cisc1110 fall 2010 lecture V.1

- *control structures*
- making decisions
- branching statements
- relational operators
- comparing numbers
- comparing strings

making decisions

- *branching statements* are used to allow computer programs to *make decisions*
- `if` a statement is true, then do one thing; otherwise, do something else
- you make decisions like this all the time:

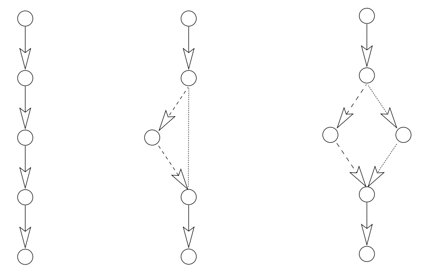
```
If the 5 train is in Nevins St station when my 2 train arrives,
then I will run across the platform and catch the 5 train to Flatbush;
otherwise, I will stay on the 2 train
```

- a computer program can make the same types of decisions
- and frequently these are made using relational operators...
- example:

```
if ( x > y ) {
  cout << "x is bigger than y\n";
}
else {
  cout << "y is bigger (or the same as x)\n";
}
```

branching statements

- the `if` statement is part of the C++ language. it is a type of *control structure*, which means that the program control can move from one "branch" to another, instead of always taking a single path.



- there are three forms of the `if` statement in C++:
(1) simple `if`, (2) `if-else`, and (3) `if-else-if`

relational operators

- relational operators are used to compare two values
- they can be used to compare numbers or characters
- comparing characters uses the ASCII table (remember asciimation?)
- the relational operators look like operators in math, except for equality:

==	equality	!=	inequality
>	greater than	<	less than
>=	greater than or equal to	<=	less than or equal to

- examples:

```
x < y
a > b
```

- relational operators are used as part of statements
- one kind of statement is a *branching statement*...

the simple `if` statement

- syntax:

```
if ( <something is true> ) {
  <follow some instructions>
}
```

- example:

```
if ( x > y ) {
  cout << "x is bigger than y\n";
}
```

the `if-else` statement

- syntax:

```
if ( <something is true> ) {
  <follow some instructions>
}
else {
  <follow some other instructions>
}
```

- example:

```
if ( x > y ) {
  cout << "x is bigger than y\n";
}
else {
  cout << "y is bigger (or the same as x)\n";
}
```

the `if-else-if` statement

- syntax:

```
if ( <something is true> ) {
  <follow some instructions>
}
else if {
  <follow some other instructions>
}
else if {
  <follow other, different instructions>
}
else {
  <follow even different instructions>
}
```

- example:

```
if ( x > y ) {
  cout << "x is bigger than y\n";
}
else if ( y > x ) {
  cout << "y is bigger\n";
}
else {
  cout << "y is the same as x\n";
}
```

comparing numbers

```
int x;
if ( x <= 0 ) {
  cout << "x is less than or equal to 0\n";
}
else {
  cout << "x is greater than 0\n";
}

//--------------------  OR  --------------------

double y;
if ( y <= 0 ) {
  cout << "y is less than or equal to 0\n";
}
else {
  cout << "y is greater than 0\n";
}
```

comparing strings

- the *comparison* operators also work with strings (==, <, <=, >, >=)
- the double equals sign (==) compares the value of two strings and returns `true` if they are the same, e.g.:

```
string s1, s2, s3;
bool a1, a2;
s1 = "david ";
s2 = "ortiz";
s3 = "david ";
a1 = ( s1 == s2 );
a2 = ( s1 == s3 );
```

  After the above code fragment:
  the value of `a1` will be `false`
  and
  the value of `a2` will be `true`

- the inequality operators (<, <=, >, >=) perform a *lexical comparison* between two strings
- a "lexical comparison" is like checking if two strings are in alphabetical order: one is less than the other if it comes before the other alphabetically
- EXCEPT, the lexical comparison is *case sensitive* and uses the ASCII table, which means that all the upper case letters (`A..Z`) come before (are less than) all the lower case letters (`a..z`), e.g.:

```
string s1, s2, s3;
bool a1, a2;
s1 = "ABC";
s2 = "DEF";
s3 = "abc ";
a1 = ( s1 < s2 );
a2 = ( s3 < s2 );
```

  After the above code fragment:
  the value of `a1` will be `true` because `"ABC" < "DEF"`
  and
  the value of `a2` will be `false` because `"abc" > "DEF"`

- NOTE that you CANNOT use relational operators with C style strings (the reason why has to do with something called *pointers* and *memory addresses*—topics that are covered in the next semester)
- Instead, you have to use the `strcmp()` function, e.g.:

```
#include <cstring>
...
char cs1[] = "ABC", cs2[] = "DEF", cs3[] = "abc "; // c style strings
...
a1 = ( strcmp( cs1, cs2 ) < 0 );
a2 = ( strcmp( cs3, cs2 ) < 0 );
a3 = ( strcmp( cs1, cs3 ) < 0 );
```

- the `strcmp()` function is in the `cstring` library so you have to `#include <cstring>` to use it
- it compares two string arguments: `strcmp( s1, s2 )` and returns:
  a value > 0 if `s1` $>$ `s2`
  a value < 0 if `s1` $<$ `s2`
  a value $== 0$ if `s1` $==$ `s2`

- you can also use `strncmp()` function, also in the `cstring` library, which compares the first `n` characters in both strings:
  `strncmp( s1, s2, n )`
  it has the same return values as `strcmp()`

example: finding the smallest element in the array

```
int smallest;
smallest = a[0];
for ( i=1; i<100; i++ ) {
  if ( a[i] < smallest ) {
    smallest = a[i];
  }
} // end for
cout << "the smallest value in the array is: " << smallest << endl;
```