

Design and Debug: Essential Concepts

CS 16: Solving Problems with Computers I
Lecture #8

Ziad Matni
Dept. of Computer Science, UCSB

Outline

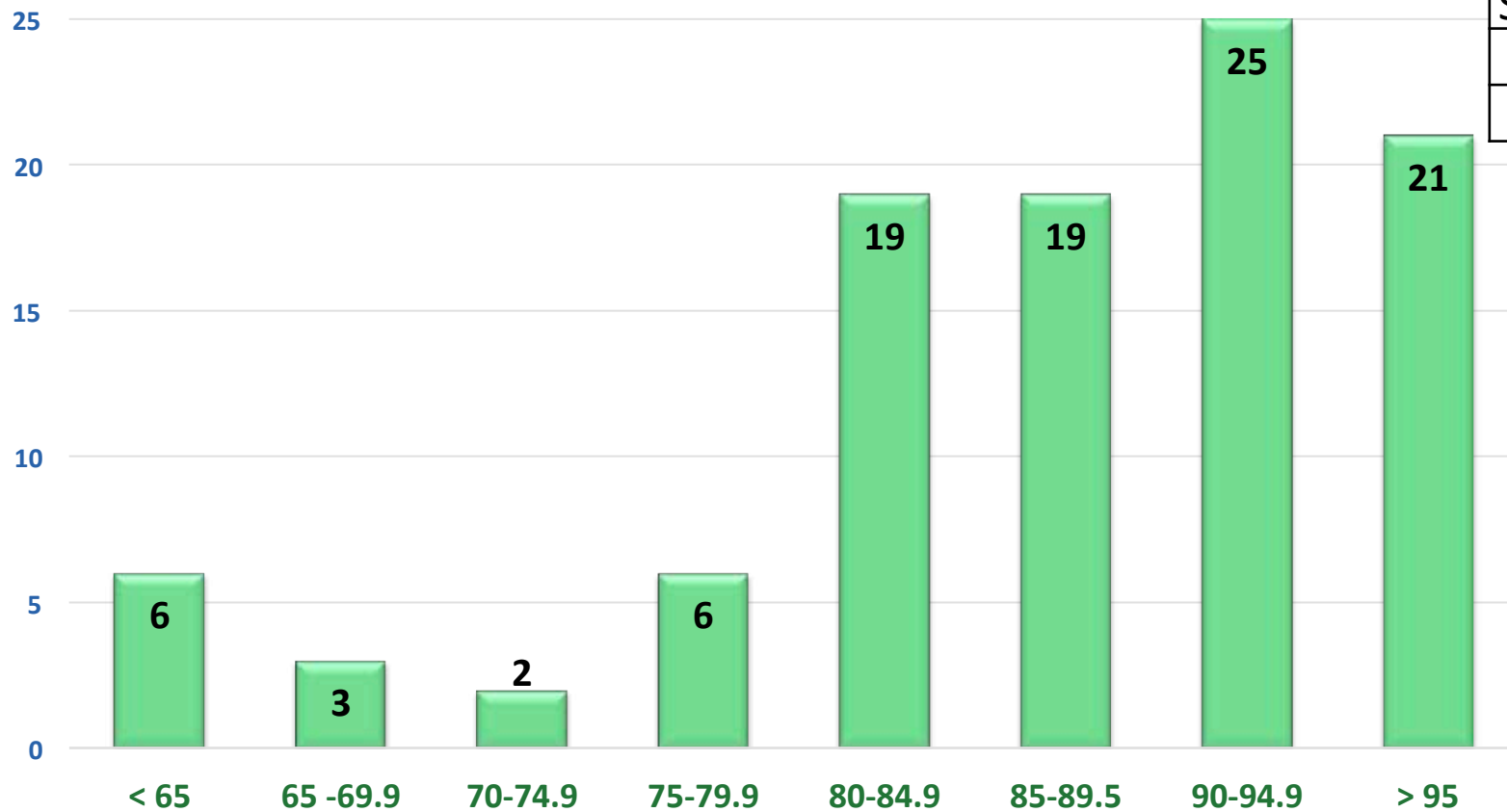
- Midterm# 1 Grades
- Review of key concepts
- Loop design help
 - *Ch. 3.3, 3.4 in the textbook*
- Debugging your code
 - *Ch. 5.4, 5.5 in the textbook*

Announcements

- **Midterm is graded!**
 - Grades online
- **Homework #7 due today**
- Lab #3 was due Monday, 5/1
- Lab #4 due today
- Homework #8 is out
- **Don't forget your TAs' and Instructor's office hours!! 😊**

Midterm #1 Results

Grade Distribution for Midterm #1
CS 16, Sp 17 (Matni)



Average	86.1
Median	89
Std. Dev.	10.1
Min	53
Max	102

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
t	1 Enter number of cents (or zero to quit):	t	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
t	1 Enter number of cents (or zero to quit):	t	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
t	1 Enter number of cents (or zero to quit):	t	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>>

