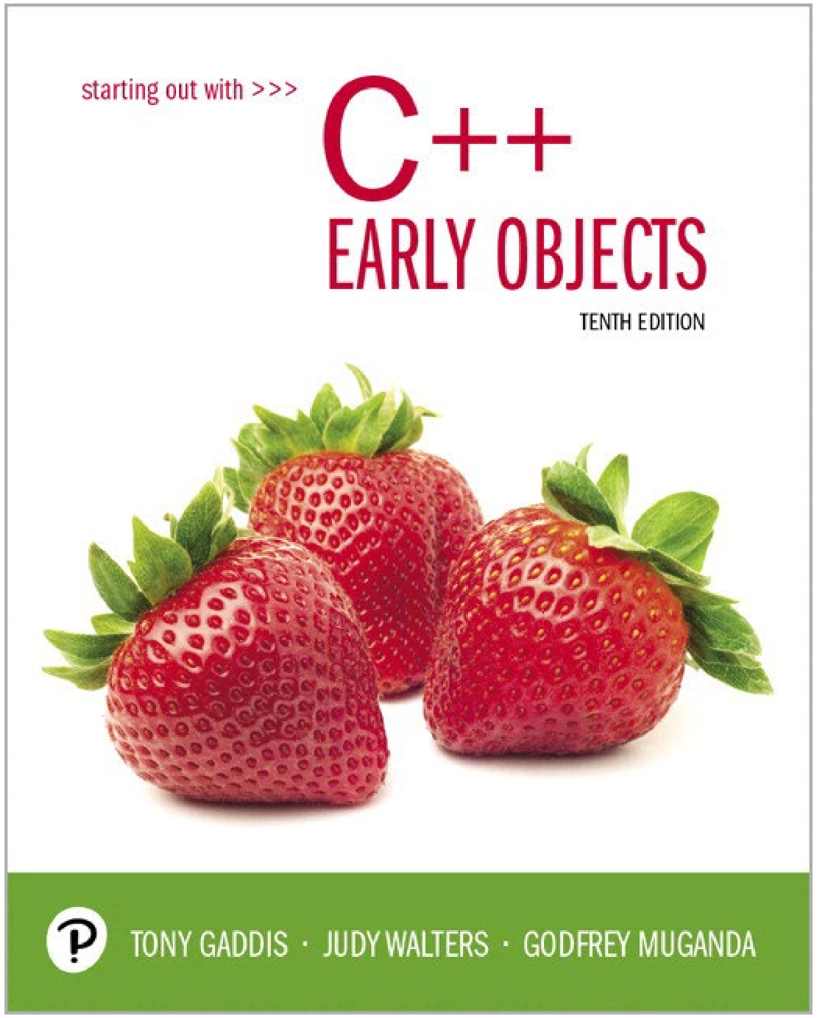
Starting Out with C++ Early Objects

Tenth Edition



Chapter 4

Making Decisions

Topics 1 of 2

Topics 2 of 2

4.1 Relational Operators

Are used to compare numeric and `char` values to determine relative order

Operator	Meaning
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

Relational Expressions 1 of 2

- Relational expressions are Boolean (*i.e.*, evaluate to **`true`** or **`false`**)
- Examples:

  `12 > 5` is **`true`**

  `7 <= 5` is **`false`**

  if **`x`** is 10, then

  `x == 10` is **`true`**,

  `x <= 8` is **`false`**,

  `x != 8` is **`true`**, and

  `x == 8` is **`false`**

Relational Expressions 2 of 2

- The value can be assigned to a variable

```
bool result = (x <= y);
```

- Assigns `0` for **`false`**, `1` for **`true`**
- Do not confuse `=` (assignment) and `==` (equal to)

Hierarchy of Relational Operators

Use this when evaluating an expression that contains multiple relational operators

