Object-Oriented Programming

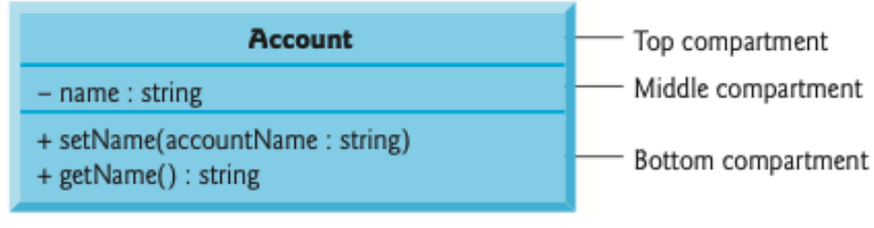
Lecture 4: Classes, Objects, Algorithm Development and Control Statements (book chapter 3 + 4)

Accounting Class and Objects
Algorithm Development & Control Statement

- Account object
- Data and function member
- Constructor
- Data validation
- Algorithms
- Control Structure
- Counter-Controlled Iteration
- Sentinel-Controlled Iteration
- Nested Control Statements

Accounting Program

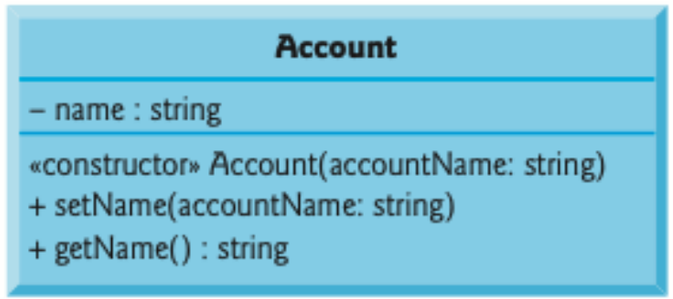
```
#include <string>
class Account {

public:
    void setName(std::string accountName) {
        name = accountName; // Store the account name
    }

    std::string getName() const {
        return name; // Return name's value to this function's caller
    }
private:
    std::string name; // Data member containing account holder's name
};
```



Accounting Program

```
#include <iostream>
#include <string>
#include "Account.h"
int main() {
    Account myAccount; // Instantiating Account object myAccount

    std::cout << "Initial account name is: " << myAccount.getName() << std::endl;

    std::cout << "\nPlease enter the account name: ";
    std::string theName;
    getline(std::cin, theName);
    myAccount.setName(theName);
    std::cout << "Name in object myAccount is: " << myAccount.getName() << std::endl;

    return 0;
}
```

Constructor



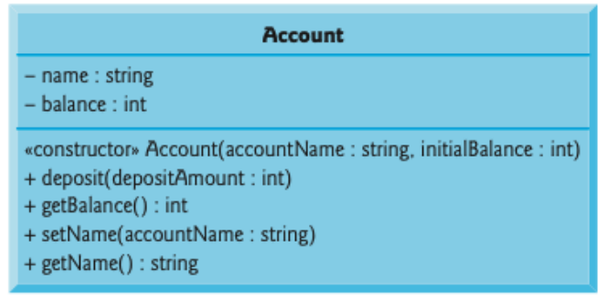

```
#include <iostream>
#include <string>

class Account {
public:
    explicit Account(std::string accountName) : name{accountName} {
    }

    void setName(std::string accountName) {
        name = accountName; // Store the account name
    }

    std::string getName() const {
        return name; // Return name's value to this function's caller
    }
private:
    std::string name; // Data member containing account holder's name
};
```

Constructor

```
int main() {
    // create two Account objects
    Account account1{"Jane Green"};
    Account account2{"John Blue"};

    // display initial value of name for each Account
    std::cout << "account1 name is: " << account1.getName() << std::endl;
    std::cout << "account2 name is: " << account2.getName() << std::endl;
}
```

Data Validation

```
#include <iostream>
#include <string>

class Account {
public:
    Account(std::string accountName, int initialBalance)
    : name{accountName} {
        if (initialBalance > 0) {
            balance = initialBalance;
        }
    }

    void deposit(int depositAmount) {
        if (depositAmount > 0) {
            balance += depositAmount;
        }
    }

    int getBalance() const {
        return balance;
    }

    void setName(std::string accountName) {
        name = accountName;
    }

    std::string getName() const {
        return name;
    }
private:
    std::string name; // account name data member
    int balance{0}; // data member with default initial value
};
```



Data Validation

```
int main() {
    Account account1{"Jane Green", 50};
    Account account2{"John Blue", 0};
    std::cout << "account1: " << account1.getName() << " balance is $" << account1.getBalance();
    std::cout << "\naccount2: " << account2.getName() << " balance is $" << account2.getBalance();

    std::cout << "\n\nEnter deposit amount for account1: "; // prompt
    int depositAmount;
    std::cin >> depositAmount; // obtain user input
    std::cout << "adding " << depositAmount << " to account1 balance";
    account1.deposit(depositAmount); // add to account1's balance

    std::cout << "\n\naccount1: " << account1.getName() << " balance is $" << account1.getBalance();
    std::cout << "\naccount2: " << account2.getName() << " balance is $" << account2.getBalance();

    std::cout << "\n\nEnter deposit amount for account2: "; // prompt
    std::cin >> depositAmount; // obtain user input
    std::cout << "adding " << depositAmount << " to account2 balance";
    account2.deposit(depositAmount); // add to account2's balance

    std::cout << "\n\naccount1: " << account1.getName() << " balance is $" << account1.getBalance();
    std::cout << "\naccount2: " << account2.getName() << " balance is $" << account2.getBalance() <<
std::endl;

    return 0; // indicate successful termination
}
```

```
account1: Jane Green balance is $50
account2: John Blue balance is $0

Enter deposit amount for account1: 25
adding 25 to account1 balance

account1: Jane Green balance is $75
account2: John Blue balance is $0

Enter deposit amount for account2: 123
adding 123 to account2 balance

account1: Jane Green balance is $75
account2: John Blue balance is $123
```