A Review of Basic Concepts

If-Else vs. Switch-Case

If-Else conditional branches:

- Great for **variable conditions** that give you a **Boolean**
- Can use **any data type**
- Can do more complex branching

Switch statement branches:

- Great for **fixed data values** that give you a **return value**
 - i.e. Menu-style
- Cannot do Boolean on the case!
- Cannot use anything other than **int, char** or **enum**

A Review of Basic Concepts

If-Else Conditionals

```
if (AmountDesc == "Not a lot") {  
    cout << "This is a small amount";  
    p += (amount - 50);  
    r = calcInterest(p);  
}  
else {  
    cout << "This may be enough";  
    p += amount;  
    r = calcInterest(p - 50);  
}
```

1. Note the syntax
2. Why is this type of conditional NOT applicable to switch/case?
3. Note the coding style

A Review of Basic Concepts

If-Else Conditionals

```
if ( (amt > 0) && (amt <= 10) )
    cout << "This is between 1 and 10\n";

else if ( (amt > 10) && (amt <= 20) )
    cout << "This is between 11 and 20\n";

else {
    cout << "This is outside the range\n";
    cout << "Enter another number: ";
    cin >> num;
}
```

Why is it ok **NOT** to have {...} here???

Why is it **NECESSARY** to have {...} here???

1. Note the syntax
2. Why is this type of conditional **NOT** applicable to switch/case?
3. Note the coding style

A Review of Basic Concepts

Switch-Case Conditionals

```
int num;  
cout << "Gimme a number! ";  
cin >> num;
```

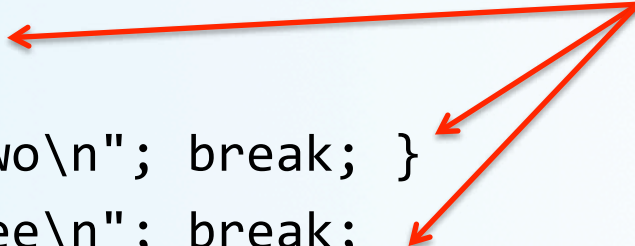
1. Note the syntax

2. Why does this type of conditional apply well to switch/case?

3. Note the coding style

```
switch (num) {  
    case 1:  
        cout << "One\n";  
        break;  
    case 2: { cout << "Two\n"; break; }  
    case 3: cout << "Three\n"; break;  
    default: cout << "Neither One, Two, nor Three\n";  
} // end switch
```

Which one is wrong?



A Review of Basic Concepts

while loops

```
int num(1);
while (num != 0) {
    cout << "Give me a number, or zero to quit: ";
    cin >> num;

    if (num != 0) cout << "Number x 10 = "
        << num * 10 << endl;

    else cout << "Quitting!\n";
}
```

Is the *logic* correct in the code?

1. Note the syntax
2. Why does this type of loop apply well to while loops?
3. Note the coding style

A Review of Basic Concepts *for loops*

Is the *logic* correct in the code?

```
int num = 25;
for (int j = 1; j <= 10; j++) {
    cout << "Loop run no." << j << endl;
    if ((num - 2*j) < 10) cout << "Condition 1 exists\n";
    else cout << "...nothing...";
}
```

1. Note the syntax
2. Why does this type of loop apply well to for loops?
3. Note the coding style

In-Class Exercise

A Review of Basic Concepts *for loops*

What does this code do?

```
int num1, num2, flag2(1), flag3(1);
cout << "Enter start, end numbers: ";
cin >> num1 >> num2;


for (int j = num1; j <= num2; j++) {
    cout << "The number " << j << endl;

    if (j % 2 == 0) cout << "Divisible by 2\n";
        else flag2 = 0;
    if (j % 3 == 0) cout << "Divisible by 3\n";
        else flag3 = 0;
    if (!(flag2 || flag3))
        cout << "Not divisible by either 2 or 3\n";

    flag2 = 1; flag3 = 1;
    cout << "-----" << endl;
}
```

1. Note the syntax
2. Why does this type of loop apply well to for loops?
3. Why is it better to use if/then here vs. switch?
4. Note the coding style

Why does this Boolean expression work?



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

- 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 $<=$?
- Check that the initialization uses the correct value
- Does the loop handle the zero iterations case?

Fixing Infinite Loops

- Check the direction of inequalities:
 < or > ?
- Test for < or > rather than equality (==)

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 to 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. $2 \times 3 \times 4 \times 5$, which, of course, is 120.
- What could go wrong?! 😊

```
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 retested
 - 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

Starting Over

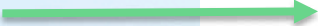
- Sometimes it is more efficient to throw out a buggy program and start over!
 - The new program will be easier to read
 - The new program is less likely to be as buggy
 - You may develop a working program faster than if you work to repair the bad code
 - The lessons learned in the buggy code will help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


To Dos

- Homework #8 for Thursday
- Lab #5

</LECTURE>