# Selection Statements

Chapter 5

# Objectives

For you will be able to

- Use relational operators, such as < and ==
- Use logical operators, &&, ||, and !
- Write if ... else ... statements correctly

# Relational Operators

x < y

- < is an *operator*
  - x and y are its operands
- ( x < y ) is called a *logical* expression.
  - Logical as in “pertaining to logic”
  - Yields a value of either true or false
    - 1 for true
    - 0 for false

# Relational Operators

- The complete set of relational operators:

| | |
|---|---|
| x < y | less than |
| x <= y | less than or equal |
| x > y | greater than |
| x >= y | greater than or equal |
| x == y | equal |
| x != y | not equal |

# Relational Operators

## Relational operators are often used in an "if" statement

```
#include <stdio.h>
int main()
{
    double t;      /* elapsed time */
    printf ("Please enter elapsed time in seconds: ");
    scanf("%lg", &t);

    /* Check for valid data */
    if (t <= 0)
    {
        printf ("Please enter a time value greater than 0\n");
        return 1;
    }

    printf ("t = %f\n", t);
    return 0;
```

This block is executed only if the condition following “if” is true.

# The Simple `if` Statement

- Things to notice

The condition MUST be enclosed in parentheses.

No semicolon following the if statement.

```
if (t < 0)
{
    printf ("Please enter a time value greater than 0\n");
    exit (1);
}
```

No semicolon following final curly bracket.

Curly brackets delimit the block to be executed if the condition is true.

# Some matters of programming style

The textbook sometimes puts the opening curly bracket on the same line as the if statement.

```
if (t < 0) {
    printf ("Please enter a time value greater than 0\n");
    exit (1);
}
```

- This is widely accepted style for “if” statements.

# Some matters of programming style

- If there is only one statement to be conditionally executed, the curly brackets may be omitted.

```
if (t < 0)
    printf ("Please enter a value greater than 0\n");
```

Or even

```
if (t < 0) printf ("Please enter a value greater than 0\n");
```

- Easy for a reader to be confused.
- Easy to introduce errors later when you need to add more statements to the conditional part.

# Some matters of programming style

Take it as an ironclad rule:

Every “if” must be followed by a block delimited by curly brackets

even if there is only one statement.

```
if (condition)
{
   /* Stuff to do if condition is true */
}
```

**The code inside the curly brackets is indented three spaces beyond the brackets.**

# Some matters of programming style

- Be aware: Programming style is important!

- Points will be deducted on projects and exams if the style is unreadable
  - even if the program is functionally correct.

- You will probably have to adapt to different coding standards in other courses and throughout your career.

# Logical Values

- In C *every* ***number*** has a logical value, in addition to its numerical value.
  - 0 represents false.
  - Any nonzero value is considered true.

```
scanf("%lg", &t);
if (t)
{
    printf ("t is true\n");
    return 1;
}
```

- This is legal C, and a widely used idiom.

- But not good programming.
  - It is better to use a logical expression with “if”.
  - Example: `if (t != 0)`

# Logical Values

- In C89, numerical values are the *only* way to represent truth value.
- C99 added the _Bool type for this purpose.
  - Discussed in the textbook.
- A _Bool variable is an integer type, but can only take on the values 0 and 1.
- Only mentioned for your information.  We will **NOT** use _Bool in this class

# Logical Values in C99

- C99 also provides a standard header file <stdbool.h>

- Defines with **`#define`**
  - bool as _Bool
  - true as 1
  - false as 0

# Recommendation

- Stick to the old way.
- Your program will compile correctly on systems using older compilers.
  - _Bool is ugly
  - Not a real boolean type, as in Java and C#
  - Still works as an int

# Relational Operators

- A frequent mistake : confusing = and ==

This compiles without error, and gives no error indication at run time.

```
if (x = 1)
{
        /* Do something */
}
```

But it's not what you meant!

The conditional block will always be executed!

This sets x to 1 and yields a value of 1, which means "true" to the if.