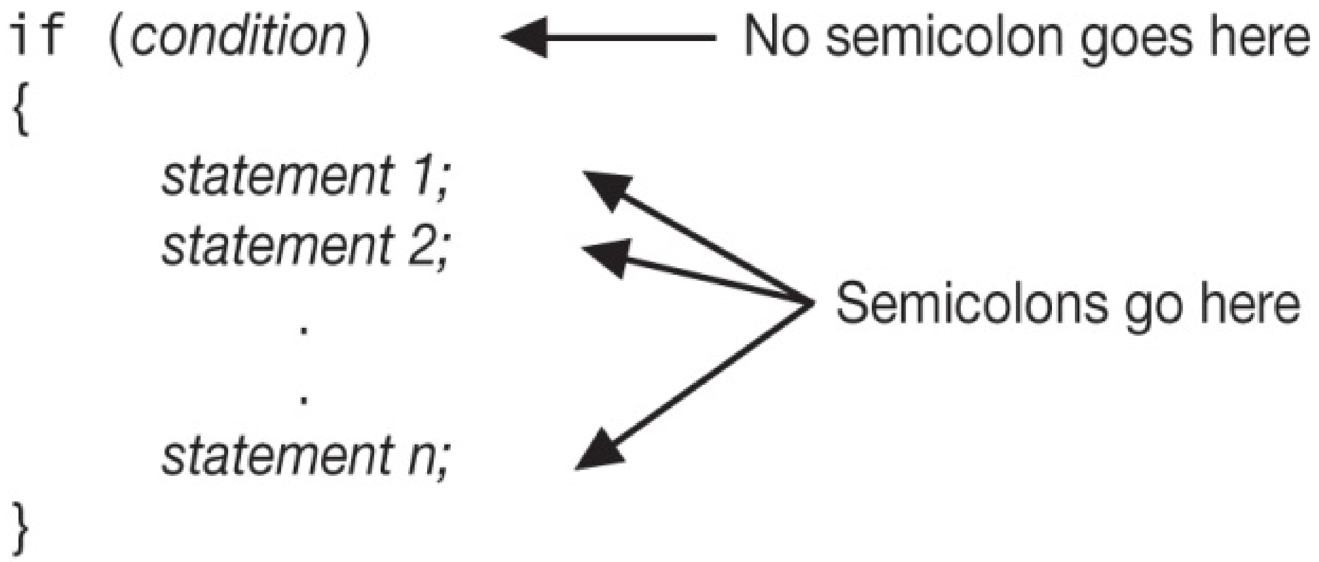
Operator	Precedence
`>` `>=` `<` `<=`	Highest
`==` `!=`	Lowest

4.2 The `if` Statement

- Supports the use of a decision structure, giving a program more than one path of execution
- Allows statements to be conditionally executed or skipped over
- It models the way we evaluate real-life situations

  "If it is cold outside,
  wear a coat and wear a hat."

Format of the `if` Statement

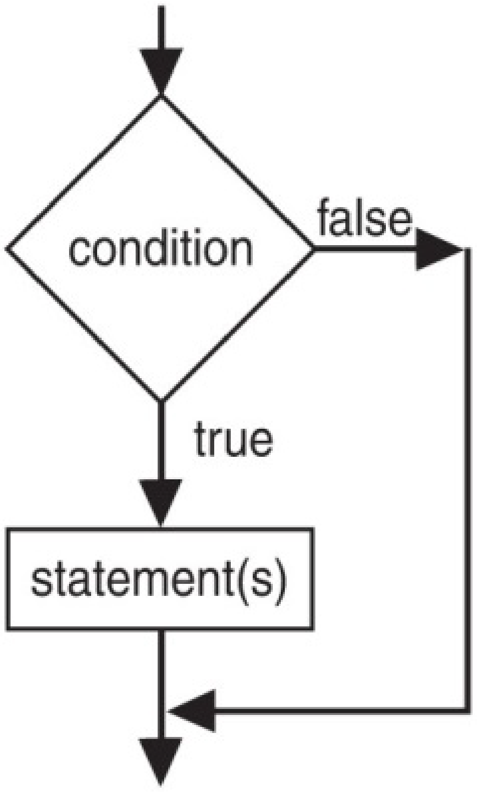


The block of statements inside the braces is called the <u>body</u> of the `if` statement. If there is only 1 statement in the body, the `{ }` may be omitted.

