Exit on Flag Example

```
int n = 1;                      //student ID number
grade = compute_grade(n);   // compute_grade() is a function
while ( (grade < 90) && ( n < number_of_students) )
{
   grade = compute_grade(n);
   cout <<"Student number "<< n <<" has a score of " << grade << endl;
   n++;
}
```

Debugging Loops

Common errors involving loops include:

- ***Off-by-one*** errors in which the loop executes one too many or one too few times
- ***Infinite loops*** usually result from a mistake in the Boolean expression that controls the loop

Fixing Off-By-One Errors

- Check your comparison: **should it be < or <= ?**
  - Saw a few mistakes like this on the exam ☹
- Check that the var. initialization uses the correct value

Fixing Infinite Loops

- Common mistake: check the direction of inequalities:

  should I use < or > ?

- Test for < or > in your loops,

  rather than equality (==) or inequality (!=)

More Loop Debugging Tips: **Tracing**

- Be sure that the mistake is *really in the loop*

- **Trace** the variable to observe *how* it changes
  - Tracing a variable is watching its value change *during* execution.
  - Best way to do this is to insert **cout** statements
    and have the program *show you* the variable
    at every iteration of the loop.

Debugging Example

- The following code is supposed to conclude with the variable "**`product`**" equal to the product of the numbers 2 through 5
  - i.e. 2 x 3 x 4 x 5, which, of course, is 120.

- What could go wrong?! ☺

Where might **you** put a trace?

```
int next = 2, product = 1;
   while (next < 5)
      {
         next++;
         product = product * next;
      }
```

DEMO!

Using variable tracing

Loop Testing Guidelines

- **Every time a program is changed, it should be re-tested**
  - Changing one part may require a change to another

- Every loop should at least be tested using input to cause:
  - Zero iterations of the loop body
  - One iteration of the loop body
  - One less than the maximum number of iterations
  - The maximum number of iterations

If all of these are ok, you likely have a very robust loop

Starting Over

- Sometimes it is more efficient to throw out a buggy program and start over!
  - The new program will be easier to read
  - The new program is less likely to be as buggy
  - You may develop a working program faster than if you work to repair the bad code
    - The lessons learned in the buggy code will help you design a better program faster

Testing and Debugging **Functions**

- Each function should be tested as a separate unit
- Testing individual functions facilitates finding mistakes
- "Driver Programs" allow testing of individual functions
- Once a function is tested, it can be used in the driver program to test other functions

Example of a Driver Test Program

```
int main( )
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

Stubs

- When a function being tested
  calls other functions that are not yet tested,
  use a **stub**
- A stub is a *simplified version of a function*
- Stubs usually **provide values for testing** rather than perform the intended calculation
  - i.e. **they're fake functions**
- **Stubs should be so simple**
  that you have full confidence they will perform correctly

Stub Example

```
1   #include <iostream>
2   #include <cmath>
3   use namespace std;
4   double WeirdCalc(double x, double y);
5
6   int main( ) {
7       double n, m, w;
8       cout << “Enter the 2 values for weird calculation: ”;
9       cin >> n >> m;
10      w = WeirdCalc(n, m) / (37 – pow(n/m, m/n) );
11      cout << “The answer is: ” << w << endl;
12      return 0;
13  }
14
```

Stub Example