# Relational Operators

## Should have said:

```
if (x == 1)
{
        /* Do something */
}
```

```
-bash-3.00$ cat test.c
#include <stdio.h>
int main()
{
    int a = 1;
    int b = 2;

    printf ("a is %d and b is %d\n", a, b);

    if (a = b)
    {
        printf ("a and b are equal\n");
    }
    else
    {
        printf ("a and b are unequal\n");
    }
    return 0;
}
-bash-3.00$ gcc -Wall *.c
test.c: In function ‘main’:
test.c:9: warning: suggest parentheses around assignment used as truth va
-bash-3.00$./a.out
a is 1 and b is 2
a and b are equal
-bash-3.00$
```

```
-bash-3.00$ cat test.c
#include <stdio.h>
int main()
{
    int a = 1;
    int b = 0;

    printf ("a is %d and b is %d\n", a, b);

    if (a = b)
    {
        printf ("a and b are equal\n");
    }
    else
    {
        printf ("a and b are unequal\n");
    }
    return 0;
}
-bash-3.00$ gcc -Wall *.c
test.c: In function ‘main’:
test.c:9: warning: suggest parentheses around assignment used as truth va
-bash-3.00$./a.out
a is 1 and b is 0
a and b are unequal
-bash-3.00$
```

# Using Logical Expressions

- The most common use of logical expressions is in conditional statements like “if”.

```
if (x < y)
{
   /* Do something */
}
```

# Using Logical Expressions

- But the expression (x < y) has a *value*.
  - 0 or 1
- The value can be stored in an integer variable.

```
int x_ok;
...
x_ok = (x < y);
```

# Using Logical Expressions

```
x_ok = (x < y);
```

is equivalent to

```
if (x < y)
{
    x_ok = 1;
}
else
{
    x_ok = 0;
}
```

# Logical Operators

- You can then say:

```
if (x_ok)
{
   /* Do something */
}
```

# Logical Values

- Many C programmers write:

```
#define false 0
#define true 1
```

as in the C99 header file <stdbool.h>

You can then write:

```
    x_ok = true;
```

or

```
    x_ok = false;
```

# Logical Values

- But don’t say:

```
if (x_ok == true)
{
  /* Do something */
}
```

This is correct C

but “== true” is redundant

- Just write:

```
if (x_ok)
{
  /* Do something */
}
```

# Logical Values

- Possible confusion with #define true 1

```
if (x_ok == true)
{
  /* Do something */
}
```

If x_ok is 2 (or any other nonzero value other than 1)

`x_ok` is *true*

but

`(x_ok == true)` is *false*.

# Logical Values

Remember:

== compares numerical values

not logical values!

End of Section

# Logical Operators

- Operators that “do logic”

| | |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

- Permit us to combine logical expressions like (a < b) and (b < c) into a single bigger expression.

# Logical Operator &&

- Example:

```
if ((x < y) && (y < z))
{
  /* Do something. */
}
```

- The conditional block will be executed if x is less than y AND y is less than z.
- Otherwise it will be skipped.

# Example

- Determine if x, y, and z are in increasing order:

```
if ((x < y) && (y < z))
```

```c
#include <stdio.h>
int main()
{
    int x,y,z;
    printf ("Please enter integers x, y, and z in increasing "
            "order\n");

    printf ("x: ");
    scanf("%d", &x);
    printf ("y: ");
    scanf("%d", &y);
    printf ("z: ");
    scanf("%d", &z);

    if ((x < y) && (y < z))
    {
        printf ("x, y, and z are in increasing order\n");
    }
    else
    {
        printf ("x, y, and z are not in increasing order\n");
    }

    return 0;
}
```

# Logical Operator ||

- Example:

```
if ((x < y) || (y < z))
{
  /* Do something. */
}
```

- The block will be executed if x is less than y  OR  y is less than z

  (including the case where both are true.)

- If neither condition is true, it will be skipped.

# Logical Operator !

- Example:

```
if (!x_ok)
{
  /* Do something. */
}
```

- The conditional block will be executed if x_ok is false.
- If x_ok is true it will be skipped.

# Logical Operators

- ! is a *unary* operator.
- Like all unary operators it has very high precedence.
  - Unary operators stick to the thing beside them.
  - All apply to the thing to their right
    - except x++ and x- -
  - You don’t need parentheses to make the meaning clear.

# Logical Operators

- && and || are binary operators.
  - Have relatively low precedence

- All arithmetic and relational operators are applied first.
  - && and || are applied to the results.

a < b && b < c means (a < b) && (b < c)

  - Use parentheses for readability!

# Logical Operators

- && has higher precedence than ||

```
a < c  ||  b < c  &&  c < d
```

means

```
(a < c)  ||  ((b < c) && (c < d))
```

This would not be obvious to most people.

Use parentheses to make the meaning clear.

