Revision in C++

Outline

1. **Introduction to C++**
2. **Control structures**
3. **Functions**
4. **Arrays**

```
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

1. Comments

2. Load `<iostream>`

3. `main`

3.1 Print `"Welcome to C++\n"`

3.2 exit (`return 0`)

Program Output

```
Welcome to C++!
```

Prints the *string* of characters contained between the quotation marks.

The entire line, including **`std::cout`**, the **`<<`** *operator*, the *string* **`"Welcome to C++!\n"`** and the *semicolon* (**`;`**), is called a *statement*.

All statements must end with a semicolon.

A Simple Program: Printing a Line of Text

- **`std::cout`**
  - Standard output stream object
  - “Connected” to the screen
  - **`std::`** specifies the "namespace" which **`cout`** belongs to
    - **`std::`** can be removed through the use of **`using`** statements
- **`<<`**
  - Stream insertion operator
  - Value to the right of the operator (right operand) inserted into output stream (which is connected to the screen)
  - **`std::cout << “Welcome to C++!\n”;`**
- **`\`**
  - Escape character
  - Indicates that a “special” character is to be output

A Simple Program: Printing a Line of Text

Escape Sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double quote character.

- There are multiple ways to print text
  - Following are more examples

```
// Fig. 1.4: fig01_04.cpp
// Printing a line with multiple statements
#include <iostream>

int main()
{
   std::cout << "Welcome ";
   std::cout << "to C++!\n";

   return 0;   // indicate that program ended successfully
}
```

1. Load `<iostream>`

2. `main`

2.1 Print `"Welcome"`

2.2 Print `"to C++!"`

2.3 newline

2.4 exit (`return 0`)

```
Welcome to C++!
```

Program Output

Unless new line `'\n'` is specified, the text continues on the same line.

Another Simple Program: Adding Two Integers

- Variables
  - Location in memory where a value can be stored for use by a program
  - Must be declared with a name and a data type before they can be used
  - Some common data types are:
    - **`int`** - integer numbers
    - **`char`** - characters
    - **`double`** - floating point numbers
  - Example: **`int myvariable;`**
    - Declares a variable named **`myvariable`** of type **`int`**
  - Example: **`int variable1, variable2;`**
    - Declares two variables, each of type **`int`**

Another Simple Program: Adding Two Integers

- **`>>`** (stream extraction operator)
  - When used with **`std::cin`**, waits for the user to input a value and stores the value in the variable to the right of the operator
  - The user types a value, then presses the *Enter* (Return) key to send the data to the computer
  - Example:

```
int myVariable;
std::cin >> myVariable;
```

    - Waits for user input, then stores input in **`myVariable`**
- **`=`** (assignment operator)
  - Assigns value to a variable
  - Binary operator (has two operands)
  - Example:

```
sum = variable1 + variable2;
```

```cpp
// Fig. 1.6: fig01_06.cpp
// Addition program
#include <iostream>

int main()
{
   int integer1, integer2, sum;           // declaration

   std::cout << "Enter first integer\n";  // prompt
   std::cin >> integer1;                  // read an integer
   std::cout << "Enter second integer\n"; // prompt
   std::cin >> integer2;                  // read an integer
   sum = integer1 + integer2;             // assignment of sum
   std::cout << "Sum is " << sum << std::endl; // print sum

   return 0;   // indicate that program ended successfully
}
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

1. Load `<iostream>`

2. `main`

2.1 Initialize variables `integer1`, `integer2`, and `sum`

2.2 Print "`Enter first integer`"

2.2.1 Get input

2.3 Print "`Enter second integer`"

2.3.1 Get input

2.4 Add variables and put result into `sum`

2.5 Print "`Sum is`"

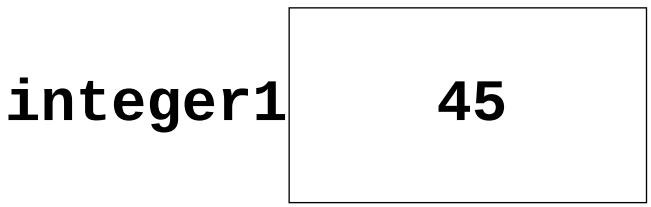
2.5.1 Output `sum`

2.6 exit (`return 0`)