```
1   #include <iostream>
2   #include <cmath>
3   use namespace std;
4   double WeirdCalc(double x, double y);
5
6   int main( ) {
7       double n, m, w;
8       cout << “Enter the 2 values for weird calculation: ”;
9       cin >> n >> m;
10      w = WeirdCalc(n, m) / (37 – pow(n/m, m/n) );
11      cout << “The answer is: ” << w << endl;
12      return 0;
13  }
14
15  double WeirdCalc(double x, double y) // Make WeirdCalc a stub – just for testing!!
16  {
17      //return ( (sqrt(pow(3*x, y%(max(x,y))) – sqrt(5*y/(x-6)) + 0.5*pow((x+y), -0.3);
18      return ( 7 );
19  }
```

Debugging Your Code

- Keep an open mind
  - Don't assume the bug is in a particular location
- **Don't randomly change code** without understanding what you are doing until the program works
  - This strategy may work for the first few small programs you write but it is doomed to failure for any programs of moderate complexity
- Show the program to someone else

General Debugging Techniques

- Check for common errors, for example:
  - Local vs. Reference Parameters
  - **=** instead of **==**
  - Did you use **&&** when you meant **||**?
  - These are typically errors that might not get flagged by a compiler
- Localize the error
  - Narrow down bugs by using **`cout`** statements to reveal internal (hidden) values of variables
  - Once you reveal the bug and fix it, remove the **cout** statements
- Your textbook has great debugging examples

Example from the Midterm

```
cout << "Enter 2 integer numbers. To quit, make either of them zero: ";
cin >> num1 >> num2;

while ( (num1 != 0) || (num2 != 0) && )
{
    if (num1 > num2) cout << "The sum is: " << num1 + num2 << endl;
    else if (num1 < num2) cout << "The product is: " << num1*num2 << endl;
    else cout << "You entered the same number, " << num1 << endl;

    cout << "Enter 2 integer numbers. To quit, make either of them zero: ";
    cin >> num1 >> num2;
} // end while

cout << "Goodbye!";
```

Example from the Midterm

```
cout << "Enter 2 integer numbers. To quit, make either of them zero: ";
cin >> num1 >> num2;

while ( (num1 != 0) && (num2 != 0) )
{
    if (num1 > num2) cout << "The sum is: " << num1 + num2 << endl;
    else if (num1 < num2) cout << "The product is: " << num1*num2 << endl;
    else cout << "You entered the same number, " << num1 << endl;

    cout << "Enter 2 integer numbers. To quit, make either of them zero: ";
    cin >> num1 >> num2;
} // end while

cout << "Goodbye!";
```

Because you want the cases when either var is zero. That is, if num1 is zero – it doesn't matter what num2 is doing – just quit (same for if num2 is zero)

Otherwise, if you use the || operator, then you are saying that while either var is zero, keep going thru the loop – this is not the intended design!!!

Other Debugging Techniques

- Use a **debugger tool**
  - Typically part of an IDE (integrated development environment)
  - Allows you to stop and step through a program line-by-line while inspecting variables

- Use the **assert** macro
  - Can be used to test pre or post conditions

```
#include <cassert>
assert(boolean expression)
```

  - If the Boolean is false then the program will abort
    - **Not** a good idea to keep in the program once you're done w/ it!!!

Assert Example

- Denominator should not be zero in Newton's Method

```
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
     double answer = 1;
     int i = 0;

