Today’s lecture

- Solving quadratic with if-statements
- General form of the if-statement
- Boolean tests (using relational operators)
- What about degenerate quadratics?
- Refining quadratic.c

Last Lecture

- Arithmetic
- Quadratic equation problem: \( ax^2 + bx + c = 0 \)
- Floating point data

Choice of variables for quadratic.c

- We will need to compute the square-root of \( b^2 - 4ac \).
- The \texttt{sqrt} function available in the math library for C is of the type \texttt{double sqrt(double x)};
- For this simple reason we use double variables for our real roots.
- Precision is not really important to us, at least not now.
C program to Solve Quadratic Equations

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

Steps of our program:
- Take in the inputs \(a, b\) and \(c\) from the user (\texttt{scanf}).
  
*Need three int variables to store these values: \(a, b, c\)* say;
- test whether \(b^2 - 4ac\) is non-negative.
  
  What we do will depend on the result of the test
  - If negative, output a message about “No real roots”.
  - If exactly 0, a repeated root.
  - Otherwise, two differing roots as per formula.
- Get the square root of \(b^2 - 4ac\) (if non-negative).
- Output both roots (or one if repeated).
- return EXIT_SUCCESS;

body of quadratic.c

```c
int s = (b*b - 4*a*c);
if (s < 0) {
    printf("No real roots to this quadratic.\n");
} else if (s == 0) {
    printf("Eq. has the repeated root \%.f.\n", -(double)b/(2.0*a));
} else {
    x1 = (-(double)b - sqrt(s))/(2.0*a);
    x2 = (-(double)b + sqrt(s))/(2.0*a);
    printf("The sols to %dx^2 +%dx +%d = 0 are ", a, b, c);
    printf("%lf and %lf.\n", x1, x2);
}
```

**Note:**
- sqrt() takes a double argument, so the int expression \(s\) is automatically promoted to a double in sqrt(s).
- Question: is the cast (double)b necessary?

Running quadratic.c

\begin{itemize}
  \item \texttt{quadratic.c} (and the refinements of this program) uses the \texttt{sqrt} function from the \texttt{math} library.
  \item \textbf{note:} Not enough to include \texttt{<math.h>} in the code.
\end{itemize}

- \texttt{<math.h>} is just a header file for the \texttt{math} library (it explains the “shape” of the \texttt{sqrt} function, and other \texttt{math} functions).
- To run our program, we need to \texttt{link} to executable code for the \texttt{math} functions.
- Link by adding \texttt{-lm} to \texttt{gcc} command when compiling:
  
  \begin{verbatim}
  gcc -Wall quadratic.c -lm
  \end{verbatim}

  \texttt{-lm} is ‘minus ell m’, NOT ‘minus one m’.

Assumptions :-(

We made some assumptions for quadratic.c

- By solving a quadratic, we (implicitly) assumed \(a\) is non-zero.
- Same for \(b\).
- We might have had a linear or constant equation.

\textbf{SOLUTION} - use the (general) if statement.
**If statement – general form**

```c
if ( condition1 ) {
    statement-sequence1
}
else if ( condition2 ) {
    statement-sequence2
}...
else {
    statement-sequence_n
}
```

- `condition1, ..., condition_{n-1}` are all boolean expressions: either true or false.
- `statement-sequence_1, ..., statement-sequence_n` are all sequences of C-programming statements.
- Note that it is possible to use `if` alone without any `else` branch.

**Warning about if**

If you look in the textbooks, you will see that when the `statement-sequence` has only one statement, you can miss out the curly brackets round it:

```c
if ( t > 0 )
    x = x + 1;
else
    x = x - 1;
```

We **recommend** that you don’t do this (at least until you’re experienced enough to understand when you can ignore our advice!).

You will see it in other people’s programs though.

Actually, we lied about the general form of the `if` statement. In truth, the general form is

```c
if ( condition ) statement1 else statement2
```

and a `statement` can have the form `{ statement-sequence }` or can be an `if`-statement itself. However, the form on the previous slide is the way we typically use it, so best to ‘learn’ that.

---

**Relational operators**

What kind of conditions can we use in `if` statements?

Assume `e_1` and `e_2` are (usually arithmetic) expressions . . .

We can apply **relational operators** to form a boolean expression.