Program Output

Memory Concepts

- Variable names
  - Correspond to locations in the computer's memory
  - Every variable has a name, a type, a size and a value
  - Whenever a new value is placed into a variable, it replaces the previous value - it is destroyed
  - Reading variables from memory does not change them
- A visual representation



Integer variables

```
// demonstrates integer variables and addition of values in them
#include <iostream>
using namespace std;
int main()
   {
   int var1;              //define var1
   int var2;              //define var2
   var1 = 20;                    //assign value to var1
   var2 = var1 + 10;          //assign value to var2
   cout << "var1+10 is ";  //output text
   cout << var2 << endl;   //output value of var2
   return 0;
   }
```

Floating point variables

```
// demonstrates floating point variables –calculates the area of a
//circle, given its radius by the
//formula, area = 3.14159 * radius *radius
#include <iostream>                    //for cout, etc.
using namespace std;
int main()
   {
   float rad;                          //variable of type float
   const float PI = 3.14159F;
   //type const float; here F at the right hand side of =, denotes a
     //floating point literal.
   cout << "Enter radius of circle: ";        //prompt
   cin >> rad;                                 //get radius
   float area = PI * rad * rad;                //find area
   cout << "Area is " << area << endl;     //display answer
   return 0;
   }
```

Arithmetic

- Arithmetic calculations
  - Use `*` for multiplication and `/` for division
  - Integer division truncates remainder
    - `7 / 5` evaluates to 1
  - Modulus operator returns the remainder
    - `7 % 5` evaluates to 2
- Operator precedence
  - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Be sure to use parenthesis when needed
  - Example: Find the average of three variables a, b and c
    - Do not use: `a + b + c / 3`
    - Use: `(a + b + c ) / 3`

Arithmetic

- Arithmetic operators:

C++ operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm	b * m
Division	/	x / y	x / y
Modulus	%	$r \bmod s$	r % s

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication Division Modulus	Evaluated second. If there are several, they re evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Decision Making: Equality and Relational Operators

- `if` structure
  - Test conditions truth or falsity. If condition met execute, otherwise ignore
- Equality and relational operators
  - Lower precedence than arithmetic operators
- Table of relational operators on next slide

Decision Making: Equality and Relational Operators

Standard algebraic equality operator or relational operator	C++ equality or relational operator	Example of C++ condition	Meaning of C++ condition
Relational operators			
$>$	**>**	**x > y**	**x** is greater than **y**
$<$	**<**	**x < y**	**x** is less than **y**
$\geq$	**>=**	**x >= y**	**x** is greater than or equal to **y**
$\leq$	**<=**	**x <= y**	**x** is less than or equal to **y**
Equality operators			
$=$	**==**	**x == y**	**x** is equal to **y**
$\neq$	**!=**	**x != y**	**x** is not equal to **y**

Control structures: The if Selection Structure

- Selection structure
  - used to choose among alternative courses of action
  - Pseudocode example:

    If student's grade is greater than or equal to 60

    Print "Passed"
  - If the condition is **`true`**
    - print statement executed and program goes on to next statement
  - If the condition is **`false`**
    - print statement is ignored and the program goes onto the next statement
  - Indenting makes programs easier to read
    - C++ ignores whitespace characters

The if Selection Structure

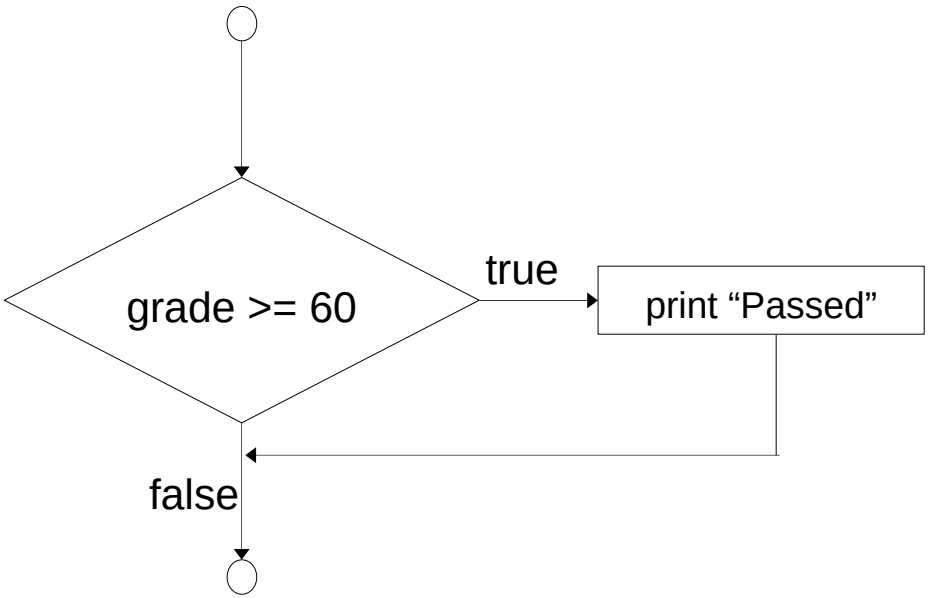
- Translation of pseudocode statement into C++:

```
if ( grade >= 60 )
   cout << "Passed";
```

