EEE102

C++ Programming and Software Engineering II

Lecture 2 From C to C++

Dr. Rui Lin / Qing Liu

Rui.Lin / Qing.Liu@xjtlu.edu.cn

Room EE512 / EE516

Office hour: 2-4pm, Tuesday & Wednesday

/ Monday & Wednesday

Outline

- Variable and constant
- Datatype
  - Built-in types
  - Derived types
  - Datatype casting
- Fundamental input and output
- Operators and expressions
- Control structures

1.1 Variable

- A *variable* is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents.
- Name of a variable
  - 1. Only alphabetic characters, numeric digits, and the underscore (_) can be used;
  - 2. The first character cannot be a numeric digit;
  - 3. Uppercase characters are considered distinct from lowercase characters;
  - 4. C++ keywords cannot be used as a name;
  - 5. No limit on the length of a name
- The C++ language is “case sensitive”

1.1 Variable

- It is encouraged to use meaningful names for variables
  - such as **dateOfBirth, my_age**
- Two popular ways of naming the variables:
  - *Camel case notation* is the practice of writing compound words without spaces, with each element's initial letter capitalized
    - **endOfFile**, **annualSalary**, and **AccountNum**.
  - *Hungarian notation*, in which a variable name starts with a group of lower-case letters representing the type of that variable, followed by whatever name the programmer has chosen
    - **fAnnualSalary** (variable is a floating-point number),
    - **lDateOfBirth** (variable is a long integer),
    - **arruMarkList** (variable is an array of unsigned integer numbers).

1.2 Constants

- Literals
  - Integer numbers
  - Floating pointer numbers
  - Characters and string literals
  - Boolean literals
- Defined literals (#define)
  - Using the #define preprocessor directive

    ```
    #define PI 3.14159
    ```
- Declared literals (const)
  - They are treated just like regular variables except that their values cannot be modified after their definition.

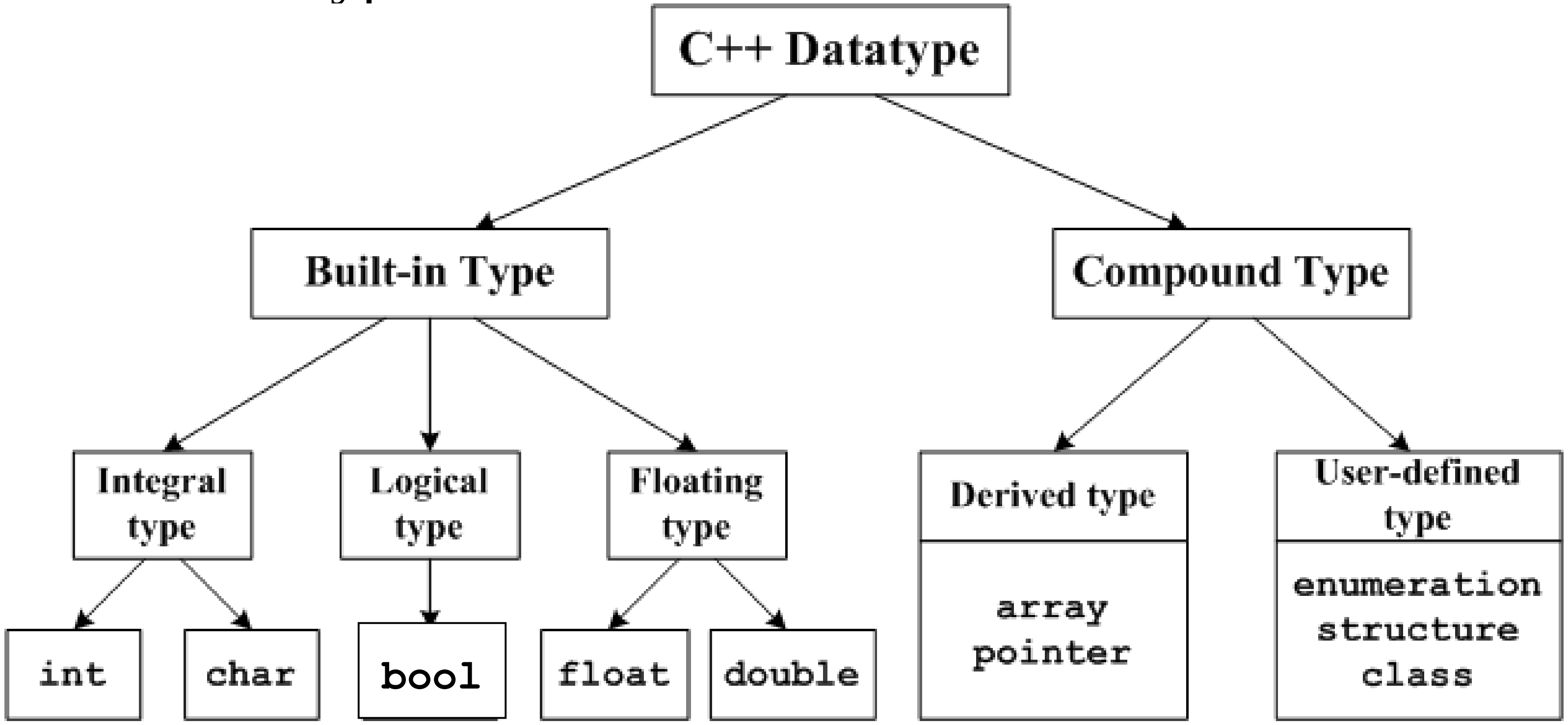
    ```
    const int pathwidth = 100;
    ```