# The **if . . . else** Statement

You can write an alternative block of code to be executed if the condition is *not* true.

```
if (condition)
{
    /* Stuff to do if condition is true */
}
else
{
    /* Stuff to do if condition is NOT true */
}
```

- Use this alignment style rather than the one shown in the textbook.
  - Align the opening and closing curly brackets.
  - Indent everything inside the brackets.

# A Series of `if ... else` Statements

- We often have several conditions
  - Need to execute one of several corresponding blocks of code according to which condition is true.

# A Series of `if ... else` Statements

```
if (condition_1)
{
   /* Stuff to do if condition_1 is true */
}
else if (condition_2)
{
   /* Stuff to do if condition_1 is false
      and condition_2 is true.         */
}
else
{
   /* Stuff to do neither condition_1
      nor condition_2 is true.         */
}
```

Exactly one of these code blocks will be executed.

# A Series of **`if ... else`** Statements

```
if (condition_1)
{
   /* Stuff to do if condition_1 is true */
}
else if (condition_2)
{
   /* Stuff to do if condition_1 is false
      and condition_2 is true.        */
}
else
{
   /* Stuff to do neither condition_1
      nor condition_2 is true.        */
}
```

If condition_1 is true, condition_2 has no effect.
Only the first block is executed.

# A Series of `if ... else` Statements

- Note that (in general) this is not the same as a series of independent if statements.
  - Different results if more than one of the conditions can be true at the same time.

# Table Lookup

- Example: Using a series of if…else statements to do a table lookup.

# Table Lookup

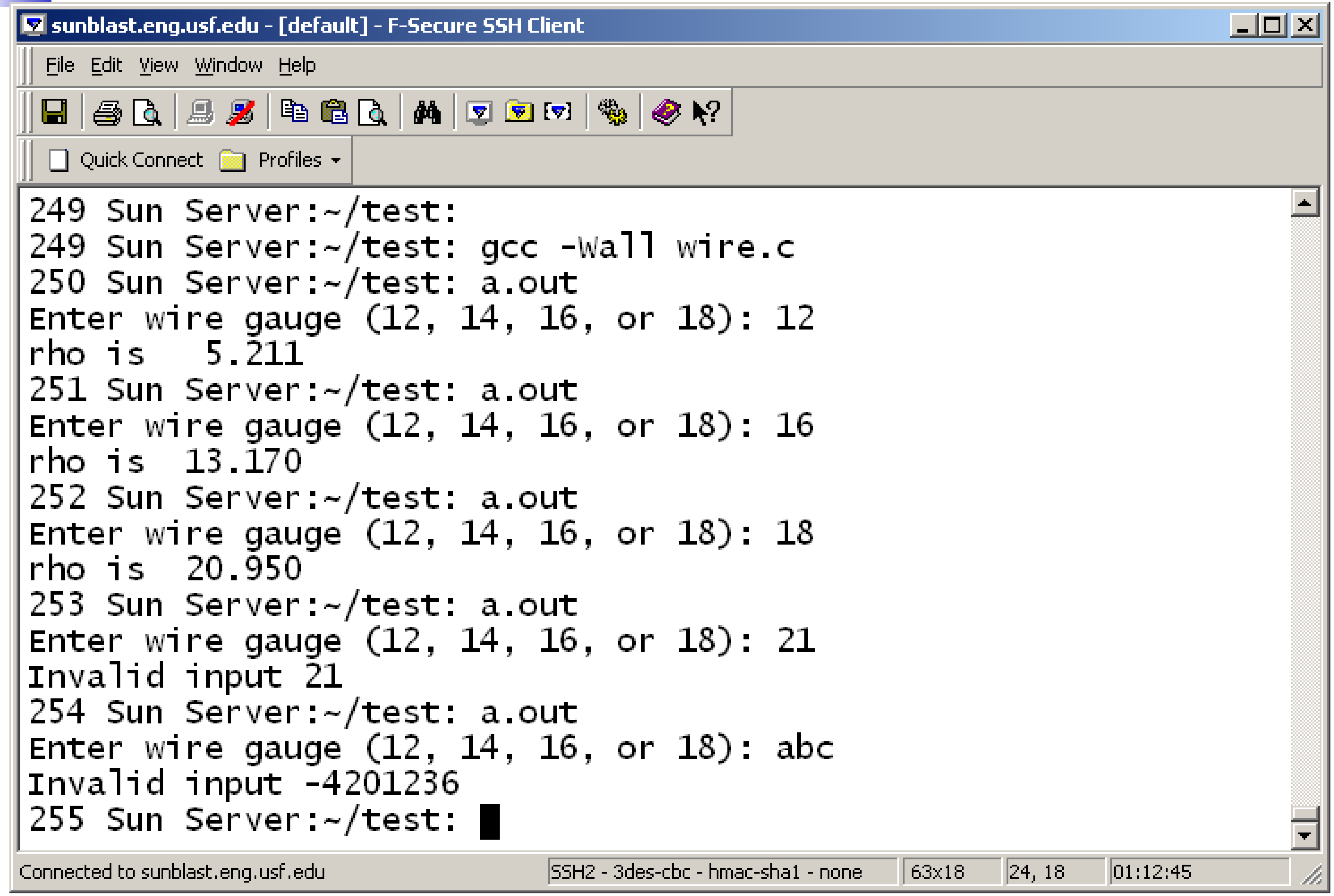
- Resistivity, rho, of electrical wire varies with the “gauge” of the wire.

