COMP1011
Programming Fundamentals

Lecture 4
Control Structures III

Lecture 4

- More repetition control structures
  - **`for`**, **`do-while`**
- **`break`** and **`continue`** statements
- Dry Run Table
- Random Number Generation
- Introduction to Array

for Repetition Structure

```
// Demonstrating Repetition Structure
// To print from 1 to 100
#include <iostream>

using namespace std;

int main() {

    for (int counter = 1; counter <= 100; counter++) {
        cout << counter << endl;
    }

    return 0;
}
```

```
1
2
3
(More numbers are to be printed. We skip here.)
```

for Repetition Structure

- General format

```
for(initialization; loop continuation test; increment) {
    statement(s)
}
```

- Example

```
for(int counter = 1; counter <= 100; counter++)  {
    cout << counter << endl;
}
```

It prints out 1 to 100.

```
for (counter = 1; counter <= 100; counter++)
{
        cout << counter << “ “;
}
```

1) Execute initialization.

counter

1

Output:

```
for (counter = 1; counter <= 100; counter++)
{
        cout << counter << “ “;
}
```

true

2) Check condition.

counter

1

Output:

```
for (counter = 1; counter <= 100; counter++)
{
        cout << counter << “ “;
}
```

3) The condition is true.
Execute loop body statement

counter

1

Output:

```
1
```

```
for (counter = 1; counter <= 100; counter++)
{
        cout << counter << “ “;
}
```

4) Update the counter.

counter

2

Output:

```
1
```

```
for (counter = 1; counter <= 100; counter++)
{
        cout << counter << “ “;
}
```

true

5) Check condition again.

counter

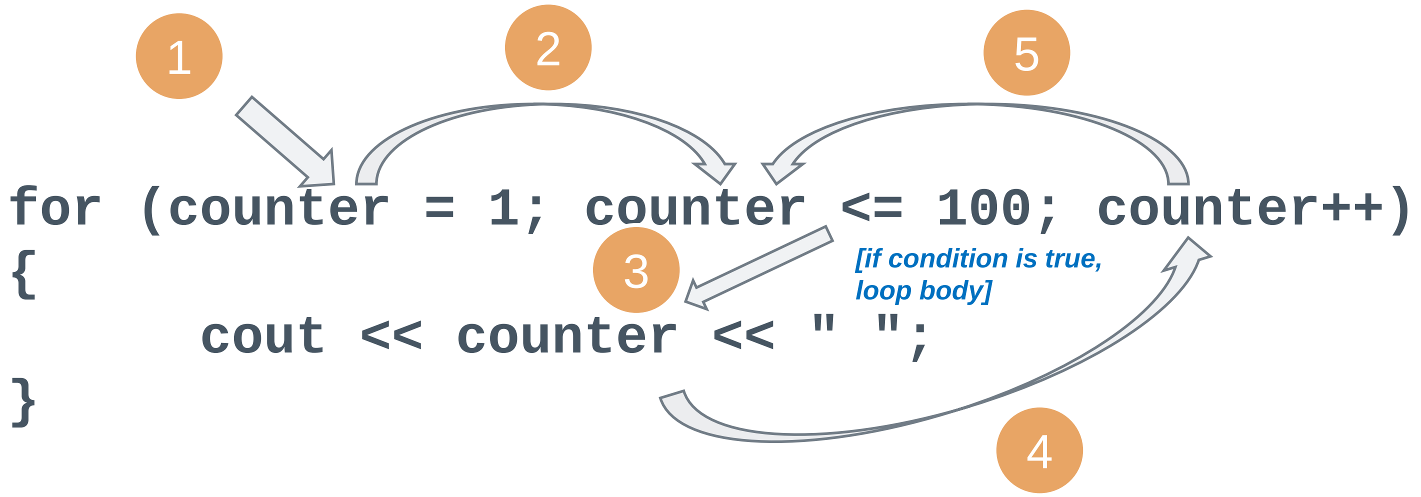
2

Output:

```
1
```

for Repetition Structure

- An illustration of the flow of for-loop



```
for (counter = 1; counter <= 100; counter++)
{
      cout << counter << " ";
}
```

Loop through steps 3 – 5 until counter <= 100 is false.

for Repetition Structure

- for loops can be rewritten as while loops

```
initialization;
while (loop continuation test) {
   statement(s)
   increment;
}
```