How the `if` Statement Works

- If `(`***`condition`***`)` is **`true`**, then the ***`statement(s)`*** in the body are executed.
- If `(`***`condition`***`)` is **`false`**, then the ***`statement(s)`*** are skipped.

`if` Statement Flow of Control



Example `if` Statements

```
if (score >= 60)
   cout << "You passed." << endl;

if (score >= 90)
{
   grade = 'A';
   cout << "Wonderful job!" << endl;
}
```

`if` Statement Notes

- `if` is a keyword. It must be lowercase
- `(`*`condition`*`)` must be in ( )
- Do not place `;` after `(`*`condition`*`)`
- Don't forget the { } around a multi-statement body
- Don’t confuse = (assignment) with == (comparison)

`if` Statement Style Recommendations

- Place each ***`statement;`*** on a separate line after (***`condition`***)
- Indent each statement in the body
- When using { and } around the body, put { and } on lines by themselves

What is `true` and what is `false`?

- An expression whose value is 0 is considered **`false`**.
- An expression whose value is non-zero is considered **`true`**.
- An expression need not be a comparison – it can be a single variable or a mathematical expression.

Flag

- A variable that signals a condition
- Usually implemented as a **`bool`**
- Meaning:
  - **`true`**: the condition exists
  - **`false`**: the condition does not exist
- The flag value can be both set and tested with **`if`** statements

Flag Example

Example:

```
bool validMonths = true;
            …
if (months < 0)
    validMonths = false;
            …
if (validMonths)
    monthlyPayment = total / months;
```

Integer Flags

- Integer variables can be used as flags
- Remember that 0 means false, any other value means true

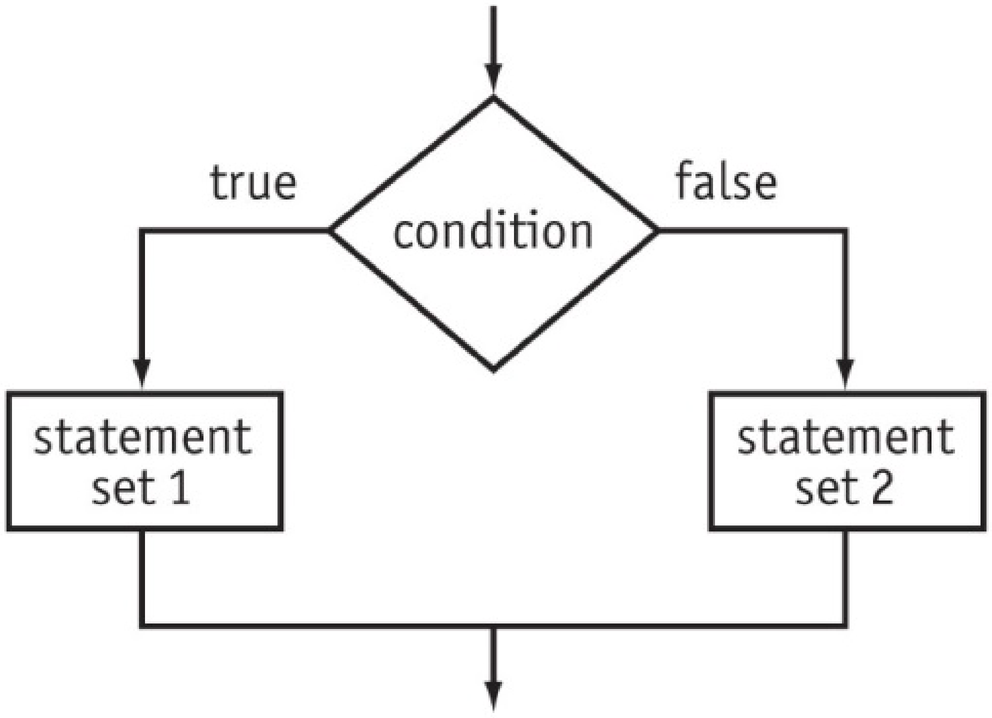
```
int allDone = 0;  // set to false
        …
 if (count > MAX_STUDENTS)
    allDone = 1;  // set to true
        …
 if (allDone)
    cout << "Task finished";
```