| Gauge | Rho |
|---|---|
| 12 | 5.211 |
| 14 | 8.285 |
| 16 | 13.170 |
| 18 | 20.950 |

Write a program that prompts the user to enter a wire gauge and outputs the resistivity of that gauge.

```c
#include <stdio.h>
int main()
{
    int gauge;
    double rho;
    printf ("Enter wire gauge (12, 14, 16, or 18): ");
    scanf("%d", &gauge);
    if (gauge == 12)
    {
        rho = 5.211;
    }
    else if (gauge == 14)
    {
        rho = 8.285;
    }
    else if (gauge == 16)
    {
        rho = 13.170;
    }
    else if (gauge == 18)
    {
        rho = 20.950;
    }
    else
    {
        printf ("Invalid input %d\n", gauge);
        return 1;
    }
    printf ("rho is %7.3f\n", rho);
    return 0;
}
```

# Program wire.c Running



sunblast.eng.usf.edu - [default] - F-Secure SSH Client
File Edit View Window Help
Quick Connect
Profiles

```
249 Sun Server:~/test:
249 Sun Server:~/test: gcc -Wall wire.c
250 Sun Server:~/test: a.out
Enter wire gauge (12, 14, 16, or 18): 12
rho is   5.211
251 Sun Server:~/test: a.out
Enter wire gauge (12, 14, 16, or 18): 16
rho is  13.170
252 Sun Server:~/test: a.out
Enter wire gauge (12, 14, 16, or 18): 18
rho is  20.950
253 Sun Server:~/test: a.out
Enter wire gauge (12, 14, 16, or 18): 21
Invalid input 21
254 Sun Server:~/test: a.out
Enter wire gauge (12, 14, 16, or 18): abc
Invalid input -4201236
255 Sun Server:~/test: █
```

Connected to sunblast.eng.usf.edu | SSH2 - 3des-cbc - hmac-sha1 - none | 63x18 | 24, 18 | 01:12:45

# Nested “if” Statements

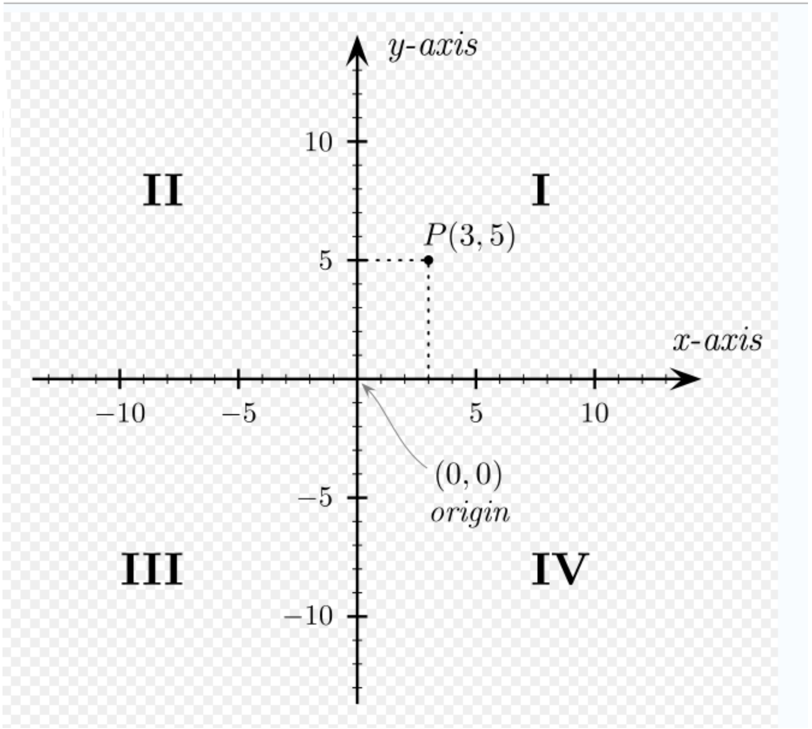
- We can put any legal C statements inside the code block following an “if” statement
  - Including other “if” statements.

