جامعة
الملك سعود
King Saud University

Control Structures
Decisive making in C++

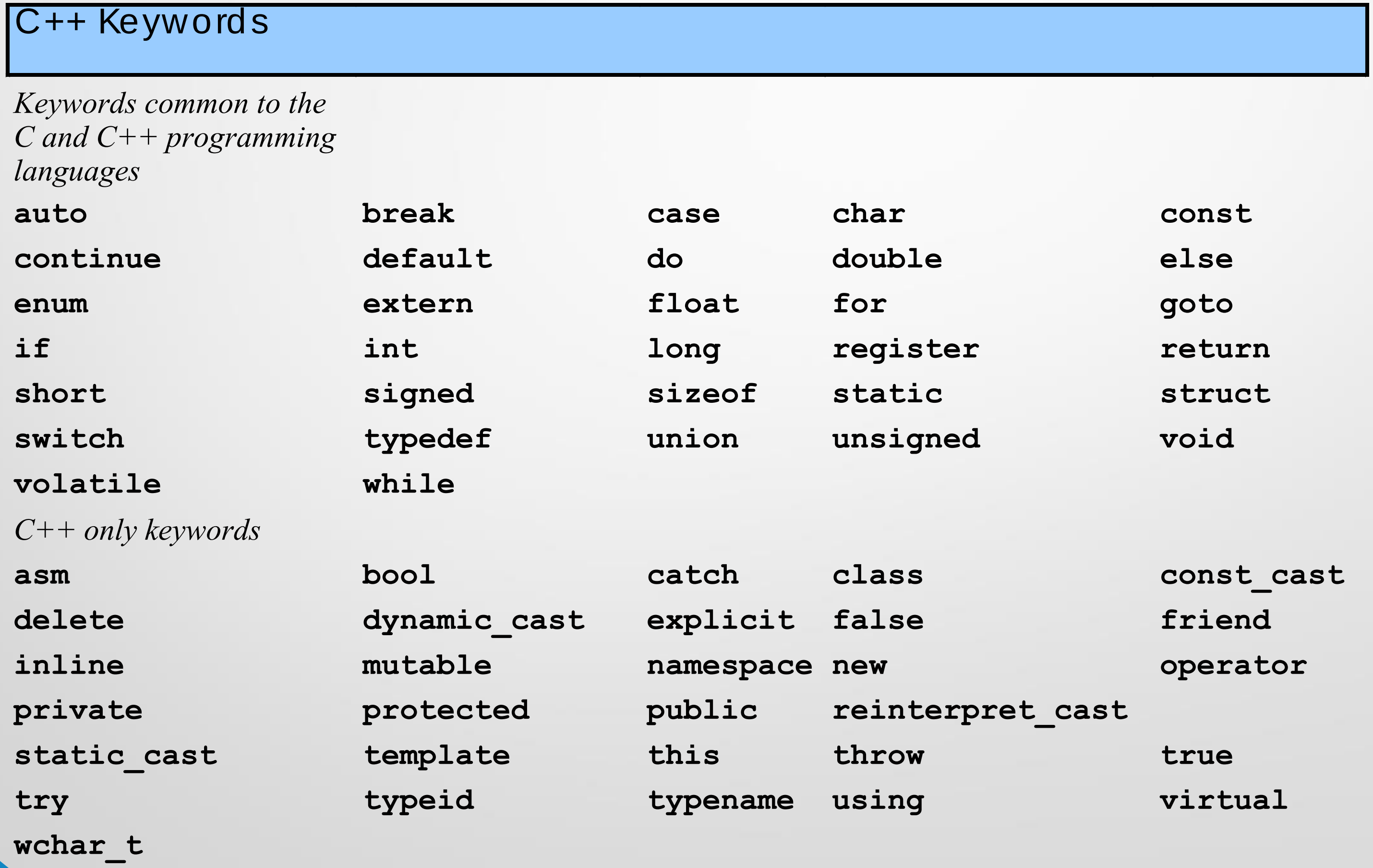
Lecture 4

Description

In this chapter we will learn how to use the repetition condition such as while loop for counter controlled and sentinel controlled repetition in our program.

Control Structures

- C++ keywords
  - Cannot be used as identifiers or variable names



C++ Keywords				
Keywords common to the C and C++ programming languages				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			
C++ only keywords				
asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

While Repetition Structure

- Repetition structure
  - Action 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;
```

Formulating Algorithms (Counter-Controlled Repetition)

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

fig02_07.cpp
(1 of 2)

```
 // Fig. 2.7: fig02_07.cpp
 // Class average program with counter-controlled repetition.
 #include <iostream>
 using namespace std;
 // function main begins program execution
int main()
{
   int total;        // sum of grades input by user
   int gradeCounter; // number of grade to be entered next
   int grade;        // grade value
   int average;      // average of grades

   // initialization phase
   total = 0;          // initialize total
   gradeCounter = 1;   // initialize loop counter
```

fig02_07.cpp
(2 of 2)

```
// processing phase
  while ( gradeCounter <= 10 ) {      // loop 10 times
    cout << "Enter grade: ";          // prompt for input
    cin >> grade;                     // read grade from user
    total = total + grade;            // add grade to total
    gradeCounter = gradeCounter + 1;  // increment counter
  }
 // termination phase
average = total / 10;                 // integer division
cout.setf (ios::fixed)
cout.setf(ios::showpoint);
cout.precision(2);
  // display result
  cout << "Class average is " << average << endl;
   return 0;   // indicate program ended successfully
   } // end function main
```

The counter gets incremented each time the loop executes. Eventually, the counter causes the loop to end.

fig02_07.cpp
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
```

Formulating Algorithms (Sentinel-Controlled Repetition)