- Diamond symbol (decision symbol)
  - indicates decision is to be made
  - Contains an expression that can be true or false.
    - Test the condition, follow appropriate path
- **`if`** structure is a single-entry/single-exit structure

The if Selection Structure

- Flowchart of pseudocode statement



A decision can be made on any expression.

zero - **`false`**

nonzero - **`true`**

Example:

`3 - 4` is **`true`**

The `if/else` Selection Structure

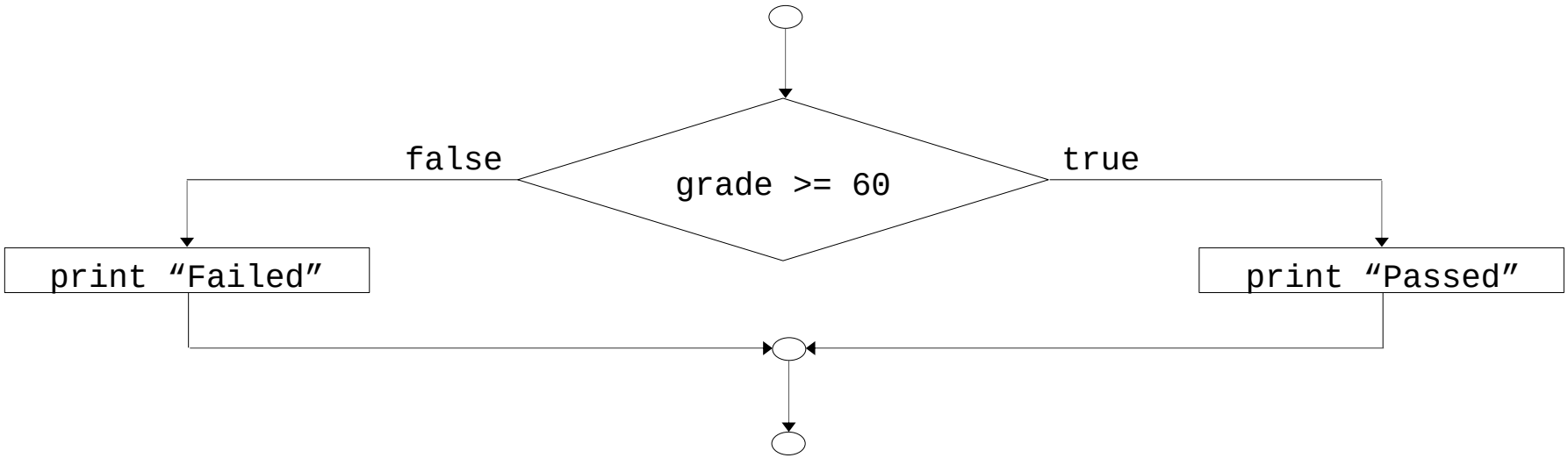
- **`if`**
  - Only performs an action if the condition is true
- **`if/else`**
  - A different action is performed when condition is true and when condition is false
- Psuedocode

  if student's grade is greater than or equal to 60
  print "Passed"
  else
  print "Failed"

- C++ code

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

The `if/else` Selection Structure



- Ternary conditional operator (**`?:`**)
  - Takes three arguments (condition, value if **`true`**, value if **`false`**)
- Our pseudocode could be written:

```
cout << ( grade >= 60 ? “Passed” : “Failed” );
```