Algorithm Development & Control Statement

Algorithm

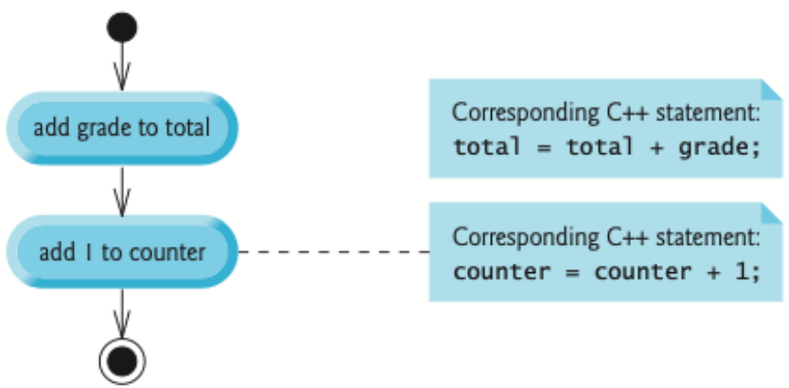
Algorithm: a procedure for solving a problem in terms of

1. the actions to execute and
2. the order in which these actions execute

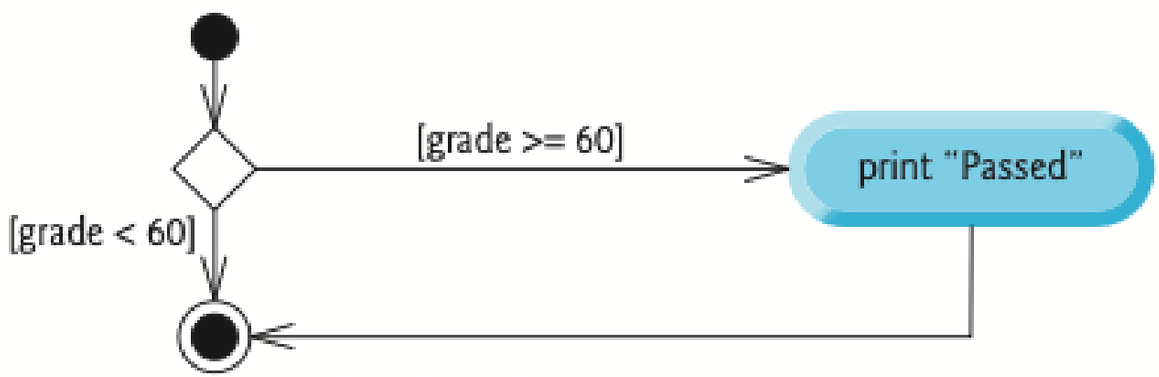
Pseudocode: is an informal language that helps you develop algorithms without having to worry about the strict details of C++ language syntax.

```
1  Prompt the user to enter the first integer
2  Input the first integer
3
4  Prompt the user to enter the second integer
5  Input the second integer
6
7  Add first integer and second integer, store result
8  Display result
```

Control Structures

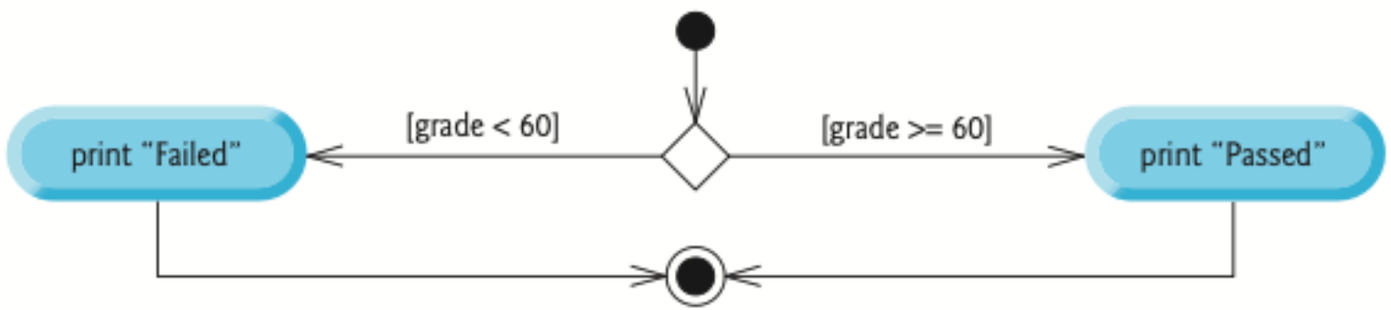


Sequence-structure activity diagram.



