More Boolean and Relational Topics

Class 14

Overlapping

- consider the bank loan rate, but with slightly different rules
- illustrated by this two-dimensional chart

		Employed	
		No	Yes
Grad	No	good	better
	Yes	better	best

- because the cases overlap, you could use a nested-if structure
- but it's simpler and more natural to use a logical operator

Logical Operators

- C++ has three logical operators
- and
  - && logical-and is a binary operator
  - its operands are both Boolean values
  - its return value is Boolean
- or
  - || logical-or is a binary operator
  - its operands are both Boolean values
  - its return value is Boolean
- not
  - ! logical-not is a unary operator
  - its operand is a Boolean value
  - its return value is Boolean

Truth Tables

- computer scientists use truth tables to explain the behavior of logical operators

a	b	a && b
f	f	f
f	t	f
t	f	f
t	t	t

a	b	a || b
f	f	f
f	t	t
t	f	t
t	t	t

a	!a
f	t
t	f

Using Logical Operators

- for the bank loan, in English
- if the customer is neither employed nor graduated, the rate is good
- if the customer is employed or graduated, but not both, the rate is better
- if the customer is both employed and graduated, the rate is best

Using Logical Operators

- for the bank loan, in English
- if the customer is neither employed nor graduated, the rate is good
- if the customer is employed or graduated, but not both, the rate is better
- if the customer is both employed and graduated, the rate is best

- this can be expressed in C++ in various ways

Loan Version 1

```
if (graduated == 'Y' && employed == 'Y')
{
  rate = "best";
}
else if (graduated == 'Y')
{
  rate = "better";
}
else if (employed == 'Y')
{
  rate = "better";
}
else
{
  rate = "good";
}
```

Loan Version 2

```
if (graduated == 'Y' && employed == 'Y')
{
  rate = "best";
}
else if (graduated == 'Y' || employed == 'Y')
{
  rate = "better";
}
else
{
  rate = "good";
}
```

Loan Version 3

```
if (graduated != 'Y' && employed != 'Y')
{
  rate = "good";
}
else if (graduated != 'Y' || employed != 'Y')
{
  rate = "better";
}
else
{
  rate = "best";
}
```

Or

- in English, or can have two different meanings
  1. do you want fries or onion rings with that burger?
  2. to help as a volunteer, you can carry boxes or set up signs
- in case 1, you can choose fries or onion rings, but not both
- in case 2, you can help by carrying boxes, by setting up signs, or by doing both
- case 1 is called exclusive or
- case 2 is called inclusive or
- the C++ || logical or operator is the inclusive or operator
- C++ does not have a logical exclusive or operator
- (it does have a bitwise exclusive or operator, which you will study later)

Equality of Characters and Strings

- testing the equality of characters is obvious
- one letter is either exactly the same as another, or not
- `'A' == 'A'` is true while `'A' == 'a'` is false

Equality of Characters and Strings

- testing the equality of characters is obvious
- one letter is either exactly the same as another, or not
- `'A' == 'A'` is true while `'A' == 'a'` is false

- two string objects are equal if they have exactly the same characters in exactly the same order, and are the same length
- assuming `string word = "Extra";` has been initialized:
- `word == "Extra"` is true
- `word == "Extravagant"` is false

Order of Characters

- the less-than and greater-than relops apply to characters
- based on the ASCII chart (Gaddis page 1287)
- a character that is listed in the chart is less than one that is listed later, and greater than one that is listed earlier

Order of Characters

- the less-than and greater-than relops apply to characters
- based on the ASCII chart (Gaddis page 1287)
- a character that is listed in the chart is less than one that is listed later, and greater than one that is listed earlier

- thus, `'a'` $<$ `'b'` is true
- and `'a'` $>$ `'A'` is true — this one is not intuitive
- and `'a'` $>$ `'9'` is true — also not intuitive

Order of Characters

- the less-than and greater-than relops apply to characters
- based on the ASCII chart (Gaddis page 1287)
- a character that is listed in the chart is less than one that is listed later, and greater than one that is listed earlier

- thus, `'a'` $<$ `'b'` is true
- and `'a'` $>$ `'A'` is true — this one is not intuitive
- and `'a'` $>$ `'9'` is true — also not intuitive

- but you should know its general structure
  - a bunch of non-printable characters
  - the space character and then some symbols
  - the digits 0 – 9
  - more symbols
  - the upper case letters
  - more symbols
  - the lower case letters
  - a few more symbols