4.3 The `if/else` Statement

- Allows a choice between statements depending on whether `(`*`condition`*`)` is **`true`** or **`false`**
- Format:

```
if (condition)
 {
     statement set 1;
 }
 else
 {
     statement set 2;
 }
```

`if/else` Flow of Control



How the `if/else` Works

- If `(`***`condition`***`)` is **`true`**, ***`statement set 1`*** is executed and ***`statement set 2`*** is skipped.
- If `(`***`condition`***`)` is **`false`**, ***`statement set 1`*** is skipped and ***`statement set 2`*** is executed.

Example `if/else` Statements

```
if (score >= 60)
  cout << "You passed.\n";
 else
    cout << "You did not pass.\n";

 if (intRate > 0)
 {   interest = loanAmt * intRate;
     cout << interest;
 }
 else
    cout << "You owe no interest.\n";
```

`if` vs. `if/else`

If there are two conditions and both of them can be true or both can be false, then use two **`if`** statements:

```
if (num > 0)
    cout << num << " is positive\n";

if (num %2 == 0)
    cout << num << " is even\n";
```

If the two conditions cannot both be true, then a single **`if/else`** statement can work:

```
if (num %2 == 0)
    cout << num << " Is even\n";
 else
    cout << num << " Is odd\n";
```

Comparisons with Floating-Point Numbers

- It is difficult to test for equality when working with floating point numbers.
- It is better to use
  - greater-than or less-than tests, or
  - test to see if value is very close to a given value

4.4 The `if/else if` Statement

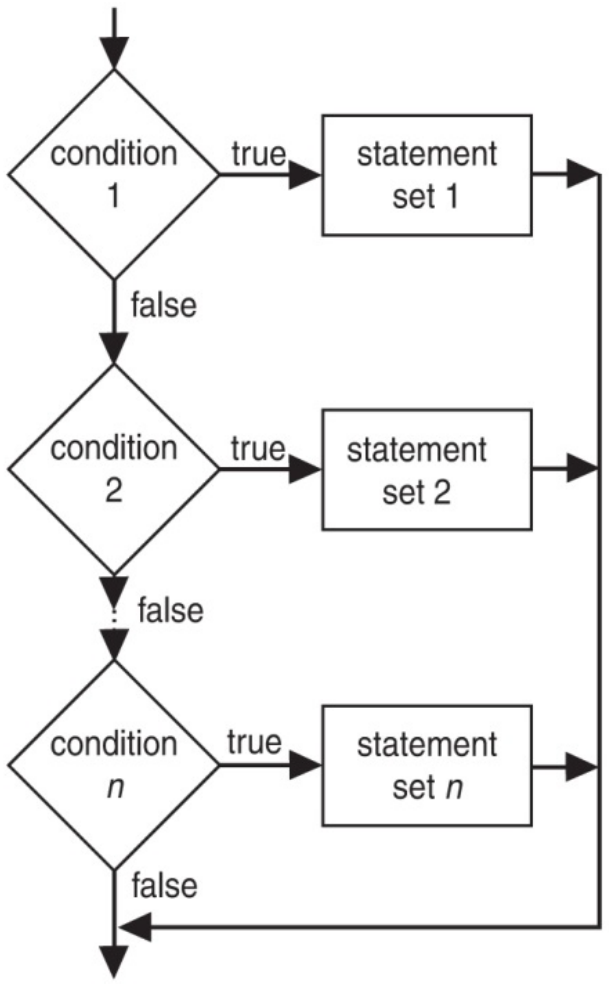
- Chain of `if` statements that test in order until one is found to be true
- This also models thought processes

  "If it is raining, take an umbrella,
  else, if it is windy, take a hat,
  else, if it is sunny, take sunglasses."