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

#define PI 3.14159

int main ()
{
   double r=5.0;
   double circle;
   circle = 2 * PI * r;
   cout << circle;

   const float area=10.0f;
   area = circle; // wrong

   return 0;
}
```

Xi'an Jiaotong-Liverpool University

2. Datatype

- A datatype (or data type) is a classification identifying one of various types of data, which determines the possible values for that type, the operations that can be done on that type, and the way the values of that type are stored.



2.1 Built-in Type

Category	Datatype	Modifier	Size	Range
Integral type	`int`	`short`	2	-32,768 ~ 32,767
		`unsigned short`	2	0 ~ 65,535
		`long` (default)	4	-2^{31} ~ $(2^{31}-1)$
		`unsigned long`	4	0 ~ $(2^{32}-1)$
	`char`	`signed` (default)	1	-128 ~ 127
		`unsigned`	1	0 ~ 255
Logical type	`bool`	-	1	`true` (1) or `false` (0)
Floating type	`float`	-	4	3.4E-38 ~ 3.4E+38 (~7 digits)
	`double`	`long` (default)	8	1.7E-308 ~ 1.7E+308 (~15 digits)

Declaration

- Declare the variable before use it;
- Syntax:

```
datatype var_name1;
datatype var1, var2, … , varN;
```

Every variable must be declared before usage!

- Example:

```
int sum;
unsigned int counter;
char cMyInitial;
float fAverMark;
bool choice;
```

- After declaration, concrete values can be assignment to these variables.

Initialisation

- Initialise: to make a variable to store a concrete value at the same moment that it is declared;
- Initialisation is the combination of declaration and assignment;
- Syntax:
  - C-like initialisation

    ```
    datatype var_name = initial_value;
    ```
    - Eg: `int sum = 0;`

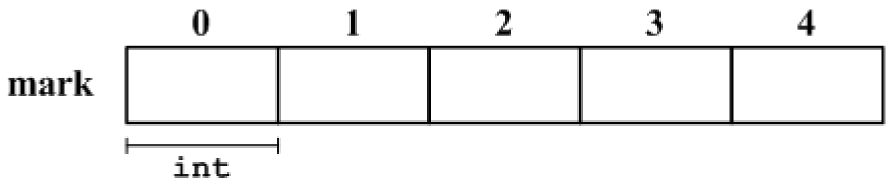
  - Constructor initialisation

    ```
    datatype var_name(initial_value);
    ```
    - Eg: `int sum(0);`

2.2 Derived Types

-- 2.2.1 Array (I)

- An *array* is a series of elements of the same type placed in continuous memory locations that can be individually referenced by adding an index to a unique identifier.
  - Eg: an array to contain 5 integer values of type int called **`mark`** :



  - The indices of the elements are numbered from 0 to 4;
  - In arrays the first index is always 0, independently of its length.

2.2.1 Array (II)

- The syntax for declaring an array is:

  `datatype array_name[element_number];`

  - Eg: `int mark[5];`
- To initialise the array with specific values, "{}" are used to enclose the elements which are separated by commas ",“
  - Eg: `int mark[5] = {65, 24, 88, 46, 100};`
- To access the elements of the array, bracket and index numbers are used to illustrate them:
  - Eg: `mark[0] = 65; mark[4] = 100;`

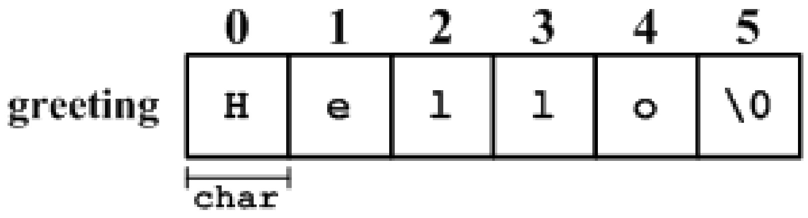
2.2.2 String (I)

- A *string* is a series of characters stored in consecutive bytes of memory.
  - C-style string
  - C++ string object

- C-style string: to treat the character sequence as a **char** type array terminated with a null character (\0). Example:

```
char greeting[5]={‘H’,‘e’,‘l’,‘l’,‘o’};
char greeting[6]={‘H’,‘e’,‘l’,‘l’,‘o’,‘\0’};
```

- The C-style string (**`char`** array) could be represented like this:



2.2.2 String (II)

- The C++ Standard expanded the C++ library by introducing a string class, the second way of dealing with character sequence.
  - To use the string class, a program has to include the **string** header file.
  - The **string** class is part of the **std** namespace, so it is necessary to provide a **using** directive "`using namespace std`" or "`using std::string`"

- To declare and initialise a string object is similar to declare and initialise an ordinary variable. Example:

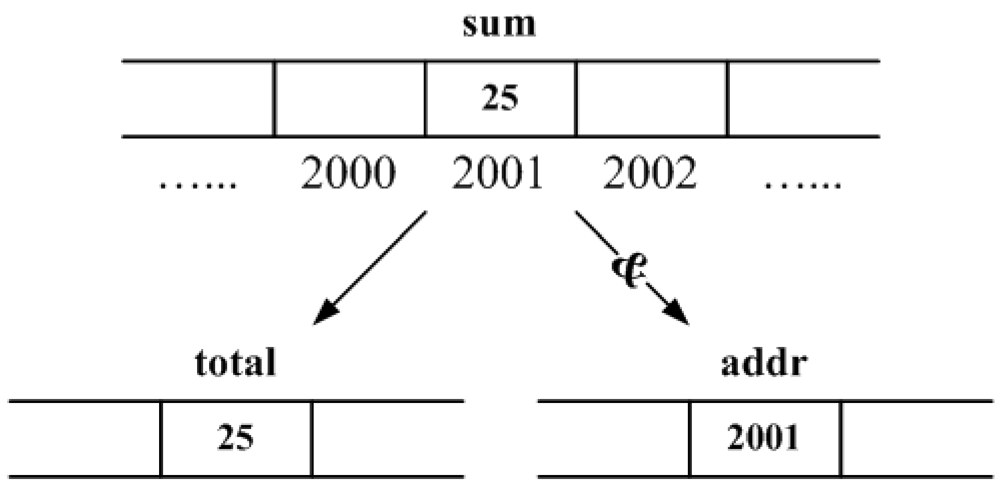
```
string str1;    // declare an empty string object
string str2 = "Hello";// initialise (assignment style)
string str3("World"); // initialise (constructor style)
```

- Check the "reading material 2" for more string operations.

2.2.3 Reference

- The address that locates a variable within memory is what we call a *reference* to that variable.
  - To assign a reference, preceding the identifier of a variable with an ampersand sign (&), known as *reference operator*.
  - Eg: **`addr = &sum;`**