if single-selection statement

```
if (studentGrade >= 60) {
   cout << "Passed";
}
```




```
if (grade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed";
}
```

if...else double-selection statement

Control Structures

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

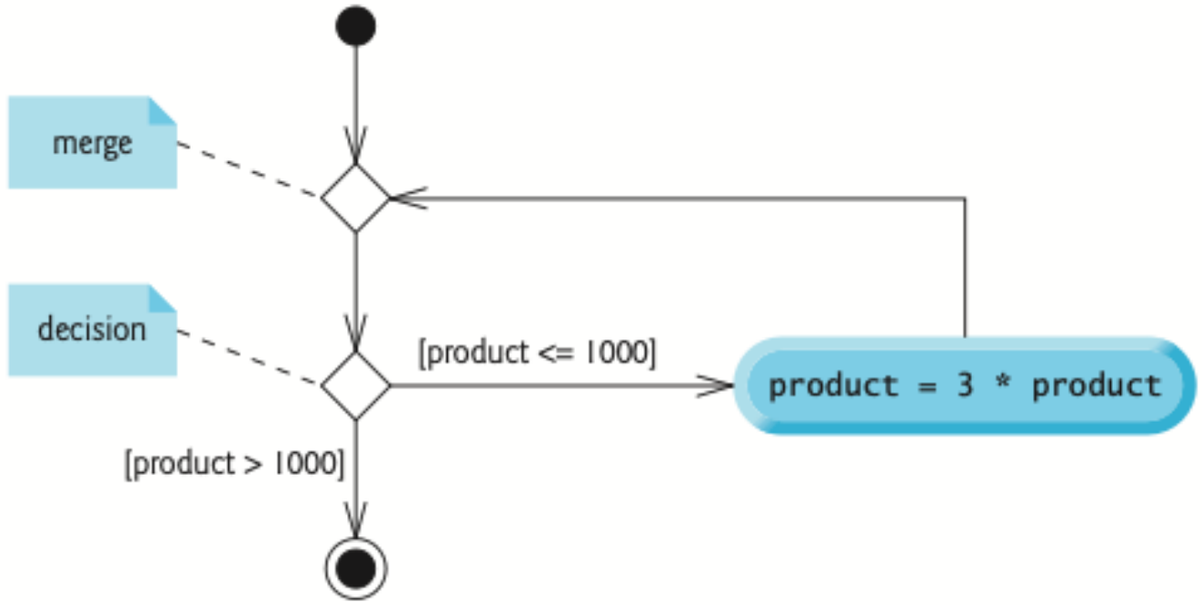
Nested if...else statements.

```
if (studentGrade >= 90) {
   cout << "A";
}
else {
   if (studentGrade >= 80) {
      cout << "B";
   }
   else {
      if (studentGrade >= 70) {
         cout << "C";
      }
      else {
         if (studentGrade >= 60) {
            cout << "D";
         }
         else {
            cout << "F";
         }
      }
   }
}
```

```
cout << (studentGrade >= 60 ? "Passed" : "Failed");
```

Conditional Operator (?:)

Control Structures




```
int product{3};

while (product <= 100) {
   product = 3 * product;
}
```

while iteration statement

Counter-Controlled

```
#include <iostream>

int main() {
   // Initialization phase
   int total = 0; // Initialize sum of grades entered by the user
   unsigned int gradeCounter = 1; // Initialize grade # to be entered next

   // Processing phase uses counter-controlled iteration
   while (gradeCounter <= 10) { // Loop 10 times
      std::cout << "Enter grade: "; // Prompt
      int grade;
      std::cin >> grade; // Input next grade
      total += grade; // Add grade to total
      ++gradeCounter; // Increment counter by 1
   }
   // Termination phase
   int average = total / 10; // Integer division yields integer result

   // Display total and average of grades
   std::cout << "\nTotal of all 10 grades is " << total;
   std::cout << "\nClass average is " << average << std::endl;
}
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

```
std::cout << "INT MAX: " << INT_MAX << " , INT MIN: " << INT_MIN << std::endl;
```