`if/else if` Format

```
if (condition 1)
{
        statement set 1;
}
else if (condition 2)
{
        statement set 2;
}
                .
                .

else if (condition n)
{
        statement set n;
}
```



Using a Trailing `else`

- Is used with a set of `if/else if` statements
- It provides a default statement or action that is performed when none of the conditions is true
- It can be used to catch invalid values or handle other exceptional situations

Example `if/else if` with Trailing `else`

```
if (age >= 21)
   cout << "Adult";
else if (age >= 13)
   cout << "Teen";
else if (age >= 2)
   cout << "Child";
else
   cout << "Baby";
```

4.5 Menu-Driven Program

- Menu: list of choices presented to the user on the computer screen
- Menu-driven program: program execution controlled by user selecting from a list of actions
- A menu-driven program can be written using `if/else if` statements

Menu-driven Program Organization

- Display a list of numbered or lettered choices for actions.
- Input user's selection of number or letter
- Test user selection in `(condition)`
  - if a match, then execute code to carry out desired action
  - if not, then test with next `(condition)`

4.6 Nested `if` Statements

- An **`if`** statement that is part of the **`if`** or **`else`** part of another **`if`** statement
- This can be used to evaluate > 1 data item or to test > 1 condition

```
if (score < 100)
{
    if (score > 90)
        grade = 'A';
}
```

Notes on Coding Nested `ifs`

- An **`else`** matches the nearest previous **`if`** that does not have an **`else`**

```
if (score < 100)
    if (score > 90)
        grade = 'A';
    else ...   // goes with second if,
               // not first one
```

- Proper indentation aids comprehension

4.7 Logical Operators

Are used to create relational expressions from other relational expressions

Operator	Meaning	Explanation
&&	**AND**	New relational expression is true if both expressions are true
||	**OR**	New relational expression is true if either expression is true
!	**NOT**	Reverses the truth value of an expression; true expression becomes false, false expression becomes true

Logical Operator Examples

```
int x = 12, y = 5, z = -4;
```

Expression	Value
`(x > y) && (y > z)`	`true or 1`
`(x > y) && (z > y)`	`false or 0`
`(x <= z) || (y == z)`	`false`
`(x <= z) || (y != z)`	`true`
`!(x >= z)`	`false`

Logical Operator and `bool` Variables

- Logical operators can be used with **`bool`** variables as well as expressions that evaluate to **`true`** or **`false`**.
- Ex:

```
bool done = false;
if ((!done) && (count < 6))
{
. . .
}
```

Short-Circuit Evaluation

- If an expression using the `&&` operator is being evaluated and the subexpression on the left side is `false`, then there is no reason to evaluate the subexpression on the right side. It is skipped.
- If an expression using the `||` operator is being evaluated and the subexpression on the left side is `true`, then there is no reason to evaluate the right side. It is skipped.