Comparing Strings

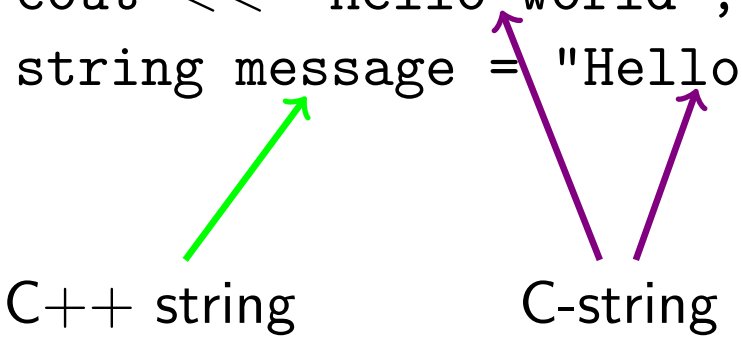
- the less-than and greater-than relops apply to string objects
- strings are compared character-by-character down the length of both strings
- the first character at which they differ determines which string is less-than the other
- if two strings have different lengths and contain exactly the same characters as far as the shorter one, the shorter one is less than the larger

f	o	o	t				
f	o	o	t	b	a	l	l

Comparing Strings Warning

- recall that C++ has two different things called "string"
  1. the old-fashioned C-strings that we wish we could ignore, but can't
  2. the easy-to-use newfangled C++ strings that we always use when we can
- the main place we've seen C-strings so far is as string literals

```
cout << "Hello world";
string message = "Hello world";
```



Comparing Strings Warning

- you can compare two C++ strings using relops
- you can compare a C++ string and a C-string using relops (because the C-string value is converted to a C++ string before comparison)
- but you cannot compare two C-strings using relops
- thus every statement in Gaddis Checkpoint 4.23 on page 200 is illegal!

Comparing Strings Warning

- you can compare two C++ strings using relops
- you can compare a C++ string and a C-string using relops (because the C-string value is converted to a C++ string before comparison)
- but you cannot compare two C-strings using relops
- thus every statement in Gaddis Checkpoint 4.23 on page 200 is illegal!

- assuming `string first = "Bill";`
  and `string last = "Monroe";`
  - legal: `first == last`
  - legal: `first >= "Bill"`
  - illegal: `"Fred" < "Bill"`

The Conditional Operator

- a very common programming construct has this structure:
- this construct will definitely give y a value
- the value will be either the opposite of x or double x, depending
- this is so common that there is a shortcut form

```
if (x < 0)
{
  y = x * -1;
}
else
{
  y = 2 * x;
}
```

```
y = x < 0 ? x * -1 : 2 * x;
```

The Conditional Operator

```
y = x < 0 ? x * -1 : 2 * x;
```

- the conditional operator is ternary
  1. first part: a Boolean expression
  2. second part: value if the Boolean expression is true
  3. third part: value if the Boolean expression is false

Flag Variables

Flag

A Boolean variable that stores the state of some condition in the program. When set to true, it indicates the condition exists. When set to false, it indicates the condition does not exist.

Flag Variables

Flag

A Boolean variable that stores the state of some condition in the program. When set to true, it indicates the condition exists. When set to false, it indicates the condition does not exist.

- in a program that calculates sales commissions, we can create the flag variable:

```
bool sales_quota_met = sale_amount >= QUOTA;
```

- later in the program we can check the state of the flag:

```
if (sales_quota_met)
{ ...
```

Flag Variables

Flag

A Boolean variable that stores the state of some condition in the program. When set to true, it indicates the condition exists. When set to false, it indicates the condition does not exist.

- in a program that calculates sales commissions, we can create the flag variable:

```
bool sales_quota_met = sale_amount >= QUOTA;
```

- later in the program we can check the state of the flag:

```
if (sales_quota_met)
{ ...
```

- this is more readable than not using the flag:

```
if (sale_amount >= QUOTA)
{ ...
```

Scope

Scope

A variable's scope is the region of the program in which the variable exists and in which its name can be legally used.

- a variable's scope extends from declaration to the end of the closest containing block
- look at program_4_29_modified.html, which is a modification of Gaddis' Program 4-29 on page 214

Scope

Scope

A variable's scope is the region of the program in which the variable exists and in which its name can be legally used.

- a variable's scope extends from declaration to the end of the closest containing block
- look at program_4_29_modified.html, which is a modification of Gaddis' Program 4-29 on page 214

- some scopes:
  1. MIN_INCOME's scope is all of lines 8 – 36
  2. income's scope is lines 12 – 36
  3. years's scope is lines 18 – 30
  4. years_remaining's scope is lines 27 – 29

Variables With the Same Name

- on page 215 Gaddis shows a program that has two variables with the same name in nested scopes
- as he points out, this is legal but a bad idea
- to emphasize the point, never do this!

```
int number;
...
if (...)
{
  int number; // the outer number is no longer visible
  ...
}
// now outer number is again visible
```

Variables With the Same Name

- however, it is fine to have two variables with the same name in parallel scopes
- because only one side of the if statement can run, only one variable named number ever exists
- there is no ambiguity

```
if (...)
{
  int number;
  ...
}
else
{
  int number;
  ...
}
```

Not Included

- note: we will not cover Gaddis Section 4.14 The `switch` Statement at this time