- `e_1 == e_2`  \( e_1 \) equal to \( e_2 \)
- `e_1 != e_2`  \( e_1 \) not equal to \( e_2 \)
- `e_1 < e_2`  \( e_1 \) less than \( e_2 \)
- `e_1 <= e_2`  \( e_1 \) less than or equal to \( e_2 \)
- `e_1 > e_2`  \( e_1 \) greater than \( e_2 \)
- `e_1 >= e_2`  \( e_1 \) greater than or equal to \( e_2 \).

**Never write** `e_1 < e_2 < e_3` if you remember to `-Wall`, the compiler will warn you if you do this!

**Note**: We can compare `float` and `double` expressions in this way - but only `int` comparisons are fully reliable. **Why is this?**

---

**More complicated Boolean expressions**

Assume `e_1` and `e_2` are boolean expressions . . .

Can build more complicated boolean expressions iteratively, using **boolean operators**.

- `0` false (always)
- `!e_1` true (always)
- `e_1 && e_2` true if \( e_1 \) is false
- `e_1 || e_2` true if \( e_1 \) is true and \( e_2 \) is true
- `e_1 > e_2` true if \( e_1 \) is true or \( e_2 \) is true

For example, can write \( ( e_1 < e_2 ) && ( e_2 < e_3 ) \).

The expressions `e_1`, `e_2` are (formally) integer expressions.

We define **boolean variables** as `int`.

Think of integers as (informally) acting as boolean ‘type’.
Boolean expressions in if-else statements

We have seen lots of simple and complex boolean expressions: whenever any of these are used as tests in a if-else statement, they must be enclosed in parentheses.

For example

→ The simple Boolean expression \( x < 5+z \), when being used as a test for an else if-branch of an if-else statement, would appear as

\[
\text{else if } (x < 5+z) \{ \ldots \}
\]

→ The complex Boolean expression \((a != 0) && (b*b > 4*a*c)\), when being used as the test for the if branch of an if-else statement, would appear as

\[
\text{if } ( (a != 0) && (b*b > 4*a*c) ) \{ \ldots \}
\]

Improving quadratic.c

Issues for improving quadratic:

(i) make our quadratic solver cope with \( a = 0 \) (quadratic2.c)

... and with \( a = 0, b = 0 \).

(ii) complex roots (homework).

Nested if-statements

→ The statement-sequence place-holder in the general if-statement allows other if-statements to be part of the program fragment.

→ This is a 'nested' use of the if-statement.

→ Example – refine the quadratic.c program further to include a solution for the \( a = 0 \) case (given a linear equation).

quadratic equations – what if \( a = 0 \)

If \( ax^2 + bx + c = 0 \) is a quadratic, and \( a \) is 0, then we have a linear equation:

\[
bx + c = 0
\]

This has ...

→ Exactly one root of value \(-c/b\), if \( b \neq 0 \).

→ No root at all, if \( b = 0 \) and \( c \neq 0 \).

→ Everything is a root, if \( b = c = 0 \).

Can incorporate this case into our code:
Start off as before ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h> // Need to include math.h to use sqrt.

int main(void) {
    int a, b, c, s;
    double x1, x2;

    printf("Input the x^2 co-efficient a: ");
    scanf("%d", &a);
    printf("Input the x co-efficient b: ");
    scanf("%d", &b);
    printf("Input the constant term c: ");
    scanf("%d", &c);
    s = b*b - 4*a*c;
```

We need to complete the else (a being 0) branch.

```c
else if (b != 0) { /* a==0 WITH b NON-ZERO */
    x1 = -((double)c)/((double)b);
    printf("1 sol to %dx^2 +%dx +%d = 0.\n", a, b, c);
    printf("It is %lf.\n", x1);
}
else if (c != 0) { /* a AND b BOTH ZERO, c NON-ZERO */
    printf("No sols to %dx^2 + %dx +%d = 0.\n", a, b, c);
}
else { /* a, b, c ALL ZERO */
    printf("Degenerate equation - everything is a solution!\n");
    return EXIT_SUCCESS;
}
```

We need to complete the else (a being 0) branch.

```c
else if (s < 0) {
    /* THIS (a != 0) */
    ...
    /* BRANCH IS */
    }
else if (s == 0) {
    /* EXACTLY WHAT */
    /* WE PUT FOR THE */
    ...
    /* BODY OF */
    }
else {
    /* quadratic.c */
    ...
    /* quadratic.c */
}
else {
    /* THIS WILL BE THE */
    ...
    /* SOLUTION FOR a==0 */
}
/* (linear equations) */
return EXIT_SUCCESS;
```