Logical Precedence

Highest	`!`
	`&&`
Lowest	`||`

Example:

```
(2 < 3) || (5 > 6) && (7 > 8)
```

is true because && is evaluated before | |

Checking Numeric Ranges with Logical Operators

- Used to test if a value is within a range

```
if ((grade >= 0) && (grade <= 100))
    cout << "Valid grade";
```

- Can also test if a value lies outside a range

```
if ((grade <= 0) || (grade >= 100))
    cout << "Invalid grade";
```

- You cannot use mathematical notation

```
if (0 <= grade <= 100) //Doesn't
                       //work!
```

4.8 Validating User Input

- Input validation: inspecting input data to determine if it is acceptable
- You want to avoid accepting bad input
- You can perform various tests
  - Range
  - Reasonableness
  - Valid menu choice
  - Zero as a divisor

4.9 More About Blocks and Scope

- The Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Variables are usually defined at the beginning of a function
- They may instead be defined close to the place where they are first used

More About Blocks and Scope

- Variables defined inside `{ }` have local or block scope
- When in a block that is nested inside another block, you can define variables with the same name as in the outer block.
  - When the program is executing in the inner block, the outer definition is not available.
  - This generally not a good idea. The program may be hard to read or understand.

4.10 More About Characters and Strings

- You can use relational operators with characters and string objects

  `if (menuChoice == 'A')`, or `if (name1 >= name2)`

- Comparing characters is really comparing the ASCII values of characters
- Comparing string objects is comparing the ASCII values of the characters in the strings. Comparison is character-by-character, starting with the first character of each string.
- You cannot compare C-style strings by using relational operators

Testing Characters 1 of 2

These functions require the **`cctype`** header file

FUNCTION	MEANING
`isalpha`	**`true`** if arg. is a letter, **`false`** otherwise
`isalnum`	**`true`** if arg. is a letter or digit, **`false`** otherwise
`isdigit`	**`true`** if arg. is a digit 0-9, **`false`** otherwise
`islower`	**`true`** if arg. is lowercase letter, **`false`** otherwise

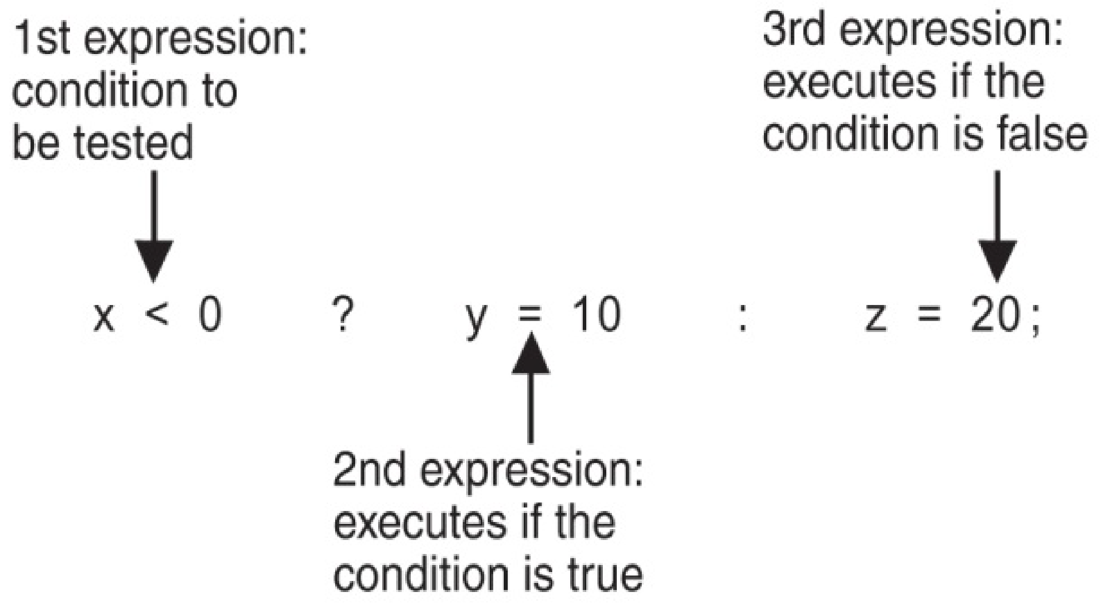
Testing Characters 2 of 2

These functions require the **`cctype`** header file

FUNCTION	MEANING
`isprint`	**`true`** if arg. is a printable character, **`false`** otherwise
`ispunct`	**`true`** if arg. is a punctuation character, **`false`** otherwise
`isupper`	**`true`** if arg. is an uppercase letter, **`false`** otherwise
`isspace`	**`true`** if arg. is a whitespace character, **`false`** otherwise