```
INT MAX: 2147483647 , INT MIN: -2147483648
```

Input Validation

1. **Type Checking**: Ensuring the input matches the expected data type. For instance, if a program expects an integer, it should verify that the user didn't enter a non-numeric value.
2. **Range Checking**: Verifying that the input falls within a specified range. For example, if a program requires an age input, it should check that the age is within a reasonable range (e.g., 0 to 120).
3. **Format Checking**: In cases where the input must follow a certain format (like dates or phone numbers), the program should confirm that the user's input adheres to this format.
4. **Consistency Checks**: Ensuring that the input is consistent with other data or constraints. For example, an end date should not be earlier than a start date.
5. **Sanitization**: In cases where input might cause security issues, such as with SQL queries or HTML data, it's important to sanitize the input to prevent attacks like SQL injection or cross-site scripting.
6. **Feedback to Users**: When input validation fails, it is crucial to provide clear and specific feedback to users so they can correct their input.

Sanitization

- The Problem:
  - Untrusted Data
  - Injection Attacks:
    - SQL Injection
    - Cross-Site Scripting (XSS)
- Sanitization as a Solution: The process of cleaning up input data to remove or neutralize potentially harmful elements:
  - Removing Dangerous Characters
  - Encoding Data
  - Using Safe APIs
  - Whitelisting

Regular Expressions (Regex)

- Mini-language for pattern matching within text.
- Use for:

  - Validation: Check if data matches a specific format (like email addresses, phone numbers, or URLs).

  - Searching: Find occurrences of patterns within large amounts of text.

  - Replacing: Modify text by searching for patterns and replacing them with other text.

  - Extracting: Pull out specific pieces of information from text.
- **std::regex** object containing the pattern.
- **regex_match** compares the entire input string to the pattern.

  - returns true only if the entire input string matches the pattern from beginning to end.

Syntax

- Literal Characters: Most characters match themselves.
- Metacharacters

. (dot): Matches any single character (except newline).
^: Matches the beginning of a string or line.
$: Matches the end of a string or line.
*: Matches the preceding element zero or more times.
+: Matches the preceding element one or more times.
?: Matches the preceding element zero or one time.
{n}: Matches the preceding element exactly n times.
{n,}: Matches the preceding element at least n times.
{n,m}: Matches the preceding element at least n times and at most m times.
[]: Defines a character set. Matches any single character within the brackets.
[abc]: Matches "a", "b", or "c".
[a-z]: Matches any lowercase letter.
[A-Z]: Matches any uppercase letter.
[0-9]: Matches any digit.
[^]: Defines a negated character set. Matches any single character not within the brackets.
[^abc]: Matches any character except "a", "b", or "c".
|: Acts as an "or" operator. Matches either the expression before or after it. - cat|dog: Matches "cat" or "dog".
(...): Creates a capturing group. Used for extracting submatches or applying quantifiers to a group of characters.
\: Escapes a metacharacter, allowing you to match it literally.
\.: Matches a literal dot.

Syntax

- Character Classes

  \d: Matches any digit (equivalent to [0-9]).
  \D: Matches any non-digit (equivalent to [^0-9]).
  \w: Matches any word character (alphanumeric and underscore).
  \W: Matches any non-word character.
  \s: Matches any whitespace character (space, tab, newline).
  \S: Matches any non-whitespace character.

Example

^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$

^: Matches the beginning of the string.
[a-zA-Z0-9._%+-]+: Matches one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens (for the username).
@: Matches the "@" symbol.
[a-zA-Z0-9.-]+: Matches one or more alphanumeric characters, dots, or hyphens (for the domain).
\.: Matches a literal dot.
[a-zA-Z]{2,}: Matches two or more alphabetic characters (for the top-level domain).
$: Matches the end of the string.

```
#include <iostream>
#include <string>
#include <regex>
#include <sstream>

bool isInteger(const std::string& str) {
    std::istringstream iss(str);
    int num;
    iss >> num;
    return iss.eof() && !iss.fail();
}
bool isValidEmail(std::string email)
bool isValidEmail(const std::string& email) {
    const std::regex emailPattern(R"(^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$)");
    return std::regex_match(email, emailPattern);
}

int main() {
    std::string input;
    int integerValue;
    float floatValue;
    std::string singleWord, emailAddress;

    // Integer input
    std::cout << "Enter an integer: ";
    while (true) {
        std::getline(std::cin, input);
        if (isInteger(input)) {
            integerValue = std::stoi(input);
            break;
        }
        std::cout << "Invalid input. Please enter an integer: ";
    }
```

```
// Floating-point input
std::cout << "Enter a floating-point number: ";
while (!(std::cin >> floatValue)) {
    std::cout << "Invalid input. Please enter a floating-point number: ";
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

// Clear the input buffer before reading single-word string
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

// Single-word string input
std::cout << "Enter a single word: ";
std::cin >> singleWord;

    // Email address input
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::cout << "Enter an email address: ";
    std::getline(std::cin, emailAddress);
    while (!isValidEmail(emailAddress)) {
        std::cout << "Invalid email. Please enter a valid email address: ";
        std::getline(std::cin, emailAddress);
    }

    std::cout << "\nInteger Value: " << integerValue
              << "\nFloating-Point Value: " << floatValue
              << "\nSingle Word: " << singleWord
              << "\nEmail Address: " << emailAddress << std::endl;

    return 0;
}
```

std::istringstream

A class provided by the C++ Standard Library in the <sstream> header file.

1. String Parsing: std::istringstream allows you to treat a string as a stream of input data. This is useful when you have a string containing several pieces of data separated by spaces or other delimiters, and you want to extract these pieces individually.

2. Type-Safe Extraction: It provides type-safe data extraction from the string. For example, you can use it to extract integers, floating-point numbers, characters, and strings from a single string, respecting their types.

3. Stream Operators: It overloads the >> (extraction) operator, enabling you to extract data from the string in a formatted and type-safe manner.

4. Error Handling: Like other C++ streams, std::istringstream provides mechanisms to check the state of the stream (such as whether an extraction operation failed) using functions like fail(), eof(), etc.

5. Conversion of String to Various Data Types: It is often used to convert a string to various other data types in a safe and controlled manner, as it respects the data types and formats during extraction.

