CS2311 Computer Programming

LT3: Basic Syntax

Part II: Operators & Basic I/O

Operators and Punctuators

- An operator specifies an operation to be performed on some values
  - These values/variables are called the operands of the operator
- Some examples: +, -, *, /, %, ++, --,>>,<<
- Some of these have meanings that depend on the context

Expressions

- An **expression** is a combination of constants, variables, and function calls that evaluate to a result

- Examples:

constants

```
x = 3.0*4.0;
```

variables

```
y = 2.0 + x;
```

function call

```
z = 5.0 + x/y - sqrt(x*3.0);
```

Assignment operator =

- Generic form

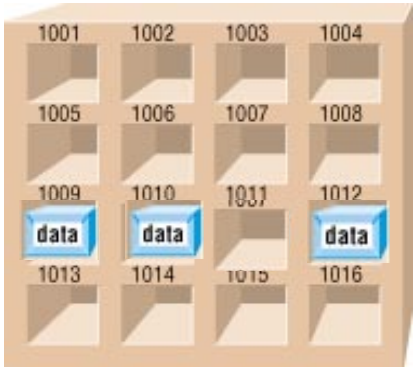
  variable = expression;

  ```
  char  x = 'a';
  ```

- Variable

Type: char Name: X Address: 1009
97

Type: char Name: Y Address: 1010
98



Assignment operator =

- Generic form

  variable = expression;

- = is an assignment operator that is different from the *mathematical equality* (which is == in C++)

- An expression itself has a value, e.g.,

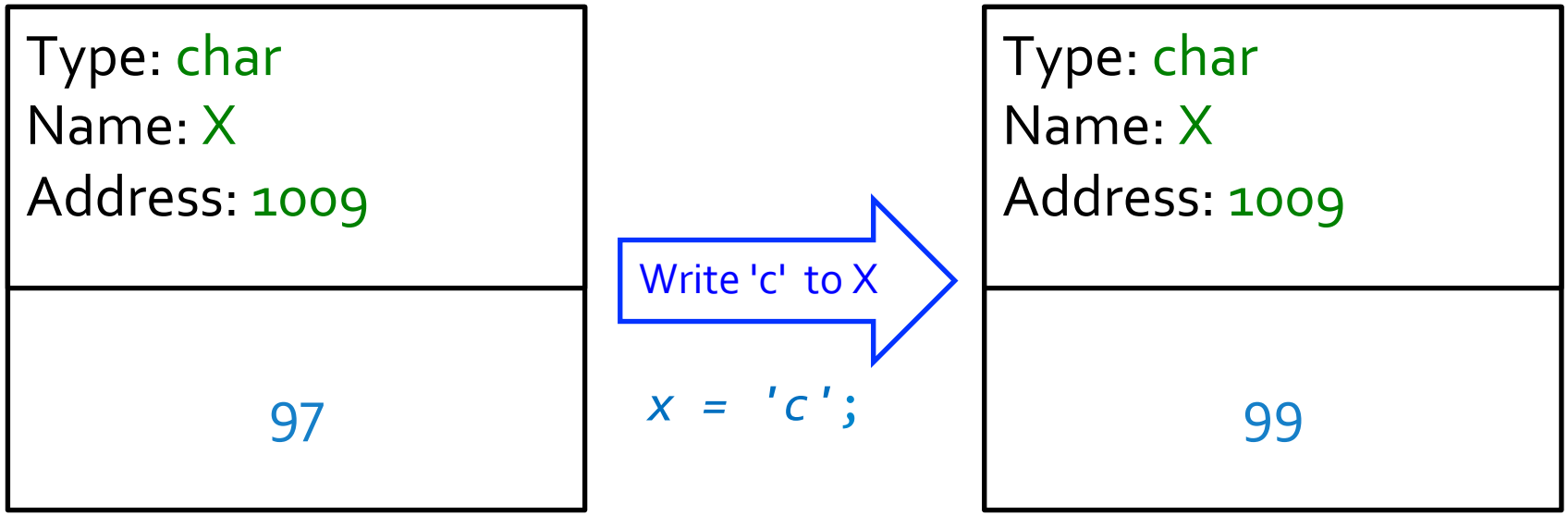
  a = (b = 2) + (c = 3);

  - An *assignment statement* has a value equal to the operand
  - In the example, the value of *assignment statement* "b=2" is 2 and "c = 3" is 3
  - Therefore, "a = ..." is 5 → *a = 2 + 3;*

Assignment operator =

- Write-to a variable

- After the write, the previous stored value in the variable no longer exists, and is replaced by the new value



Examples of Assignment Statements

Error!!

```
// Invalid: left hand side must be a variable
a + 10 = b;

// assignment to constant is not allowed
2=c;

// valid but not easy to understand
int a, b, c;
a = (b = 2) + (c = 3);

// avoid complex expressions
int a, b, c;
b = 2;
c = 3;
a = b + c;
```

Increment & Decrement Operators

- Increment and decrement operators: **++** and **--**
  - `k++` and `++k` are equivalent to `k = k+1`, or `k += 1`
  - `k--` and `--k` are equivalent to `k = k-1`, or `k -= 1`