- Initialization and increment
  - For multiple variables, use comma-separated lists

```
for (int i = 0, j = 0; j + i <= 10; i++, j++) {
   cout << 2 * j + i << endl;
}
```

```
// Class average program with counter-controlled repetition.
// for repetition structure
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
     double total;        // sum of marks input by user
     double mark;         // mark value
     double average;      // average of marks
     const int CLASS_SIZE = 10; // class size

     // initialization phase
     total = 0;        // initialize total

     // processing phase
     for (int counter = 1; counter <= CLASS_SIZE; counter++) { // loop 10 times
          cout << "Enter a mark: ";  // prompt for input
          cin >> mark;               // read mark from user
          total = total + mark;      // add mark to total
     }

     // termination phase
     average = total / CLASS_SIZE;         // integer division

     // display result
     cout << "Class average is " << setprecision(2) << fixed << average << endl;

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

Formatting Decimal Numbers

- **`setprecision(2)`**
  - prints 2 digits past decimal point (rounded to fit precision)
- **`fixed`**
  - forces output to print in fixed point format (not scientific notation)
  - forces trailing zeros and decimal point to print
- Programs that use the above must contains the **`#include <iomanip>`** preprocessing directive

Exercise

- Write a program to print the balance of the savings account over a n-year period
- Initial balance: $10,000
- Annual interest rate: 5%
- Do not use the function **`pow()`**
- If n is equal to 5, the output should look something like that:

```
Enter number of years: 5
  1  10500.00
  2  11025.00
  3  11576.25
  4  12155.06
  5  12762.82
```

Exercise

- Write down your code here:

`do-while` Repetition Structure

- Similar to **`while`** structure
  - Makes *loop continuation test* at the end, not at the beginning
  - Loop body executes at least once
- Format:

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

do-while Repetition Structure

```cpp
// Demonstrating do-while Repetition Structure
// To print from 1 to 100
#include <iostream>

using namespace std;

int main() {

    int counter = 1;

    do {
        cout << counter << endl;
        counter++;
    } while (counter <= 100);

    return 0;
}
```

```
1
2
3
(More numbers are to be printed. We skip here.)
```

`do-while` Repetition Structure

- `do-while` is particularly useful for input validation
  - Logic:

```
do {
        user input;
} while (user input does not follow input specification);
```

- Read the example in the next slide
  - What is the input specification?

do-while Repetition Structure

```cpp
// Demonstrating do-while Repetition Structure
// An input validation example
#include <iostream>

using namespace std;

int main() {

    char input;

    do {

        cout << "Do you want to quit?" << endl;
        cin >> input;

    } while (!(input == 'Y' || input == 'y' || input == 'N' || input == 'n'));

    if (input == 'Y' || input == 'y') {
        cout << "Bye!" << endl;
    }
    else {
        cout << "The program continues." << endl;
    }

    return 0;
}
```

`break` statement

- Immediate exit from current **`switch`**, **`while`**, **`for`**, or **`do-while`** control structures
- Program continues with the immediate statement after the structure
- Common uses
  - Skip the remaining part of **`switch`**
  - Escape early from a loop

break statement

```
// Demonstrating the break statement
#include <iostream>

using namespace std;

int main() {

    for (int counter = 1; counter <= 1000; counter++) {
        cout << counter << endl;
        if (counter == 500) {
                cout << "End earlier." << endl;
                break;
        }
    }
    return 0;
}
```

`continue` statement

- Used in **`while`**, **`for`**, or **`do-while`** control structures
- Skip the rest of the statements after **`continue`** within the structure, and go directly to
  - increment part
    - **`for`**
  - condition-checking part
    - **`while`**
    - **`do-while`**

continue statement

```cpp
// Demonstrating the continue statement
#include <iostream>

using namespace std;

int main() {

    for (int counter = 1; counter <= 10; counter++) {
        if (counter == 5) {
                continue;
        }
        cout << counter << " ";
    }

    cout << endl;

    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
```

`break` `and` `continue` statements

- In repetition structures, try to avoid using **`break`** and **`continue`** statements
  - **`break`** and **`continue`** make the program difficult to follow and debug
  - There must be a way to rewrite the structures to have the same logic
- Can you rewrite the programs on Slides 16 and 18 without using **`break`** and **`continue`**?

Dry Run Tables

- Consider the following program code