The `if/else` Selection Structure

- Nested **`if/else`** structures
  - Test for multiple cases by placing **`if/else`** selection structures inside **`if/else`** selection 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"
```

  - Once a condition is met, the rest of the statements are skipped

The if/else Selection Structure

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if ( grade >= 60 )
   cout << "Passed.\n";
else {
   cout << "Failed.\n";
   cout << "You must take this course
again.\n";
}
```

  - Without the braces,

```
cout << "You must take this course again.\n";
```

  would be automatically executed

- Block
  - Compound statements with declarations

The if/else Selection Structure

- Syntax errors
  - Errors caught by compiler
- Logic errors
  - Errors which have their effect at execution time
    - Non-fatal logic errors
      - program runs, but has incorrect output
    - Fatal logic errors
      - program exits prematurely

The while Repetition Structure

- Repetition structure
  - Programmer specifies an action to be repeated while some condition remains true
  - Psuedocode

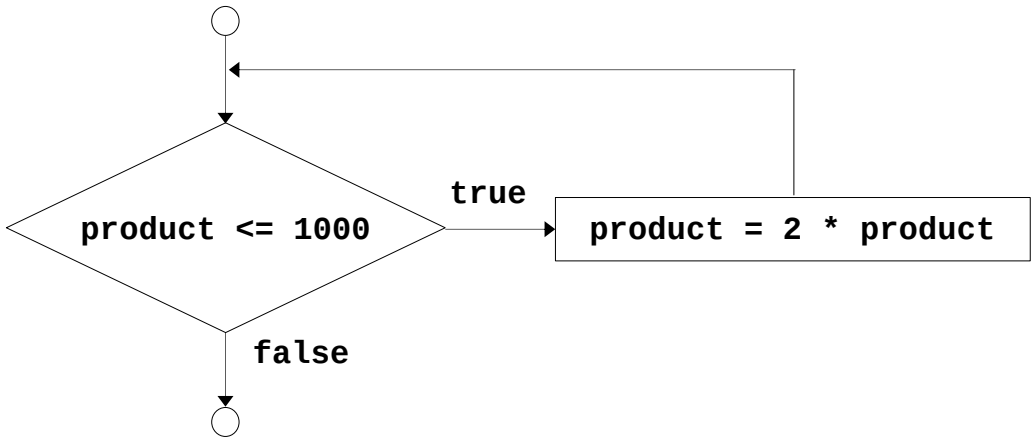
    while there are more items on my shopping list

    Purchase next item and cross it off my list
  - **`while`** loop repeated until condition becomes false.
- Example

```
int product = 2;
while ( product <= 1000 )
   product = 2 * product;
```

The while Repetition Structure

- Flowchart of **`while`** loop



Formulating Algorithms (Counter-Controlled Repetition)

- Counter-controlled repetition
  - Loop repeated until counter reaches a certain value.
- Definite repetition
  - Number of repetitions is known
- Example

  A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

Formulating Algorithms (Counter-Controlled Repetition)

- Pseudocode for example:

```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
        Input the next grade
        Add the grade into the total
        Add one to the grade counter
Set the class average to the total divided by ten
   Print the class average
```

- Following is the C++ code for this example

```cpp
// Fig. 2.7: fig02_07.cpp
// Class average program with counter-controlled repetition
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int total,        // sum of grades
       gradeCounter, // number of grades entered
       grade,        // one grade
       average;      // average of grades

   // initialization phase
   total = 0;                           // clear total
   gradeCounter = 1;                    // prepare to loop

   // processing phase
   while ( gradeCounter <= 10 ) {       // loop 10 times
      cout << "Enter grade: ";          // prompt for input
      cin >> grade;                     // input grade
      total = total + grade;            // add grade to total
      gradeCounter = gradeCounter + 1;  // increment counter
   }

   // termination phase
   average = total / 10;                // integer division
   cout << "Class average is " << average << endl;

   return 0;   // indicate program ended successfully
}
```

1. **Initialize Variables**
2. **Execute Loop**
3. **Output results**

Program Output

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Increment and Decrement Operators

- Increment operator (**++**) - can be used instead of **`c += 1`**
- Decrement operator (**`--`**) - can be used instead of **`c -= 1`**
  - Preincrement
    - When the operator is used before the variable (**`++c`** or **`–c`**)
    - Variable is changed, then the expression it is in is evaluated.
  - Posincrement
    - When the operator is used after the variable (**`c++`** or **`c--`**)
    - Expression the variable is in executes, then the variable is changed.
- If **`c = 5`**, then
  - **`cout << ++c;`** prints out **`6`** (**`c`** is changed before **`cout`** is executed)
  - **`cout << c++;`** prints out **`5`** (**`cout`** is executed before the increment. **c** now has the value of **6**)

Increment and Decrement Operators

- When Variable is not in an expression
  - Preincrementing and postincrementing have the same effect.

```
++c;
cout << c;
```

and

```
c++;
cout << c;
```

have the same effect.