- Post-increment and post-decrement: `k++` and `k--`
  - k's value is altered **AFTER** the expression is evaluated

```
int k = 0, j;
j = k++;  // result: j = k, k = k+1
```

- Pre-increment and pre-decrement: `++k` and `--k`
  - k's value is altered **BEFORE** evaluating the evaluation

```
int k = 0, j;
j = ++k;  // result: k = k+1, j = k
```

Post-increment and **post**-decrement: k++ and k--

- k's value is altered AFTER the expression is evaluated

```
int k = 1, j;
j = k++;          // result: j is 1, k is 2
```

Pre-increment and **pre**-decrement: ++k and --k

- k's value is altered BEFORE evaluating the expression

```
int k = 1, j;
j = ++k;          // result: j is 2, k is 2
```

```
k=0;

i=1+(k++);
```

0

Use original value of k

```
i = 1+0
  = 1
```

```
k=0;

i=1+(++k);
```

1

Use updated value of k

```
i = 1+1
  = 2
```

Value of k will be 1 in both cases

What values are printed?

```
int k=0,i=0;
cout << "i= " << i << endl;

k=0;
i=1+(k++);
cout << "i= " << i << endl;
cout << "k= " << k << endl;


k=0;
i=1+(++k);
cout << "i= " << i << endl;
cout << "k= " << k << endl;
```

Output

```
i= 0
i= 1
k= 1
i= 2
k= 1
```

Precedence & Associativity of Operators

- An expression may have more than one operator and its precise meaning depends on the precedence and associativity of the involved operators
- What are the values of variables `a`, `b` and `c` after the execution of the following statements

```
int a, b = 2, c = 1;
a = b+++c;
```

- Which of the following interpretation is right?

```
a = (b++) + c; // right
```

or

```
a = b + (++c); // wrong
```

Precedence & associativity of operators

- **Precedence**: order of evaluation for different operators.
  - Precedence determines how an expression like x **R** y **S** z should be evaluated (now **R** and **S** are ***different*** operators, e.g., x **+** y **/** z).

- **Associativity**: order of evaluation for operators with the same precedence.
  - Associativity means whether an expression like x **R** y **R** z (where R is a operator, e.g., x **+** y **+** z) should be evaluated '**left-to-right**' i.e. as **(**x **R** y**)** **R** z or
  - '**right-to-left**' i.e. as x **R** **(**y **R** z**)**;

Precedence & Associativity of Operators

Operator Precedence (high to low)						Associativity
`::`						None
`.`	`->`	`[]`				Left to right
`()`	`++(postfix)`	`--(postfix)`				Left to right
`+(unary)`	`-(unary)`	`++(prefix)`	`--(prefix)`			Right to left
`*`	`/`	`%`				Left to right
`+`	`-`					Left to right
`=`	`+=`	`-=`	`*=`	`/=`	`etc.`	Right to left

Example:
```
a = b+++c
a = (b++)+c; or
a = b+(++c);
```

Example:
```
int a, b=1;
a = b = 3+1;
```

→

```
b = 3+1;
```

↓

```
a = b;
```

Swapping the Values

- We want to swap the content of two variables.
- What's wrong with the following program?

```
int main() {
    int a = 3, b = 4;
    a = b;
    b = a;
    return 0;
}
```

a=~~3~~ 4

b=~~4~~ 3

c=3

Swapping the Values

- We want to swap the content of two variables.
- What's wrong with the following program? [demo]

```
int main() {
    int a = 3, b = 4;
    a = b;
    b = a;
    return 0;
}
```

- We need to make use of a temporary variable

```
c = b; // save the old value of b
b = a; // put the value of a into b
a = c; // put the old value of b to a
```

Efficient/Compound Assignment

- The generic form of *efficient* assignment operators:

  variable op = expression; where op is an operator

  The meaning is

  variable = variable op (expression);

- Efficient assignment operators include

  `+=` `-=` `*=` `/=` `%=` (arithmetic operators)