```
#include <iostream>
#include <sstream>
#include <string>

int main() {
    std::string data = "123 45.67 Hello";
    std::istringstream iss(data);

    int intValue;
    double doubleValue;
    std::string stringValue;

    iss >> intValue;       // Extracts an integer (123)
    iss >> doubleValue;    // Extracts a double (45.67)
    iss >> stringValue;    // Extracts a string ("Hello")

    std::cout << "Integer: " << intValue << "\n"
              << "Double: " << doubleValue << "\n"
              << "String: " << stringValue << std::endl;
    return 0;
}
```

Sentinel-Controlled Iteration

- Also known as a "Sentinel Loop”
- A programming pattern where the loop continues to execute until it encounters a specific value within the input, which is referred to as the "sentinel" value.
- This sentinel value acts as a signal to **terminate** the loop and is typically chosen such that it wouldn't normally occur as a valid input in the loop's processing logic.

Sentinel-Controlled Iteration

- Choose a Sentinel: Select a value that won't naturally occur. For example, if you're processing positive numbers, you might use -1 as the sentinel.
- Loop and Check: Use a loop (typically a while loop) to continuously read input. Inside the loop, immediately check if the input is equal to the sentinel value.
- Process or Terminate: If the input is not the sentinel, process it as needed. If it is the sentinel, terminate the loop.

```
#include <iostream>

int main() {
    int input;
    const int SENTINEL = -1; // Sentinel value chosen as -1

    std::cout << "Enter numbers (enter -1 to stop): ";

    while (true) {
        std::cin >> input;
        if (input == SENTINEL) {
            break; // Exit the loop if the sentinel value is encountered
        }
        // Process the input
        std::cout << "You entered: " << input << std::endl;
    }

    std::cout << "End of input." << std::endl;
    return 0;
}
```

Floating-Point Formatting

```
#include <iostream>
#include <iomanip>

int main() {
    double number = 123456.789;

    // Default output
    std::cout << "Default format: " << number << "\n";
    // Fixed-point notation with precision
    std::cout << std::fixed;
    std::cout << "Fixed-point with 2 decimal places: " << std::setprecision(2) << number << "\n";
    std::cout << "Fixed-point with 5 decimal places: " << std::setprecision(5) << number << "\n";
    // Scientific notation
    std::cout << std::scientific;
    std::cout << "Scientific notation with 2 decimal places: " << std::setprecision(2) << number << "\n";
    std::cout << "Scientific notation with 5 decimal places: " << std::setprecision(5) << number << "\n";
    // Setting width and filling with zeros
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Fixed-point with width 20, filled with zeros: " << std::setw(20) << std::setfill('0') <<
number << "\n";
    // Resetting format to default
    std::cout.unsetf(std::ios_base::floatfield);
    // Showpoint
    std::cout << "Showpoint: " << std::showpoint << number << "\n";
    // Show trailing zeroes with fixed-point format
    std::cout << std::fixed << "Fixed-point with trailing zeroes: " << number << "\n";
    return 0;
}
```

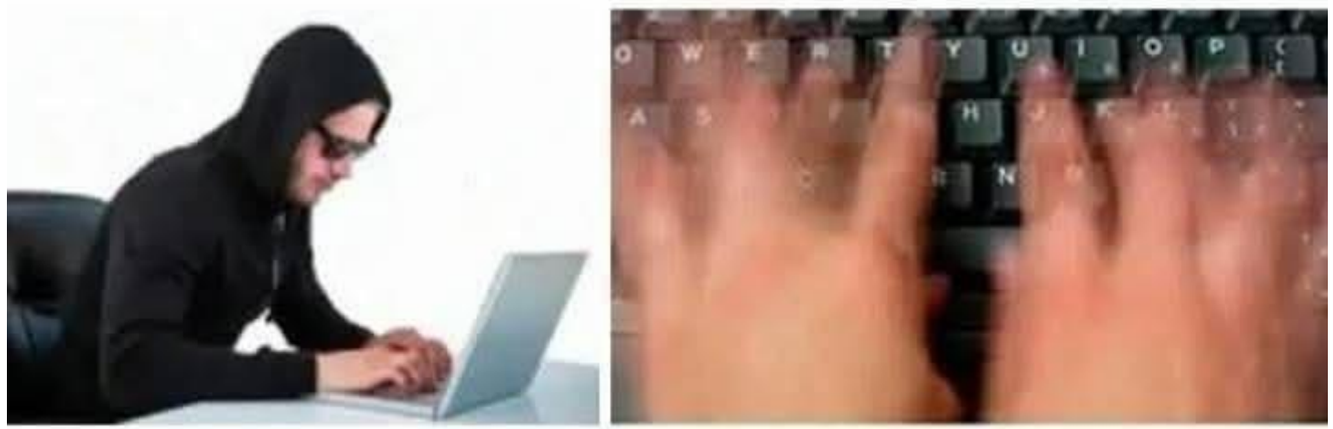
Nested Control Statements

```cpp
#include <iostream>
using namespace std;

int main() {
   // initializing variables in declarations
   unsigned int passes{0};
   unsigned int failures{0};
   unsigned int studentCounter{1};

   // process 10 students using counter-controlled loop
   while (studentCounter <= 10) {
      // prompt user for input and obtain value from user
      cout << "Enter result (1 = pass, 2 = fail): ";
      int result;
      cin >> result;

      // if...else is nested in the while statement
      if (result == 1) {
         passes = passes + 1;
      }
      else {
         failures = failures + 1;
      }

      // increment studentCounter so loop eventually terminates
      studentCounter = studentCounter + 1;
   }

   // termination phase; prepare and display results
   cout << "Passed: " << passes << "\nFailed: " << failures << endl;