- The nesting can continue indefinitely
  - But *should* not go beyond three levels.
  - The C compiler can handle deep nesting but humans cannot.

# Example: Which Quadrant?

- The x-y plane can be divided into four *quadrants*.

http://en.wikipedia.org/wiki/Cartesian_coordinate_system#Two-dimensional_coordinate_sys



# Example: Which Quadrant?

- Given a point, (x,y), determine its quadrant.
  - Include zeroes with the positive values.

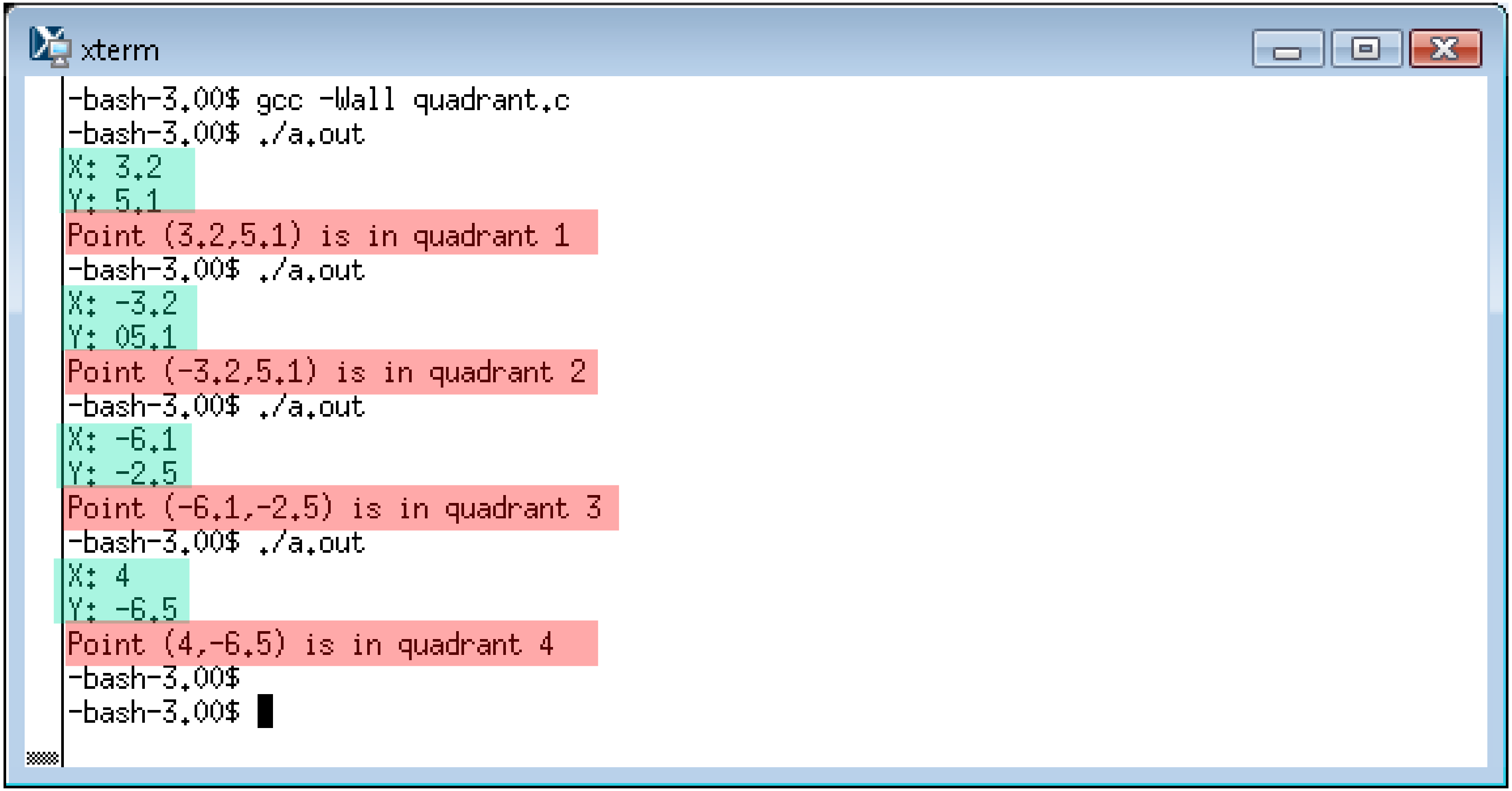
# quadrant.c

```
#include <stdio.h>

int main()
{
    double x = 0.0;
    double y = 0.0;
    int quadrant = 0;

    printf ("X: ");
    scanf ("%lg", &x);
    printf ("Y: ");
    scanf ("%lg", &y);
```