```
#include <iostream>

using namespace std;

int main() {

    int n;
    int sum = 0;
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cout << sum + i * i << endl;
        sum += i * i;
    }

    return 0;
}
```

Dry Run Tables

- Sometimes you may be confused when tracing the flow of a repetition structure (loop)
- Do not always rely on the debugger tool!
  - Train up your brain!!!
- The best way is to
  - get a pencil and a piece of paper
  - write down the values of the variables in every step of the loop
  - get a “feeling” of the logic

Dry Run Tables

Step	i	i * i	sum
1	1	1	1
2	2	4	5
3	3	9	14
4	4	16	30
5	5	25	55
6	6	36	91
7	7	49	140
8	8	64	204
9	9	81	285
10	10	100	385

Random Number Generation

- Generating random numbers is common in programming
- Applications
  - Computer games
  - Gambling programs
  - Modern data encryption

Random Number Generation

- `rand()` function
  - `unsigned int i = rand();`
  - Generates an integer between **`0`** and **`RAND_MAX`** (usually 32767)
- However, most likely we need a random number that does not fall in this range
  - So, we need **Scaling** and **Shifting**
    - Modulus (remainder) operator: %
      - 10 % 3 is 1
      - x % y falls between 0 and y – 1
    - Example

```
        i = rand() % 6 + 1;
```

      - `rand() % 6` generates a number between 0 and 5 (scaling)
      - +1 makes the range 1 to 6 (shifting)

Random Number Generation

- Calling **`rand()`** repeatedly
  - Gives the same sequence of numbers
- Pseudorandom numbers
  - Preset sequence of “random” numbers
  - Same sequence generated
- To get different random sequences
  - Provide a seed value
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
  - **`srand(seed);`**
    - `seed` is an unsigned integer
    - Used before `rand()` to set the seed
    - Using ONCE is enough throughout the program

Random Number Generation

- Can use the current time to set the seed
  - Why? The time is always changing!
  - No need to explicitly set seed every time

```
srand(time(0));
```

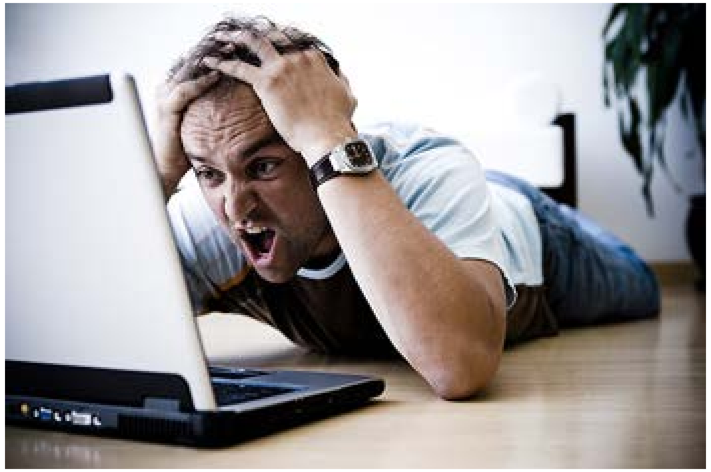
  - `time(0)`
    - returns current time in seconds since January 1, 1970

Random Number Generation

```cpp
#include <iostream>
#include <time.h>

using namespace std;

int main() {

    srand(time(0));
    unsigned int i = rand();
    cout << "The random number is: " << i << endl;

    i = rand();
    cout << "The next random number is: " << i << endl;

    return 0;
}
```

What is Array?

- Suppose we need to solve a programming problem that involves a large set of numbers.
- According to what we have learnt so far, we have to declare the variables separately.
  - Tedious!



Source: https://1funny.com/wp-content/uploads/2009/11/frustrated-laptop.jpg

What is Array?

```cpp
#include <iostream>

using namespace std;

int main() {

    int number1;
    int number2;
    int number3;
    int number4;
    int number5;
    int number6;
    int number7;
    int number8;
    int number9;
    int number10;

    // The logic . . .
}
```

What is Array?

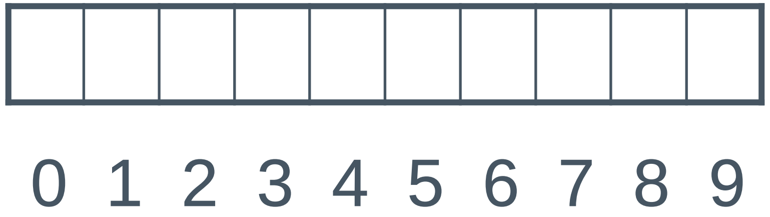
- A collection of variables having the **same data type**
- Variables in an array are arranged consecutively in memory
- Only one variable name represents all variables in the array
- An index (starting with **0**) is used to identify each variable in the array

Array

- To declare an array of 10 integers, we write,