4.11 The Conditional Operator

- This can be used to create short **`if/else`** statements
- Format: **`expr ? expr : expr;`**


1st expression: condition to be tested
3rd expression: executes if the condition is false
x < 0 ? y = 10 : z = 20;
2nd expression: executes if the condition is true

The Value of a Conditional Expression

- A conditional expression is an expression that uses a conditional operator
- The value of a conditional expression is determined by whichever of the subexpressions is executed

```
int num = 13;
string result= (num%2 ==0) ? "even" : "odd";
cout << num << " is " << result;
```

Pearson

4.12 The `switch` Statement

- It uses the value of an integer expression to determine the statements to execute

- It may sometimes be used instead of **`if/else if`** statements

switch Statement Format

```
switch (IntExpression)
{
    case exp1: statement set 1;
    case exp2: statement set 2;
    ...
    case expn: statement set n;
    default:   statement set n+1;
}
```

`switch` Statement Requirements

1) ***`IntExpression`*** must be an integer variable or a **`char`**,or an expression that evaluates to an integer value

2) ***`exp1`*** through ***`expn`*** must be constant integer type expressions and must be unique in the **`switch`** statement

3) **`default`** is optional but recommended

How the `switch` Statement Works

1) ***`IntExpression`*** is evaluated

2) The value of ***`IntExpression`*** is compared against ***`exp1`*** through ***`expn`***.

3) If ***`IntExpression`*** matches value ***`expi`***, the program branches to the statement(s) following ***`expi`*** and continues to the end of the **`switch`**

4) If no matching value is found, the program branches to the statement after **`default:`**

The `break` Statement

- Is used to stop execution in the current block
- It is also used to exit a `switch` statement
- It is used to execute the statements for a single `case` expression without executing the statements of the cases following it

Example switch Statement

```
switch (plusOrMinus)
{
  case '+': cout << "plus";
            break;
  case '-': cout << "minus";
            break;
  default : cout << "invalid value";
}
```

Using **`switch`** with a Menu

A **`switch`** statement is a natural choice for a menu-driven program

- display menu
- get user input
- use user input as **`IntExpression`** in **`switch`** statement
- use menu choices as **`exp`** values to test against in the **`case`** statements

4.13 Enumerated Data Types

- Is a data type created by the programmer
- Contains a set of named integer constants
- Format:

enum ***name*** **{*val1*, *val2*, … *valn*};**

- Examples:

```
enum Fruit {apple, grape, orange};
enum Days {Mon, Tue, Wed, Thur, Fri};
```

Enumerated Data Type Variables

- To define variables, use the enumerated data type name

```
Fruit snack;
Days workDay, vacationDay;
```

- A variable may contain any valid value for the data type

```
snack = orange;         // no quotes
if (workDay == Wed)  // none here either
```

Comparison of Enumerated Data Type Values

- Enumerated data type values are associated with integers, starting at 0

```
enum Fruit {apple, grape, orange};
                0      1       2
```

- You can override the default association

```
  enum Fruit {apple = 2, grape = 4,
              orange = 5};
```

Enumerated Data Type Values

- Enumerated data type values can be compared using their integer values

```
if (snack == 1)
```

- Enumerated data type values cannot be assigned using their integer values

```
snack = 2;  // won't work
```

Enumerated Data Type Notes

- Enumerated data types improve the readability of a program
- Enumerated variables can not be used with input statements, such as **`cin`**
- An enumerated variable will not display the name associated with the value of an enumerated data type if used with **`cout`**

Copyright