     assert((n > 0) && (num_iterations> 0));
     while (i <num_iterations)
     {
          answer = 0.5 * (answer + n / answer);
          i++;
     }
     return answer;
}
```

Pre- and Post-Conditions

Concepts of pre-condition and post-condition in functions

The textbook recommends you use these concepts when making comments

Pre-condition: What must "be" before you call a function

- States what is assumed to be true when the function is called
- Function should not be used unless the precondition holds

Post-condition: What the function will do once it is called

- Describes the effect of the function call
- Tells what will be true after the function is executed (when the precondition holds)
- If the function returns a value, that value is described
- Changes to call-by-reference parameters are described

Why use Pre- and Post-conditions?

- Pre-conditions and post-conditions should be the first step in designing a function
- Specify what a function should do BEFORE designing it
  - This minimizes design errors and time wasted writing code that doesn't match the task at hand
- Read textbook's "Supermarket Pricing" case study
  - Ch. 5, from pg. 276 – 281

Numerical Conversions in CS

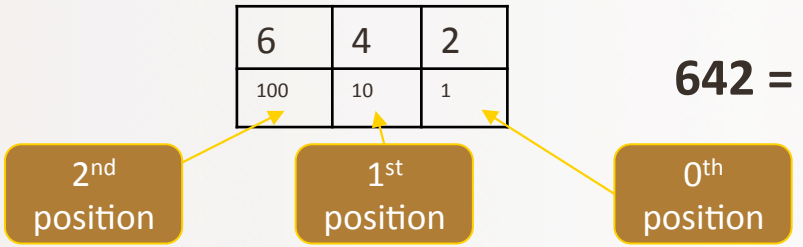
Counting Numbers in Different Bases

- We "normally" count in 10s
  - Base 10: decimal numbers
  - Number symbols are 0 thru 9
- Computers count in 2s
  - Base 2: binary numbers
  - Number symbols are 0 and 1
  - Represented with **1 bit** ($2^1 = 2$)
- Other convenient bases in computer architecture:
  - Base 8: octal numbers
  - Number symbols are 0 thru 7
  - Represented with **3 bits** ($2^3 = 8$)
  - Base 16: hexadecimal numbers
  - Number symbols are 0 thru F
    - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Represented with **4 bits** ($2^4 = 16$)
  - **Why are 4 bit representations convenient???**

Natural Numbers

Counting **642** as 600 + 40 + 2
is counting in TENS (aka BASE 10)

There are 6 HUNDREDS
There are 4 TENS
There are 2 ONES



642 = 600 + 40 + 2

Positional Notation in Decimal

Continuing with our example…

642 in base 10 *positional notation* is:

$$
\begin{aligned}
6 \times 10^2 &= 6 \times 100 &&= 600 \\
+\ 4 \times 10^1 &= 4 \times 10 &&= 40 \\
+\ 2 \times 10^0 &= 2 \times 1 &&= 2 \quad = \text{642 in base 10}
\end{aligned}
$$

Positional Notation

Anything → DEC

What if "642" is expressed in the base of 13?

$$6 \times 13^2 = 6 \times 169 = 1014$$
$$+ 4 \times 13^1 = 4 \times 13 = 52$$
$$+ 2 \times 13^0 = 2 \times 1 = 2$$
$$= 1068 \text{ in base } 10$$

So, "642" in base 13 is equivalent to "1068" in base 10

BUT WHO COUNTS IN BASE 13???!?!?

Maybe, aliens with 13 fingers???

COMPUTERS ARE DIGITAL (Binary) MACHINES

THEY ARE DESIGNED TO COUNT IN…

2

Positional Notation in Binary