```
int number[10];
```

- An array of 10 integer variables are created



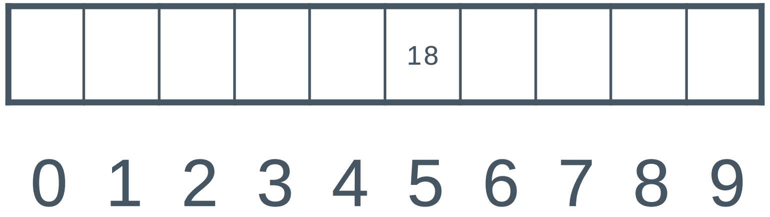
0 1 2 3 4 5 6 7 8 9

Array

- If we write,

```
number[5] = 18;
```

- 18 will be stored in the **6th element** **(index 5)** of the array



					18				
0	1	2	3	4	5	6	7	8	9

Array

- General Format
  - Declaration

    ```
    data-type array-name[array-size]
    ```

    - E.g.,

      ```
      char letter[15];
      ```
  - Usage

    ```
    array-name[index]
    ```

    - E.g.,

      ```
      letter[3] = 'A';
      ```

Array

- REMEMBER!
  - The index of an array starts from **ZERO**
  - Therefore, the variables in the array of the previous slide are
    - letter[0], letter[1], letter[2] … letter[14]
  - During *declaration*, the number represents the number of slots in the array
    - E.g., char letter[15];
    - But, this number cannot be used to access any of the slots of the array. Why?

Array

- Array elements behave like other variables
  - Assignment

```
number[0] = 3;
```

  - Printing an integer array element

```
cout << number[0];
```

- Can perform operations on the index (which is an integer)

`number[5 – 2]` same as **`number[3]`**

Array

- Declaring multiple arrays of same type
  - Use comma separated list, like regular variables

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

Initializing Arrays

- Using a loop
  - Set each element one by one

```
#include <iostream>

using namespace std;

int main() {

    int num[10];

    for (int i = 0; i < 10; i++) {
        num[i] = 0;
    }
}
```

Initializing Arrays

- Initializer list
  - Specify each element when array is declared

```
int n[5] = {7, 4, 3, 2, 8};
char abc[5] = {'H', 'e', 'l', 'l', 'o'};
```

- If not enough initializers, rightmost elements are automatically set to 0
  - For **char**, set to '\0' (called a NULL value), which represents all ZEROs to be stored in the memory location for that variable
- If too many, syntax error

Initializing Arrays

- To set every element to 0

```
int num[5] = {0};
```

  - Only we can set to all zero, but not other values. Why?

- If array size omitted, the size of array is determined automatically

```
int num[] = {5, 4, 1, 9, 3, 2};
```

  - The size of above array is 6

Importance of Initialization

- In C++, when a variable/array is declared, memory location(s) are allocated for holding values
- However, the value in the allocated memory location is not reset (to 0) by default
  - The current value is not predictable
- That is why we have to set the value explicitly before we use the variable
  - It is called **Initialization**
- In what circumstance that we do not have to initialize variables?

Array

- Note
  - For all arrays that you are currently using, you must specify the size of an array in your source code BEFORE compilation
  - For example,

```
const int n = 10;
int abc[n];         /* OK */
```

  - But,

```
int n;
cin >> n;
int abc[n];         /* Wrong! */
```

  - If the array is created DURING program execution, dynamic memory allocation is required.

Array Example 1

- The following program creates a 10-element integer array and displays the content in a tabular format.

Array Example 1

example1.cpp

```cpp
// Initializing an array with a declaration.
#include <iostream>
#include <iomanip>

using namespace std;

int main() {

    // use initializer list to initialize array n
    int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

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

    // output contents of array n in tabular format
    for (int i = 0; i < 10; i++) {
        cout << setw(7) << i << setw(13) << n[i] << endl;
    }

    return 0;
}
```

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

Array Example 2

- The following program calculates the sum of all values in a 10-element integer array.

Array Example 2

example2.cpp

```cpp
// Compute the sum of the elements of the array.
#include <iostream>

using namespace std;

int main() {

    const int ARRAY_SIZE = 10;

    int noList[ARRAY_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    int total = 0;

    // sum contents of array a
    for (int i = 0; i < ARRAY_SIZE; i++) {
        total += nolist[i];
    }

    cout << "Total of array element values is " << total << endl;

    return 0;
}
```

```
Total of array element values is 55
```

Summary

- More repetition control structures
  - `for`, `do-while`
- `break` and `continue` statements
- Dry Run Table
- Random Number Generation
- Introduction to Array