```
sum = 25;
total = sum;
addr = &sum;
```

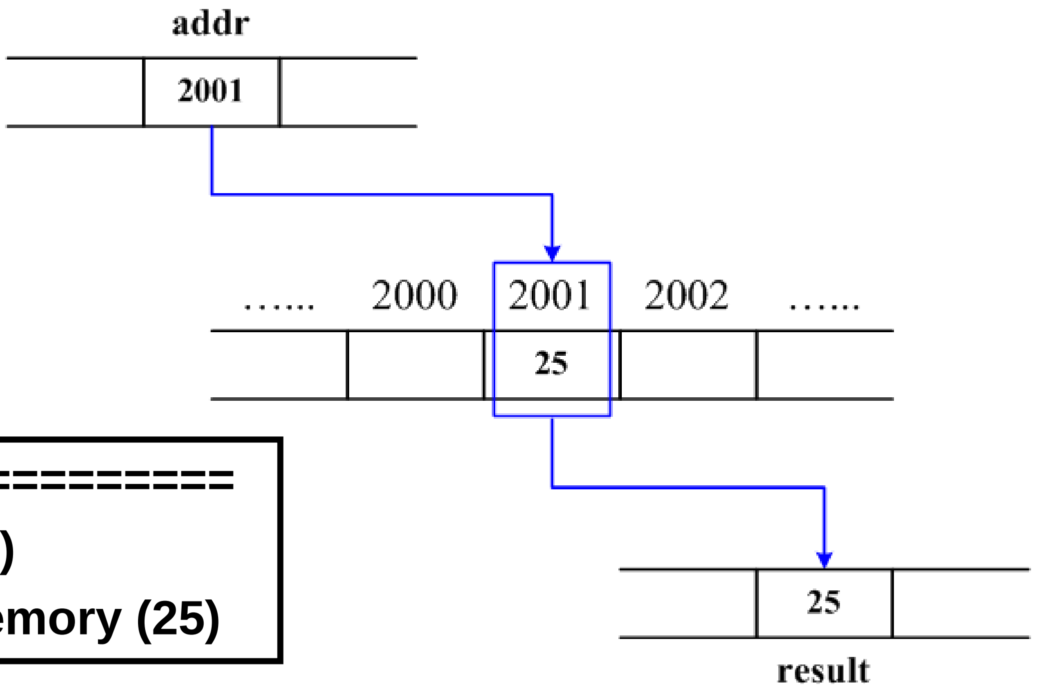


The variable that stores the reference to another variable (like **`addr`** in the example) is what we call a *pointer*.

2.2.4 Pointer (I)

- A variable which stores a reference to another variable is called a *pointer*.
- Dereference operator '**`*`**'
  - Using a dereference operator we can directly access the value stored in the pointer. It can be literally translated to "value pointed by".
  - Eg: **`result=*addr;`** **`% result`** equals to the value pointed by **`addr`**

```
sum =25;

addr = &sum;

result = *addr;
```



=============== Notice ================

addr is the memory address (2001)

***addr is the value stored in the memory (25)**

2.2.4 Pointer (II) - declaration

- It is necessary to specify in its declaration which datatype a pointer is going to point to.
- It is not the same thing to point to a **`char`** as to point to an **`int`** or a **`float`**.
- Syntax:

```
datatype * ptr_name;
```

- Example:

```
int *ptrInt;
char *ptrChar1, *ptrChar2;
float *fNum1, fNum2;
```

2.2.4 Pointer (III) - initialisation

- When declaring a pointer, we can explicitly specify which variable the pointer points to.
- Syntax:

```
    datatype *ptr_name = &var_name;
```

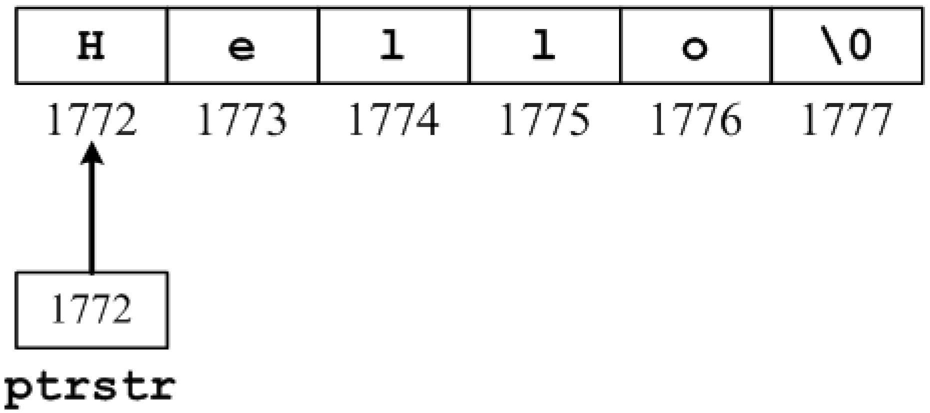
  where the var_name is pre-declared variable with the same datatype.

- Example 1:

```
    int number; int *tommy; tommy = &number;
```

- Example 2:

```
    char *ptrstr = “hello”;
```



2.2.4 Pointer (IV) - manipulation

- Pointer is one of the efficient tools to access elements of an array.
  - For example:

    ```
    int arr[5] = {1,2,3,4,5};
    int *ip = &arr[0];
    ```

  - The pointer **`ip`** can be used to access the elements of the array indirectly

    ```
    cout << *ip;
    *(ip+1) = 20;
    ip++;
    ```

Try “`cout << *ptrstr ;`”

2.3 User-defined Datatypes

-- 2.3.1 Enumeration

- Enumeration in C/C++ language lets the programmer define new types in a fairly restricted fashion.
  - For example:

```
enum shape {circle, square, triangle, ellipse};
enum colour {red, yellow, blue};
```

  - The enumerated tag names (**shape** and **colour**) can be treated as new type. They can be used as normal type to declare variables like:

```
colour background;
background = red;
```

2.3.2 Structure (I)

- ***Structure*** is introduced as a set of data elements grouped together under one name. These data elements, known as members, can have different types and different sizes.
- Declaration syntax:

Declaration of the structure `student`		Using `student` to declare variables	
1_1	`struct student`	2_1	`student henry;`
1_2	`{`	2_2	`student tom, jerry;`
1_3	`    int age;`	2_3	`student *stPtr;`
1_4	`    char name[20];`	2_4	`student Y1[10];`
1_5	`};`		

2.3.2 Structure (II)

- Initialisation of structures:
  - values on the same line

```
student henry = {20, “henry”};
```

- Access the members of a structure
  - Use a dot (.) inserted between the structure variables name and the member name. For example:

```
cout <<“Name: ” <<henry.name <<“Age: ” <<henry.age <<endl;
```

  - Access through a pointer which points to the structure variable. For example:

```
student *stPtr;
stPtr = &henry;
cout <<“Name: ” <<stPtr->name <<“Age: ” << stPtr->age <<endl;
```

2.4 Datatype casting

- In C and C++, every data has its own data type which cannot be freely changed. Converting a variable of a given type into another type is known as type-casting.
- ***Implicit Conversions: No Operator.***
  - They are automatically performed when a value is copied to a compatible type.

```
double c = 3.9;
int d = c;
```

- ***Explicit Conversions: ( ) – Cast Notation.***
  - C++ is a strong-typed language. Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion.

```
double c = 3.9;
int d = (int)c;
```

3. Basic Input and Output (I)

- Standard output (**`cout`**)
  - By default, the standard output of a program is the screen, and the C++ stream object defined to access it is **`cout`**.
  - **`cout`** is used in conjunction with the insertion operator, which is written as **`<<`** (two "less than" signs).
  - Example:

```
cout << "Output sentence"; // prints content in “” on screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints content of x on screen
```

  - **`cout`** does not add a line break after its output

```
cout << "Hello”;
cout << "Hello\n”; or cout << “Hello” << endl;
```

  - The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, I am " << age << " years old." << endl;
cout << “End of the line \n”;
```

3. Basic Input and Output (II)

- Standard input (**`cin`**)
  - The standard input device is usually the keyboard.
  - Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream.
  - The operator must be followed by the variable that will store the data that is going to be extracted from the stream.
  - Example:

```
int age;
cin >> age;
```

  - **`cin`** can be used to request more than one datum input from keyboard:

```
cin >>a >>b;
```

  - **`cin`** can also be used to get strings:

```
cin >>mySurname;
getline (cin, myFullName);
```

3. Operators and Expressions

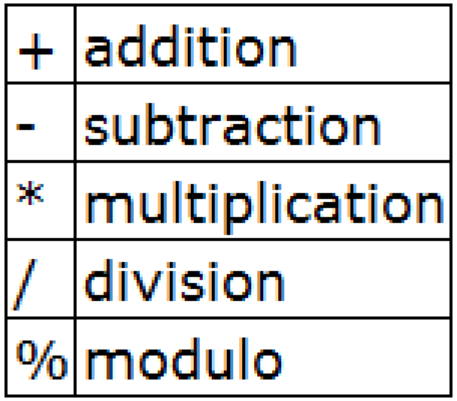
-- 3. 1 Operators (I)

- 1. Assignment (**=**): assigns a value to a variable.
  - Example:

```
a=5;
b=a;
```

- 2. Arithmetic operators (`+, -, *, /, %`).
  - Example:

```
sum = a+b;
average = sum / 5;
remainder = 11 % 3;
```