The for Repetition Structure

- The general format when using **`for`** loops is

```
for ( initialization; LoopContinuationTest;
      increment )
   statement
```

- Example:

```
for( int counter = 1; counter <= 10; counter++ )
   cout << counter << endl;
```

  – Prints the integers from one to ten

No semicolon after last statement

The for Repetition Structure

- **`For`** loops can usually be rewritten as **`while`** loops:

```
initialization;
while ( loopContinuationTest){
   statement
   increment;
}
```

- Initialization and increment as comma-separated lists

```
for (int i = 0, j = 0;  j + i <= 10; j++, i++)
   cout << j + i << endl;
```

Examples Using the for Structure

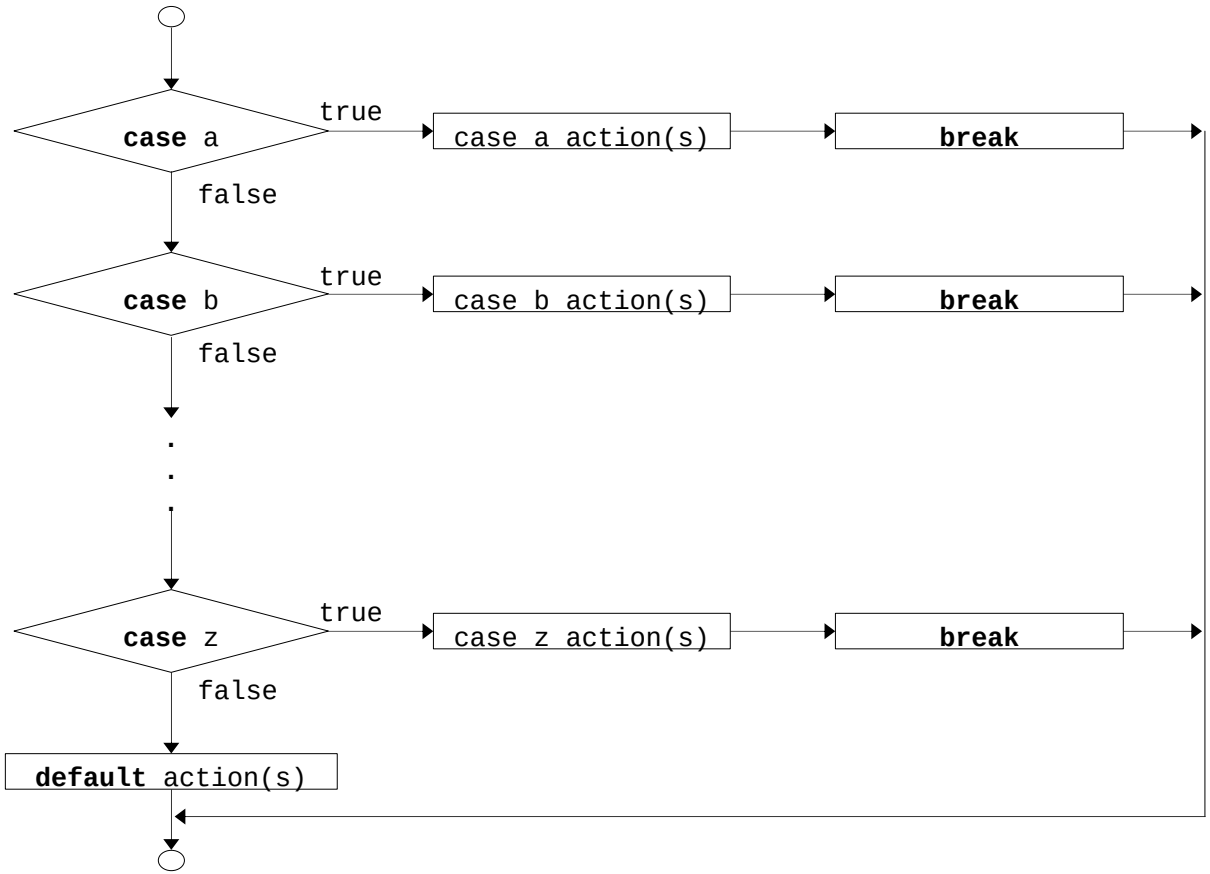
- Program to sum the even numbers from 2 to 100

```
// Fig. 2.20: fig02_20.cpp
// Summation with for
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int sum = 0;

   for ( int number = 2; number <= 100; number += 2 )
      sum += number;

   cout << "Sum is " << sum << endl;

   return 0;
}
```

```
Sum is 2550
```