   // determine whether more than 8 students passed
   if (passes > 8) {
      cout << "Bonus to instructor!" << endl;
   }
}
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

What people think programming is like



What programming is actually like

Compouhd Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Increment and Decrement Operators

Operator	Operator name	Sample expression	Explanation
++	prefix increment	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postfix increment	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	prefix decrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postfix decrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

```cpp
#include <iostream>
using namespace std;

int main() {
   // demonstrate postfix increment operator
   unsigned int c{5}; // initializes c with the value 5
   cout << "c before postincrement: " << c << endl; // prints 5
   cout << "    postincrementing c: " << c++ << endl; // prints 5
   cout << " c after postincrement: " << c << endl; // prints 6

   cout << endl; // skip a line

   // demonstrate prefix increment operator
   c = 5; // assigns 5 to c
   cout << " c before preincrement: " << c << endl; // prints 5
   cout << "     preincrementing c: " << ++c << endl; // prints 6
   cout << "  c after preincrement: " << c << endl; // prints 6
}
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

Accounting Class and Objects
Algorithm Development & Control Statement

- for Interation statement
- do . . . while interation
- switch statement
- Logical operator
- C++ program components
- Math library functions
- Function-prototype
- Standard C++ library headers
- Scope rules
- Function-call stack
- Inline function
- Reference and reference parameters
- Default arguments
- Unary scope resolution operator
- Function Overloading
- Recursion
- Recursion vs. iteration

Essentials of Counter-Controlled Iteration

```
#include <iostream>
using namespace std;

int main() {
   unsigned int counter{1}; // declare and initialize control variable

   while (counter <= 10) { // loop-continuation condition
      cout << counter << "   ";
      ++counter; // increment control variable
   }

   cout << endl;
}
```

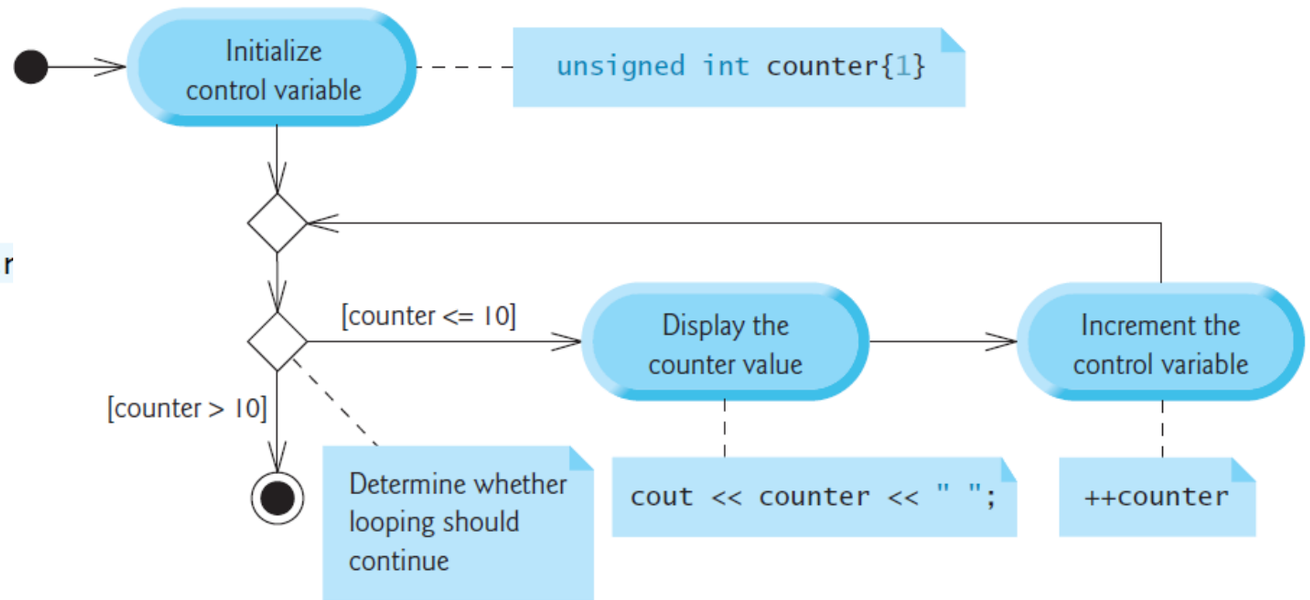
Control variable

Initial value

Increment / decrement

for Iteration Statement

```cpp
#include <iostream>
using namespace std;

int main() {
   // for statement header includes initialization,
   // loop-continuation condition and increment
   for (unsigned int counter{1}; counter <= 10; ++counter
      cout << counter << "   ";
   }

   cout << endl;
}
```




```cpp
counter = counter + 1
counter += 1
++counter
counter++
```

```cpp
for (unsigned int j = x; j <= 4 * x * y; j += y / x)
```

```cpp
for (unsigned int i{1}; i <= 100; i++)
```

```cpp
for (unsigned int i{100}; i >= 1; i--)
```

Application: Compound-Interest Calculations

A person invests $1,000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p\,(1 + r)^n$$

where

p is the original amount invested (i.e., the principal)

r is the annual interest rate (e.g., use 0.05 for 5%)

n is the number of years

a is the amount on deposit at the end of the nth year.

Application: Compound-Interest Calculations