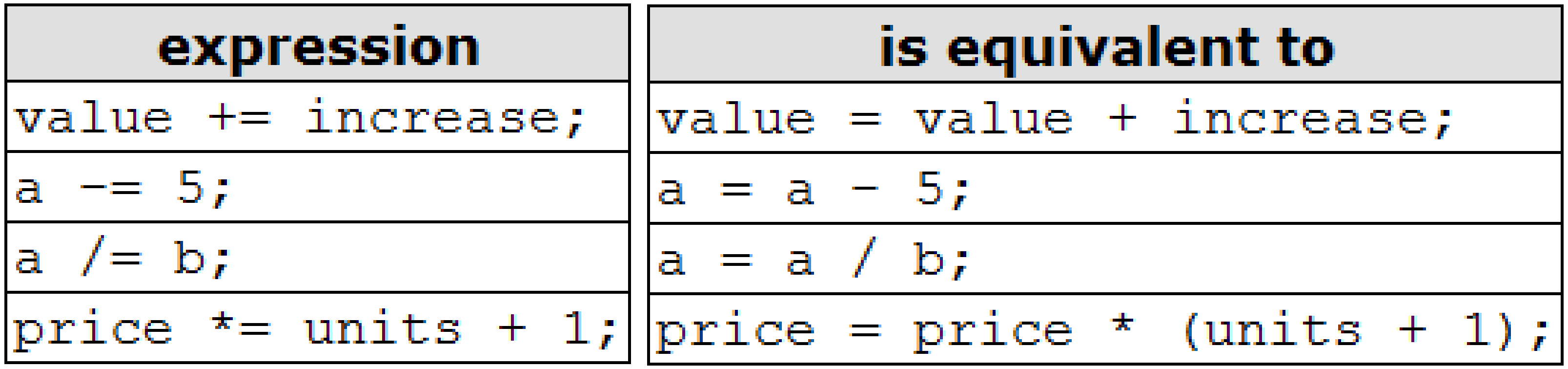
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

3.1 Operators (II)

- 3. Compound assignment

  `(+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)`

  – To modify the value of a variable by performing an operation on the value currently stored in that variable



expression	is equivalent to
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

3.1 Operators (III)

- 4. Increase or decrease **`(++, --)`**
  - the increase (++) and the decrease (--) means increase or reduce by one

    ```
    c++;
    c+=1;
    c=c+1;
    ```

  - **`++`** and **`--`** can be used both as a prefix and as a suffix
    - as a prefix **`(++a)`** the value is increased before the result of the expression is evaluated
    - as a suffix **`(a++)`** the value is increased after the result of the expression is evaluated

```
B=3;
A=B++;
// Results: A=3, B=4
```

```
B=3;
A=++B;
// Results: A=4, B=4
```

3.2 Expressions

- Relational and equality operators `(==, !=, >, <, >=, <= )`
  - To evaluate a comparison between two expressions
  - The result of a relational operation is a Boolean value (true or false)

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

!	NOT
&&	AND
||	OR

- Logical operators (`!, &&, ||`)
  - Examples:

```
!(5 == 5)                  // evaluates to false
((5 == 5) && (3 > 6))      // evaluates to false
((5 == 5) || (3 > 6))      // evaluates to true
```

3.3 Conditional operator

- The conditional operator is the only three-operand operator in C++.
- It evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false.