The switch Multiple-Selection Structure

- **`switch`**
  - Useful when variable or expression is tested for multiple values
  - Consists of a series of **`case`** labels and an optional **`default`** case



```
// Fig. 2.22: fig02_22.cpp
// Counting letter grades
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int grade,       // one grade
       aCount = 0,  // number of A's
       bCount = 0,  // number of B's
       cCount = 0,  // number of C's
       dCount = 0,  // number of D's
       fCount = 0;  // number of F's

   cout << "Enter the letter grades." << endl
        << "Enter the EOF character to end input." << endl;

   while ( ( grade = cin.get() ) != EOF ) {

      switch ( grade ) {      // switch nested in while

         case 'A':  // grade was uppercase A
         case 'a':  // or lowercase a
            ++aCount;
            break;  // necessary to exit switch

         case 'B':  // grade was uppercase B
         case 'b':  // or lowercase b
            ++bCount;
            break;

```

1. Initialize variables

2. Input data

2.1 Use switch loop to update count

```
35          case 'C':  // grade was uppercase C
36          case 'c':  // or lowercase c
37             ++cCount;
38             break;
39
40          case 'D':  // grade was uppercase D
41          case 'd':  // or lowercase d
42             ++dCount;
43             break;
44
45          case 'F':  // grade was uppercase F
46          case 'f':  // or lowercase f
47             ++fCount;
48             break;
49
50          case '\n': // ignore newlines,
51          case '\t': // tabs,
52          case ' ':  // and spaces in input
53             break;
54
55          default:   // catch all other characters
56             cout << "Incorrect letter grade entered."
57                  << " Enter a new grade." << endl;
58             break;  // optional
59       }
60    }
61
62    cout << "\n\nTotals for each letter grade are:"
63         << "\nA: " << aCount
64         << "\nB: " << bCount
65         << "\nC: " << cCount
66         << "\nD: " << dCount
67         << "\nF: " << fCount << endl;
68
69    return 0;
70 }
```

2.1 Use switch loop to update count

3. Print results

Program Output

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

The do/while Repetition Structure

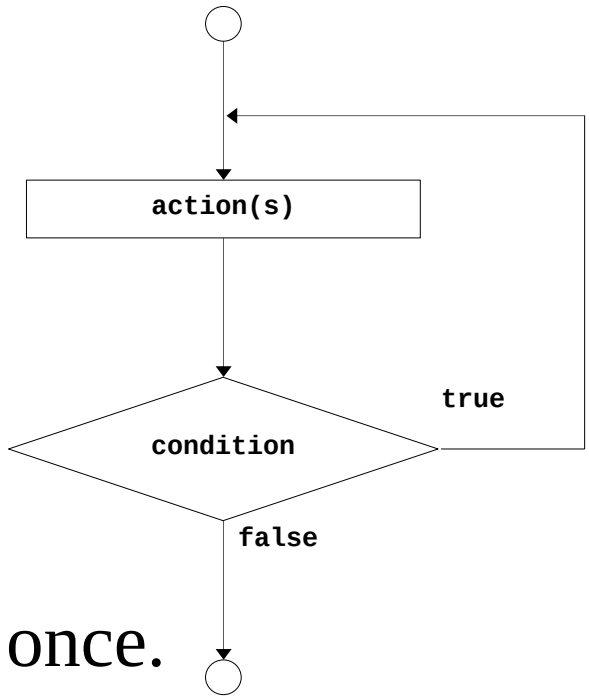
- The **`do/while`** repetition structure is similar to the **`while`** structure,
  - Condition for repetition tested after the body of the loop is executed
- Format:

```
do {
   statement
} while ( condition );
```

- Example (letting counter = 1):

```
do {
  cout << counter << " ";
} while (++counter <= 10);
```

  - This prints the integers from **`1`** to **`10`**
- All actions are performed at least once.