```
#include <iostream>
#include <iomanip>
#include <cmath> // Added for pow function
using namespace std;

int main() {
   // set floating-point number format
   cout << fixed << setprecision(2);

   double principal{1000.00}; // initial amount before interest
   double rate{0.05}; // interest rate

   cout << "Initial principal: " << principal << endl;
   cout << " Interest rate: " << rate << endl;

   // display headers
   cout << "\nYear" << "Amount on deposit" << endl; // Corrected the double <<

   // calculate amount on deposit for each of ten years
   for (unsigned int year{1}; year <= 10; year++) {
      // calculate amount on deposit at the end of the specified year
      double amount = principal * pow(1.0 + rate, year);
      // display the year and the amount
      cout << setw(4) << year << setw(20) << amount << endl;
   }
}
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

```
class DollarAmount {
public:
   // initialize amount from an int64_t value
   explicit DollarAmount(int64_t value) : amount{value} { }

   // add right's amount to this object's amount
   void add(DollarAmount right) {
      // can access private data of other objects of the same class
      amount += right.amount;
   }

   // subtract right's amount from this object's amount
   void subtract(DollarAmount right) {
      // can access private data of other objects of the same class
      amount -= right.amount;
   }

   // uses integer arithmetic to calculate interest amount,
   // then calls add with the interest amount
   void addInterest(int rate, int divisor) {
      // create DollarAmount representing the interest
      DollarAmount interest{
         (amount * rate + divisor / 2) / divisor
      };

      add(interest); // add interest to this object's amount
   }

   // return a string representation of a DollarAmount object
   std::string toString() const {
      std::string dollars{std::to_string(amount / 100)};
      std::string cents{std::to_string(std::abs(amount % 100))};
      return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
   }
private:
   int64_t amount{0}; // dollar amount in pennies
};
```

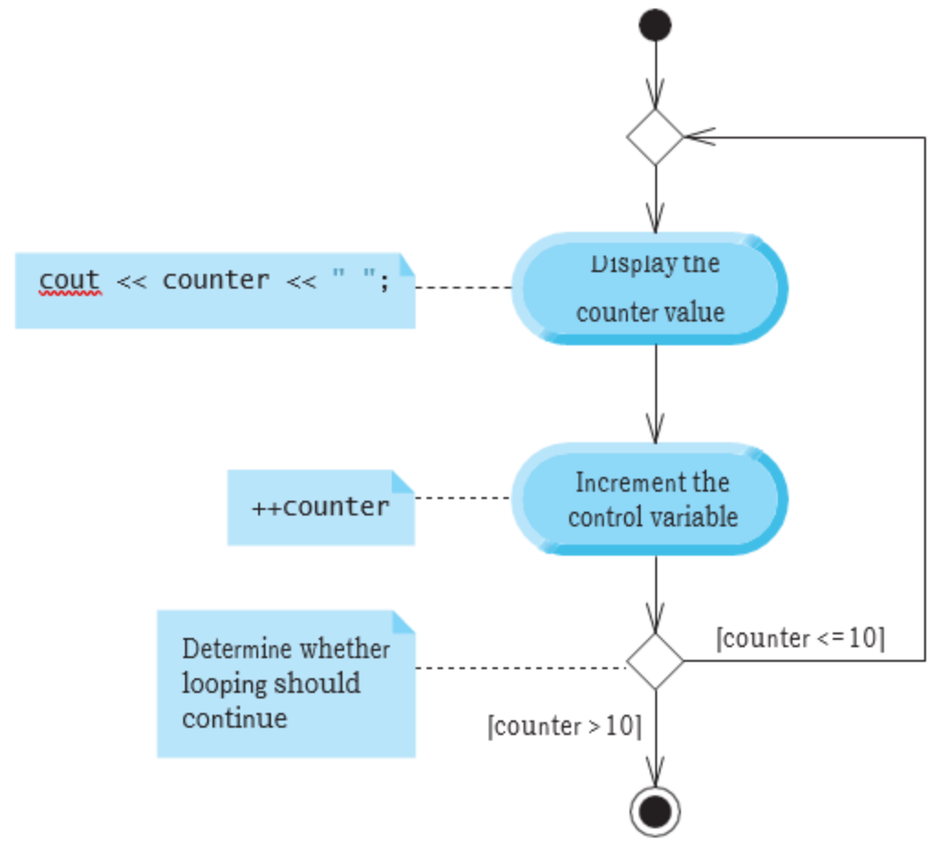
```
int main() {
   DollarAmount d1{12345}; // $123.45
   DollarAmount d2{1576}; // $15.76

   cout << "After adding d2 (" << d2.toString() << ") into d1 ("
      << d1.toString() << "), d1 = ";
   d1.add(d2); // modifies object d1
   cout << d1.toString() << "\n";

   cout << "After subtracting d2 (" << d2.toString() << ") from d1 ("
      << d1.toString() << "), d1 = ";
   d1.subtract(d2); // modifies object d1
   cout << d1.toString() << "\n";

   cout << "After subtracting d1 (" << d1.toString() << ") from d2 ("
      << d2.toString() << "), d2 = ";
   d2.subtract(d1); // modifies object d2
   cout << d2.toString() << "\n\n";

   cout << "Enter integer interest rate and divisor. For example:\n"
      << "for     2%, enter:    2 100\n"
      << "for   2.3%, enter:   23 1000\n"
      << "for  2.37%, enter:  237 10000\n"
      << "for 2.375%, enter: 2375 100000\n> ";
   int rate; // whole-number interest rate
   int divisor; // divisor for rate
   cin >> rate >> divisor;

   DollarAmount balance{100000}; // initial principal amount in pennies
   cout << "\nInitial balance: " << balance.toString() << endl;

   // display headers
   cout << "\nYear" << setw(20) << "Amount on deposit" << endl;

   // calculate amount on deposit for each of ten years
   for (unsigned int year{1}; year <= 10; year++) {
      // increase balance by rate % (i.e., rate / divisor)
      balance.addInterest(rate, divisor);

      // display the year and the amount
      cout << setw(4) << year << setw(20) << balance.toString() << endl;
   }
}
```

```
After adding d2 (15.76) into d1 (123.45), d1 = 139.21
After subtracting d2 (15.76) from d1 (139.21), d1 = 123.45
After subtracting d1 (123.45) from d2 (15.76), d2 = -107.69

Enter integer interest rate and divisor. For example:
for     2%, enter:    2 100
for   2.3%, enter:   23 1000
for  2.37%, enter:  237 10000
for 2.375%, enter: 2375 100000
> 5 100

Initial balance: 1000.00

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.29
   6             1340.10
   7             1407.11
   8             1477.47
   9             1551.34
  10             1628.91
```

```
After adding d2 (15.76) into d1 (123.45), d1 = 139.21
After subtracting d2 (15.76) from d1 (139.21), d1 = 123.45
After subtracting d1 (123.45) from d2 (15.76), d2 = -107.69

Enter integer interest rate and divisor. For example:
for     2%, enter:    2 100
for   2.3%, enter:   23 1000
for  2.37%, enter:  237 10000
for 2.375%, enter: 2375 100000
> 525 10000

Initial balance: 1000.00

Year   Amount on deposit
   1             1052.50
   2             1107.76
```