# Nested “if” Statements

```
    if (x >= 0.0)
    {
        if (y >= 0.0)
        {
            quadrant = 1;
        }
        else
        {
            quadrant = 4;
        }
    }
    else
    {
        if (y >= 0)
        {
            quadrant = 2;
        }
        else
        {
            quadrant = 3;
        }
    }
    printf ("Point (%lg,%lg) is in quadrant %d\n", x, y, quadrant);
    return 0;
}
```

# Program quadrant.c Running



```
-bash-3.00$ gcc -Wall quadrant.c
-bash-3.00$ ./a.out
X: 3.2
Y: 5.1
Point (3.2,5.1) is in quadrant 1
-bash-3.00$ ./a.out
X: -3.2
Y: 05.1
Point (-3.2,5.1) is in quadrant 2
-bash-3.00$ ./a.out
X: -6.1
Y: -2.5
Point (-6.1,-2.5) is in quadrant 3
-bash-3.00$ ./a.out
X: 4
Y: -6.5
Point (4,-6.5) is in quadrant 4
-bash-3.00$
-bash-3.00$ 
```

# The "Dangling Else" Problem

Nested “if” statements can lead to confusion about the “else”

```
if (x > 0)
     if (y > 0)
        sum = x + y;
else
    printf ("Invalid input value\n");
```

Which "**if**" does the "**else**" go with?

Answer: The inner one

Avoid confusion by *always* putting curly brackets after the "if".

# The "Dangling Else" Problem

```
if (x > 0)
{
    if (y > 0)
    {
        sum = x + y;
    }
    else
    {
        printf ("Invalid input value\n");
    }
}
```

If this is what you mean.

# The "Dangling Else" Problem

Or

```
if (x > 0)
{
    if (y > 0)
    {
        sum = x + y;
    }
}
else
{
    printf ("Invalid input value\n");
}
```

If this is what you mean.

# The "Dangling Else" Problem

- If you want the error message if *either* condition is false, write

```
if ((x > 0) && (y > 0))
{
     sum = x + y;
}
else
{
     printf ("Invalid input value\n");
}
```

# Lazy Evaluation

```
if ((x > 0) && (y > 0))
{
    /* Do something. */
}
```

If x is not greater than 0, we don’t need to look at y.

We already know that `((x > 0) && (y > 0))` is false

*regardless of the value of y*.

## Lazy Evaluation

- The C compiler takes advantage of this.
  - Skip the evalutation of the second expression if the first one is false.
  - Saves some CPU time.
  - A convenient way to avoid dividing by 0

```
if ( (x != 0) && (b/x < 0.1) )
{
    /* Do something. */
}
```

# Lazy Evaluation

Consider the case:

```
if ((x < 0) && (y++ < 0))
{
   /* Do something. */
}
```

Will y be incremented?

- Only if x is less than 0!

An example of how side effects can cause trouble.

# Lazy Evaluation

- In the case of ||, the second expression will not be evaluated if the first one is *true*.

## Summary – Lazy Evaluation

Work left to right, and only evaluate what you have to evaluate in order to determine the overall result.

End of Section

# The Conditional Operator

- An expression that will have one of two possible values depending on a condition.

- C's only ternary (three operand) operator

- Example:
  - $\underbrace{(i >= j)}_{\text{Condition}}\ ?\ \underset{\underset{\text{Value if condition is true}}{\uparrow}}{i}\ :\ j \longleftarrow \text{Value if condition is false}$

# The Conditional Operator

```
x = (i >= j) ? i : j;
```

is equivalent to

```
if (i >= j)
{
  x = i;
}
else
{
  x = j;
}
```

- More concise.   Less understandable.

# Assignment 1

- You can now start working on Assignment 1
- Look at scanf_error_check.c in the Tutorial folder as it is useful for this assignment