The break and continue Statements

- **`Break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement:
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure

The break and continue Statements

- **`Continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure and proceeds with the next iteration of the loop
  - In **`while`** and **`do/while`**, the loop-continuation test is evaluated immediately after the **`continue`** statement is executed
  - In the **`for`** structure, the increment expression is executed, then the loop-continuation test is evaluated

Logical Operators

- **`&&`** (logical **`AND`**)
  - Returns **`true`** if both conditions are **`true`**
- **`||`** (logical **`OR`**)
  - Returns **`true`** if either of its conditions are **`true`**
- **`!`** (logical **`NOT`**, logical negation)
  - Reverses the truth/falsity of its condition
  - Returns **`true`** when its condition is **`false`**
  - Is a unary operator, only takes one condition
- Logical operators used as conditions in loops

Expression	Result
`true && false`	`false`
`true || false`	`true`
`!false`	`true`

Functions

- Functions
  - Allow the programmer to modularize a program
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - Local variables passed when the function is called that provide the function with outside information

Function Definitions

- Create customized functions to
  - Take in data
  - Perform operations
  - Return the result
- Format for function definition:

  return-value-type function-name(*parameter-list*)
  {
  declarations and statements
  }
- Example:

```
int square( int y)
{
  return y * y;
}
```

```
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function
#include <iostream>

using std::cout;
using std::endl;

int square( int );   // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```

1. Function prototype
2. Loop
3. Function definition

Program Output

```
1  4  9  16  25  36  49  64  81  100
```

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;
```

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

```

   return 0;
}

// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

3. Function definition

Program Output

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 92 35 14
Maximum is: 92

Enter three integers: 45 19 98
Maximum is: 98
```

Function Prototypes

- Function prototype
  - Function name
  - Parameters
    - Information the function takes in
  - Return type
    - Type of information the function passes back to caller (default **`int`**)
    - **`void`** signifies the function returns nothing
  - Only needed if function definition comes after the function call in the program
- Example:

```
int maximum( int, int, int );
```

  - Takes in 3 **`int`**s
  - Returns an **`int`**

Assignment Operators

- Assignment expression abbreviations

  `c = c + 3;` can be abbreviated as `c += 3;` using the addition assignment operator

- Statements of the form

```
variable = variable operator expression;
```

  can be rewritten as

```
variable operator= expression;
```

- Examples of other assignment operators include:

```
d -= 4      (d = d - 4)
e *= 5      (e = e * 5)
f /= 3      (f = f / 3)
g %= 9      (g = g % 9)
```

References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Used to prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original
- Reference parameter alias for argument
  - **&** is used to signify a reference

```
void change( int &variable )
    { variable += 3; }
```

  - Adds 3 to the variable inputted

```
int &y = x.
```

  - A change to **y** will now affect **x** as well

```
// Fig. 3.20: fig03_20.cpp
// Comparing call-by-value and call-by-reference
// with references.
#include <iostream>

using std::cout;
using std::endl;

int squareByValue( int );
void squareByReference( int & );

int main()
{
   int x = 2, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;
}

int squareByValue( int a )
{
   return a *= a;   // caller's argument not modified
}
```

Notice the use of the **&** operator

1. Function prototypes

1.1 Initialize variables

2. Print x

2.1 Call function and print x

2.2 Print z

2.3 Call function and print z

3. Function Definition of `squareByValue`

```
void squareByReference( int &cRef )
{
   cRef *= cRef;    // caller's argument modified
}
```