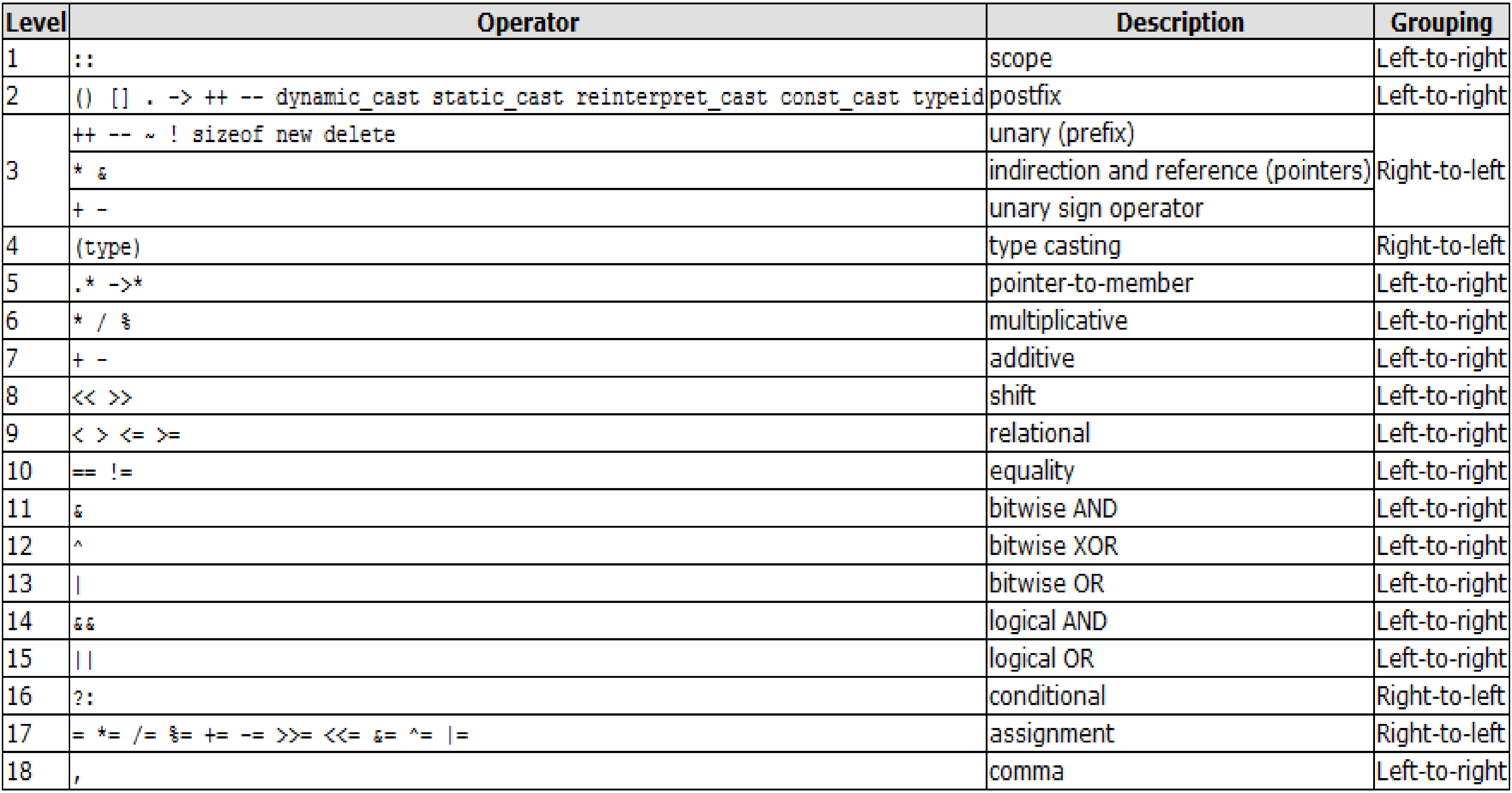
- Syntax:

```
condition ? result1 : result2
```

- If condition is true the expression will return result1, if it is not it will return result2.
  - For example:

```
7==5 ? 4 : 3 // return 3, since 7 is not equal to 5.
a>b ? a : b  // return whichever is greater, a or b.
```