11011 in base 2 *positional notation* is:

$$
\begin{aligned}
&1 \times 2^4 = 1 \times 16 = 16 \\
+\ &1 \times 2^3 = 1 \times 8 = 8 \\
+\ &1 \times 2^2 = 1 \times 4 = 4 \\
+\ &0 \times 2^1 = 1 \times 2 = 0 \\
+\ &1 \times 2^0 = 1 \times 1 = 1
\end{aligned}
$$

So, **1011** in base 2 is 16 + 8 + 0 + 2 + 1 = **27** in base 10

Converting Binary to Decimal

*Q: What is the decimal equivalent of the binary number **1101100**?*

A: Look for the position of the digits in the number.

This one has 7 digits, therefore positions 0 thru 6

1	1	0	1	1	0	0
64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

$1 \times 2^6 = 1 \times 64 = 64$
$+ 1 \times 2^5 = 1 \times 32 = 32$
$+ 0 \times 2^4 = 0 \times 16 = 0$
$+ 1 \times 2^3 = 1 \times 8 = 8$
$+ 1 \times 2^2 = 1 \times 4 = 4$
$+ 0 \times 2^1 = 0 \times 2 = 0$
$+ 0 \times 2^0 = 0 \times 1 = 0$
$= \underline{108}$ in base 10

Other Relevant Bases

- In Computer Science/Engineering, other binary-related numerical bases are used too.
- OCTAL: Base 8 (note that 8 is 2^3)
  - Uses the symbols: 0, 1, 2, 3, 4, 5, 6, 7
- HEXADECIMAL: Base 16(note that 16 is 2^4)
  - Uses the symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Converting Binary to Octal and Hexadecimal

(or any base that's a power of 2)

- Binary is 1 bit
- Octal is 3 bits ($2^3 = 8$) *octal is base 8*
- Hexadecimal is 4 bits ($2^4 = 16$) *hex is base 16*

- Use the "**group the bits**" technique
  - Always start from the *least significant digit*
  - Group every 3 bits together for bin → oct
  - Group every 4 bits together for bin → hex

Converting Binary to Octal and Hexadecimal

- Take the example: ***10100110***

…to octal:

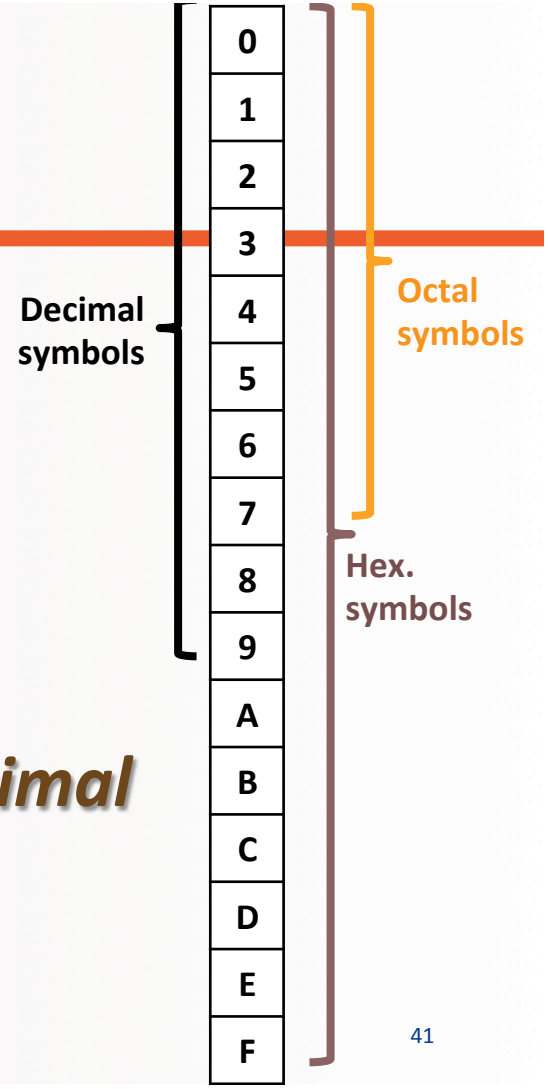
1 0	1 0 0	1 1 0
2	***4***	***6***

246 in octal

…to hexadecimal:

1 0 1 0	0 1 1 0
10	***6***

A6 in hexadecimal



Converting Decimal to Other Bases

Algorithm for converting number in base 10 to other bases

While (the **quotient** is not zero)

1. Divide the decimal number by the new base
2. Make the remainder the next digit to the left in the answer
3. Replace the original decimal number with the quotient
4. Repeat until your quotient is zero

EXAMPLE:

Convert the decimal (base 10) number **79** into hexadecimal (base 16)

79 / 16 = 4 R 15 *(15 in hex is the symbol "F")*

4 / 16 = 0 R 4

The answer is: **4F**

Converting Decimal into Binary

Convert 54 (base 10) into binary and hex:

- 54 / 2 = 27 R **0**
- 27 / 2 = 13 R **1**
- 13 / 2 = 6 R **1**
- 6 / 2 = 3 R **0**
- 3 / 2 = 1 R **1**
- 1 / 2 = 0 R **1**

Sanity check:
110110
= 2 + 4 + 16 + 32
= 54

54 (decimal) = 110110 (binary)
= 36 (hex)

YOUR TO-DOs

- Turn in HW4 on Thursday
- Lab 4 due Fri. 10/27
- HW5 will be released on Thursday, will be due in 1 week.
- Visit Prof's and TAs' office hours if you need help!

</LECTURE>