3.1 Function Definition of `squareByReference`

Program Output

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Arrays

- Array
  - Consecutive group of memory locations
  - Same name and type
- To refer to an element, specify
  - Array name and position number
- Format: *arrayname[position number]*
  - First element at position 0
  - **n** element array **c**:

    c[0], **c[1]**…**c[n - 1]**
- Array elements are like normal variables

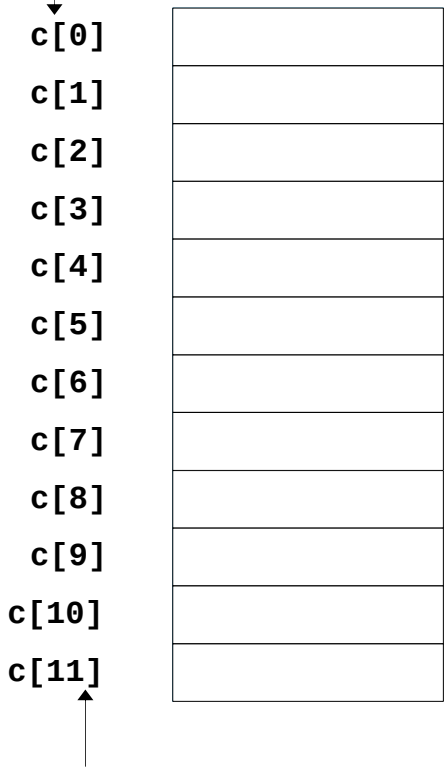
```
c[ 0 ] =  3;
cout << c[ 0 ];
```

- Performing operations in subscript. If **x = 3**,

```
c[ 5 - 2 ] == c[ 3 ] == c[ x ]
```

Arrays

Name of array (Note that all elements of this array have the same name, **c**)



Position number of the element within array **c**

Declaring Arrays

- Declaring arrays - specify:
  - Name
  - Type of array
  - Number of elements
  - Examples

```
int c[ 10 ];
float hi[ 3284 ];
```

- Declaring multiple arrays of same type
  - Similar format as other variables
  - Example

```
int b[ 100 ], x[ 27 ];
```

Examples Using Arrays

- Initializers

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

  - If not enough initializers, rightmost elements become 0
  - If too many initializers, a syntax error is generated

```
int n[ 5 ] = { 0 }
```

  - Sets all the elements to **0**

- If size omitted, the initializers determine it

```
int n[] = { 1, 2, 3, 4, 5 };
```

  - 5 initializers, therefore **n** is a 5 element array

```
// Fig. 4.4: fig04_04.cpp
// Initializing an array with a declaration
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;
}
```

Notice how they array is declared and elements referenced.

1. Initialize array using a declaration

2. Define loop

3. Print out each array element

Program Output

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Examples Using Arrays

- Strings
  - Arrays of characters
  - All strings end with **`null`** (**`'\0'`**)
  - Examples:

```
char string1[] = "hello";
char string1[] = { 'h', 'e', 'l', 'l', 'o',
                   '\0' };
```

  - Subscripting is the same as for a normal array

    `String1[ 0 ]` is **`'h'`**

    `string1[ 2 ]` is **`'l'`**

- Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

  - Takes user input
  - Side effect: if too much text entered, data written beyond array

```
// Fig. 4_12: fig04_12.cpp
// Treating character arrays as strings
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   char string1[ 20 ], string2[] = "string literal";

   cout << "Enter a string: ";
   cin >> string1;
   cout << "string1 is: " << string1
        << "\nstring2 is: " << string2
        << "\nstring1 with spaces between characters is:\n";

   for ( int i = 0; string1[ i ] != '\0'; i++ )
      cout << string1[ i ] << ' ';

   cin >> string1;  // reads "there"
   cout << "\nstring1 is: " << string1 << endl;

   cout << endl;
   return 0;
}
```

Inputted strings are separated by whitespace characters. **"there"** stayed in the buffer.

Notice how string elements are referenced like arrays.

1. Initialize strings

2. Print strings

2.1 Define loop

2.2 Print characters individually

2.3 Input string

ing

Program Output

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
string1 is: there
```

Multiple-Subscripted Arrays

- Multiple subscripts - tables with rows, columns
  - Like matrices: specify row, then column.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Column subscript

Array name

Row subscript

- Initialize

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

1	2
3	4

  - Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

Multiple-Subscripted Arrays

- Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

  - Will output the value of **0**
  - Cannot reference with commas

```
cout << b( 0, 1 );
```

    - Will try to call function **b**, causing a syntax error

SUMMARY

- By the end of this revision session, students would acquire a good idea on:

  – Introduction to C++ syntax
  – Control structures
  – Functions
  – Arrays