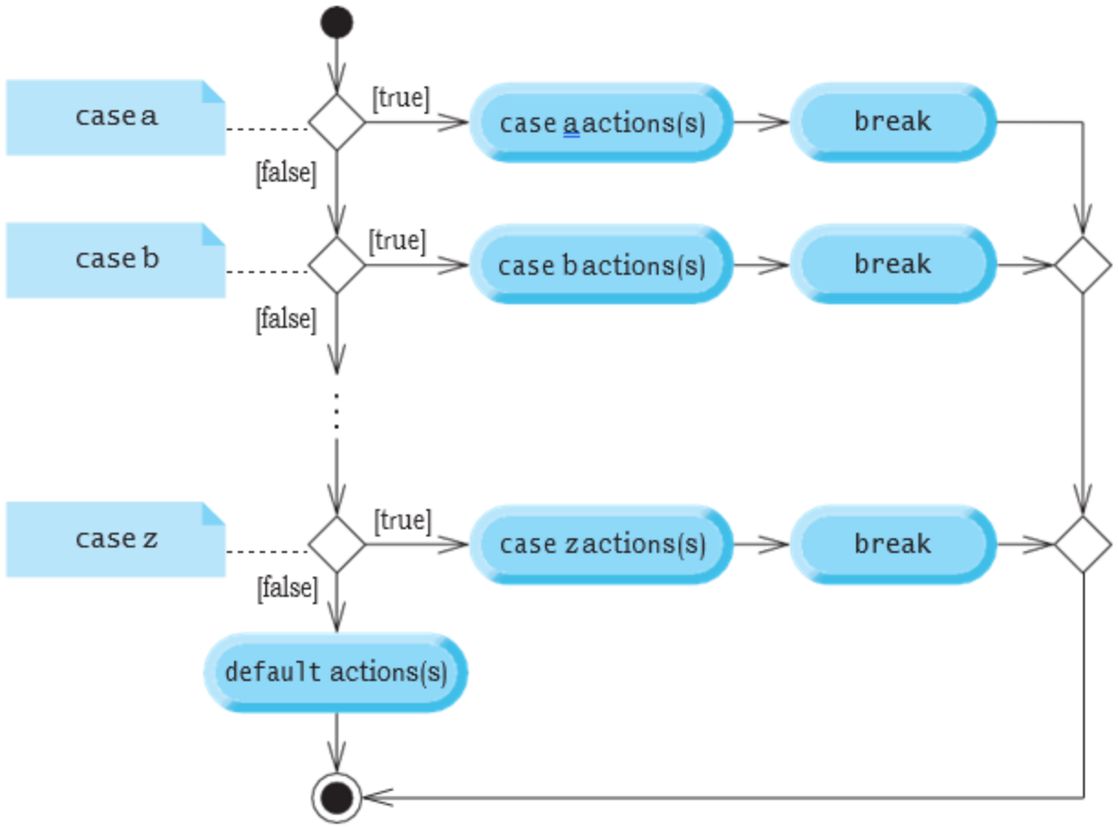
do…while Iteration Statement

```
#include <iostream>
using namespace std;

int main() {
   unsigned int counter{1};

   do {
      cout << counter << " ";
      ++counter;
   } while (counter <= 10); // end do...while

   cout << endl;

   return 0;
}
```



switch Multiple-Selection Statement

Lettergrade.cpp



Q & A