- Suppose problem becomes:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run

  - Unknown number of students
  - How will program know when to end?
- Sentinel value
  - Indicates “end of data entry”
  - Loop ends when sentinel input
  - Sentinel chosen so it cannot be confused with regular input
    - -1 in this case

Formulating Algorithms (Sentinel-Controlled Repetition)

- Many programs have three phases
  - Initialization
    - Initializes the program variables
  - Processing
    - Input data, adjusts program variables
  - Termination
    - Calculate and print the final results
  - Helps break up programs for top-down refinement

fig02_09.cpp
(1 of 3)

```
// Fig. 2.9: fig02_09.cpp
// Class average program with sentinel-controlled repetition.
    #include <iostream>
   #include <iomanip>       // parameterized stream manipulators
   using namespace std;
// sets numeric output precision
// function main begins program execution
int main()
 {
  int total;        // sum of grades
      int gradeCounter;  // number of grades entered
      int grade;         // grade value

      double average;    // number with decimal point for average

      // initialization phase
      total = 0;         // initialize total
     gradeCounter = 0;  // initialize loop counter
```

Data type **double** used to represent decimal numbers.

fig02_09.cpp

(2 of 3)

```
    // processing phase
   // get first grade from user
   cout << "Enter grade, -1 to end: ";  // prompt for input
   cin >> grade;                        // read grade from user
   // loop until sentinel value read from user
   while ( grade != -1 ) {
      total = total + grade;
      gradeCounter = gradeC
       cout << "Enter grade,
      cin >> grade;
   } // end while
   // termination phase
   // if user entered at least one grade ...
   if ( gradeCounter != 0 ) {
    // calculate average of all grades entered
      average = static_cast< double >( total ) / gradeCounter
```

`static_cast<double>()` treats **`total`** as a **`double`** temporarily (casting).

Required because dividing two integers truncates the remainder.

`gradeCounter` is an **`int`**, but it gets *promoted* to **`double`**.

```
;

        // display average with two digits of precision
       cout.setf (ios::fixed);
       cout.setf(ios::showpoint);
       cout.precision(2);
     cout << "Class average is " <<average << endl;
```

fixed forces output to print in fixed point format (not scientific notation). Also, forces trailing zeros and decimal point to print. Include **<iostream>**

setprecision(2) prints two digits past decimal point (rounded to fit precision). Programs that use this must include **<iomanip>**

```
   } // end if part of if/else

   else // if no grades were entered, output appropriate message
      cout << "No grades were entered" << endl;

   return 0;   // indicate program ended successfully

} // end function main
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Nested Control Structures

- Problem statement

  A college has a list of test results (1 = pass, 2 = fail) for 10 students. Write a program that analyzes the results. If more than 8 students pass, print "Raise Tuition".

- Notice that
  - Program processes 10 results
    - Fixed number, use counter-controlled loop
  - Two counters can be used
    - One counts number that passed
    - Another counts number that fail
  - Each test result is 1 or 2
    - If not 1, assume 2

fig02_11.cpp
(1 of 2)

```
// Fig. 2.11: fig02_11.cpp
// Analysis of examination results.
#include <iostream>
using namespace std;
// function main begins program execution
int main()
{
   // initialize variables in declarations
   int passes = 0;           // number of passes
   int failures = 0;         // number of failures
   int studentCounter = 1;   // student counter
   int result;               // one exam result
// 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): ";
      cin >> result;

```

fig02_11.cpp
(2 of 2)

```
      // if result 1, increment passes; if/else nested in while
      if ( result == 1 )       // if/else nested in while
         passes = passes + 1;
      else  // if result not 1, increment failures
         failures = failures + 1;
// increment studentCounter so loop eventually terminates
      studentCounter = studentCounter + 1;
   } // end while
          // termination phase; display number of passes and failures
   cout << "Passed " << passes << endl;
   cout << "Failed " << failures << endl;
          // if more than eight students passed, print "raise tuition"
   if ( passes > 8 )
      cout << "Raise tuition " << endl;
          return 0;   // successful termination

} // end function main
```

fig02_11.cpp
output (1 of 1)

```
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
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): 2
Passed 6
Failed 4
```

```
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): 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
Passed 9
Failed 1
Raise tuition
```

Assignment Operators

- Assignment expression abbreviations
  - Addition assignment operator

    `c = c + 3;` abbreviated to

    `c += 3;`
- Statements of the form

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

can be rewritten as

```
variable operator= expression;
```

- Other assignment operators

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

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.

Increment and Decrement Operators

- Increment operator (**++**)
  - Increment variable by one
  - **`c++`**
    - Same as **`c += 1`**
- Decrement operator (**--**) similar
  - Decrement variable by one
  - **`c--`**

Increment and Decrement Operators

- Preincrement
  - Variable changed before used in expression
    - Operator before variable (**`++c`** or **`--c`**)

- Postincrement
  - Incremented changed after expression
    - Operator after variable (**`c++, c--`**)

Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - Name of control variable/loop counter
  - Initial value of control variable
  - Condition to test for final value
  - Increment/decrement to modify control variable when looping

```
// Fig. 2.16: fig02_16.cpp (1 of 1)
// Counter-controlled repetition.
#include <iostream>
using namespace std;
// function main begins program execution
int main()
{
   int counter = 1;            // initialization

   while ( counter <= 10 ) {   // repetition condition
      cout << counter << endl; // display counter
      ++counter;               // increment

   } // end while

   return 0;  // indicate successful termination

} // end function main
```

Summary

- ♦ Understanding the repetitive statement such as while loop
- ♦ Counter-controlled loop
- ♦ Sentinel-controlled loop
- ♦ Increment and decrement operator
- ♦ Difference between pre-increment and post-increment

Thank You