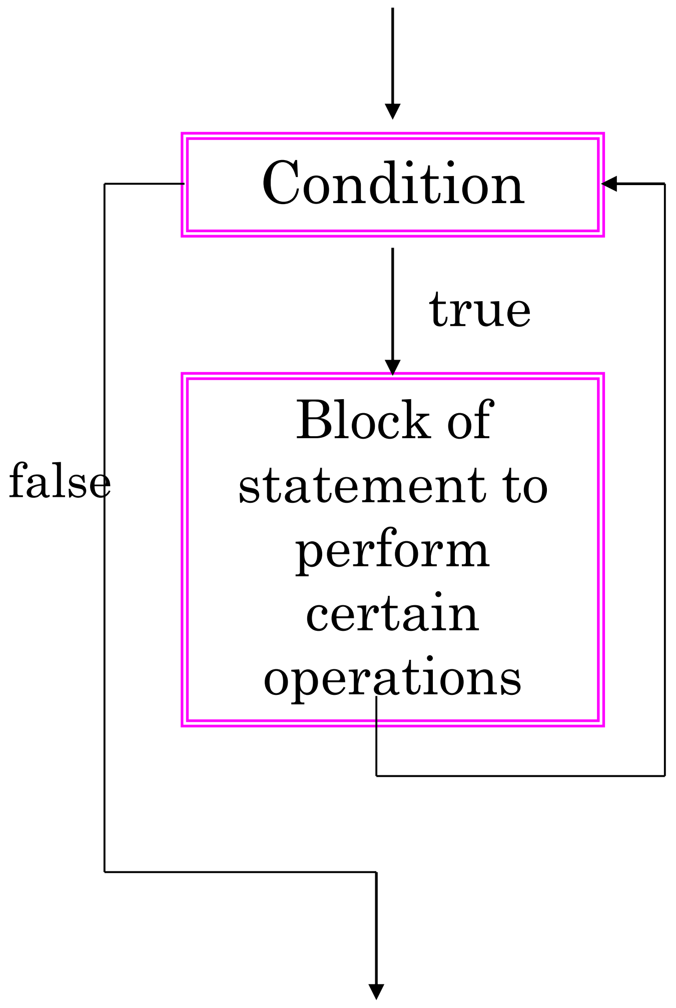
3.4 Precedence of operators



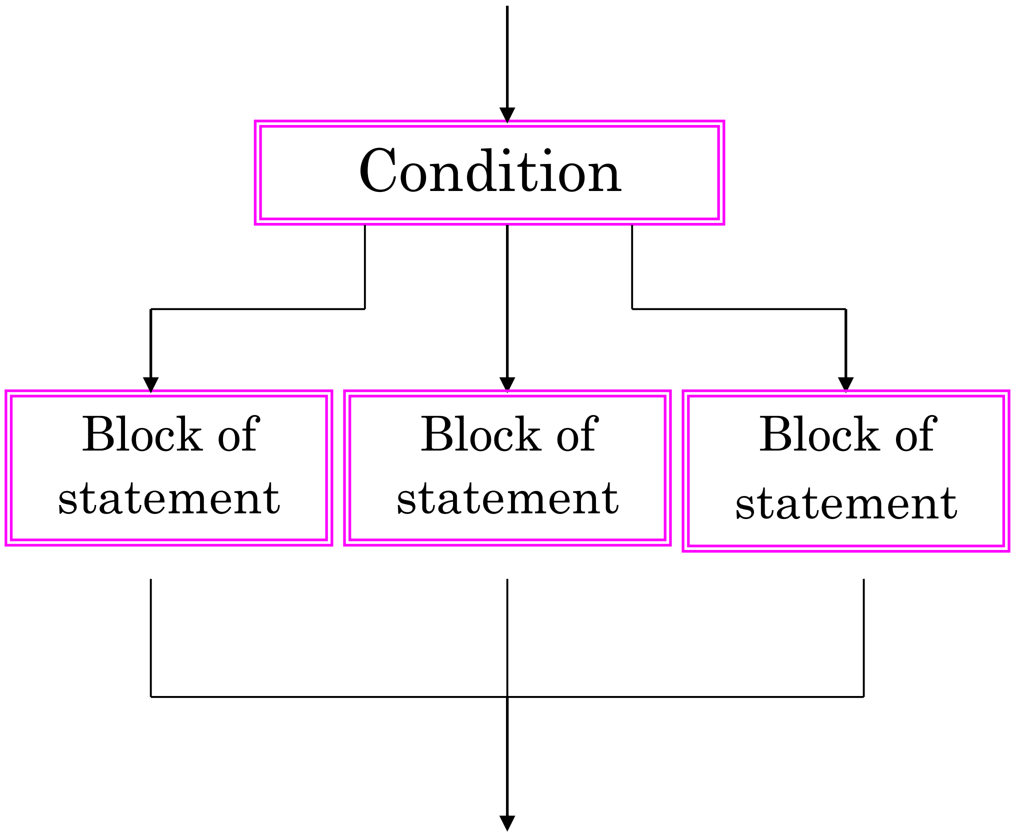
Level	Operator	Description	Grouping
1	`::`	scope	Left-to-right
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	postfix	Left-to-right
3	`++ -- ~ ! sizeof new delete`	unary (prefix)	Right-to-left
	`* &`	indirection and reference (pointers)	
	`+ -`	unary sign operator	
4	`(type)`	type casting	Right-to-left
5	`.* ->*`	pointer-to-member	Left-to-right
6	`* / %`	multiplicative	Left-to-right
7	`+ -`	additive	Left-to-right
8	`<< >>`	shift	Left-to-right
9	`< > <= >=`	relational	Left-to-right
10	`== !=`	equality	Left-to-right
11	`&`	bitwise AND	Left-to-right
12	`^`	bitwise XOR	Left-to-right
13	`|`	bitwise OR	Left-to-right
14	`&&`	logical AND	Left-to-right
15	`||`	logical OR	Left-to-right
16	`?:`	conditional	Right-to-left
17	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment	Right-to-left
18	`,`	comma	Left-to-right

5. Logical structures

Repetition:



Selection (branching):



5. Logical structures

- Selection
  - if… else…
  - switch…case…

- Repetition
  - for loop
  - while loop
  - do…while…

  - break, continue