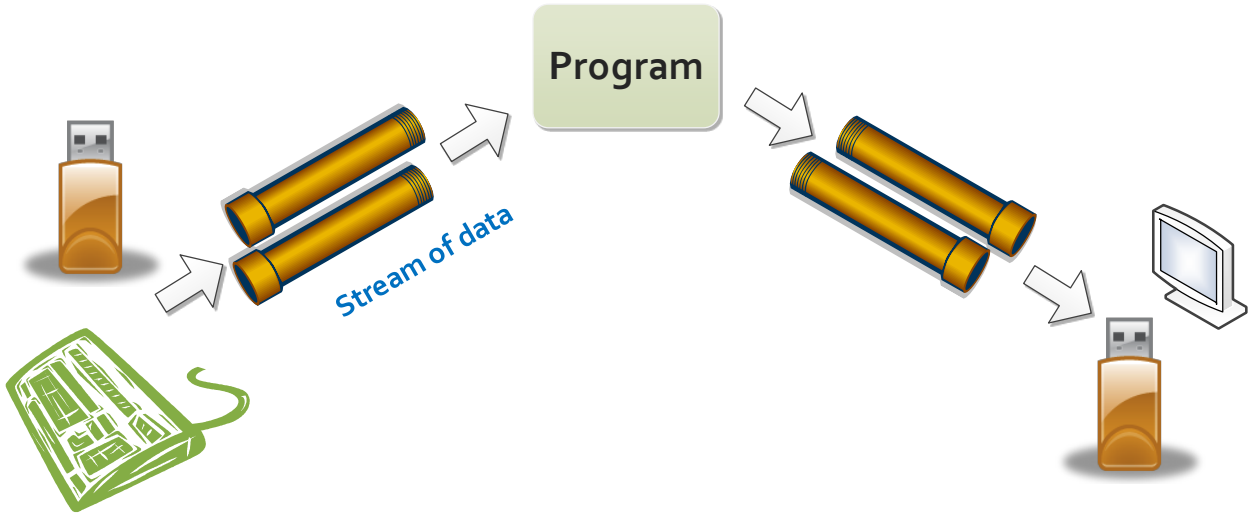
- Examples:

```
a += 5;
a -= 5;
a += b*c;
a *= b + c;
```

- Also known as **compound assignment** operators

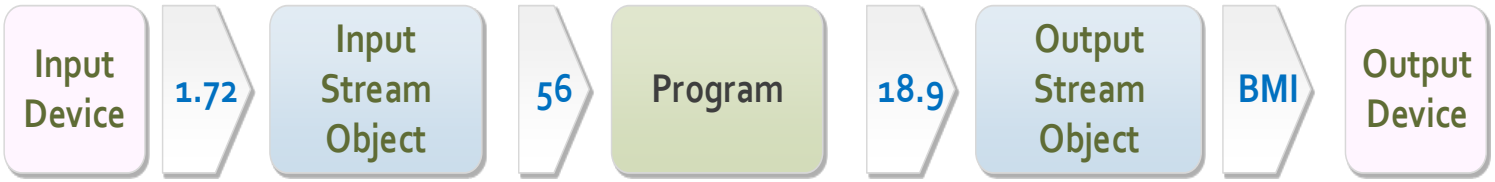
Basic I/O – Keyboard and Screen

- A program can do little if it can't take input and produce output
- Most programs read user input from keyboard and secondary storage
- After processing the input data, result is commonly displayed on screen or write to storage (disk)



Basic I/O – `cin` and `cout`

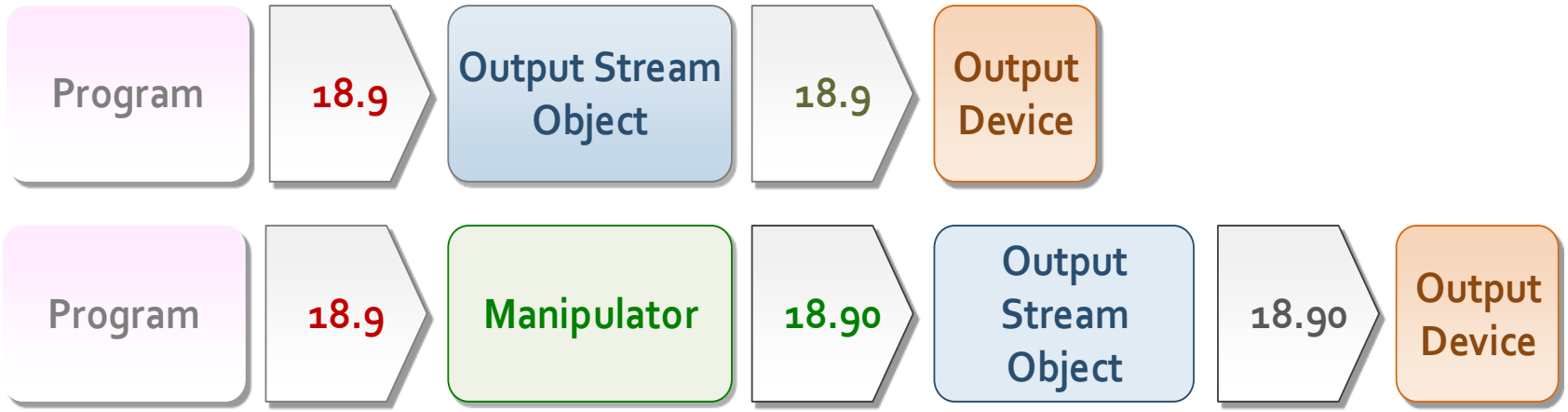
- C++ comes with an `iostream` package (library) for basic I/O.
- `cin` and `cout` are objects defined in `iostream` for keyboard input and screen display respectively
- To read data from `cin` and write data to `cout`, we need to use extraction/input operator (`>>`) and insertion/output operator (`<<`)




```
cin >> weight;
cin >> height;
cout << "bmi";
cout << weight/height/height;
```

cout: Output Operator <<

- Preprogrammed for **all** standard C++ **data types**
- It sends **bytes** to an output stream object, e.g. **cout**
- Predefined "*manipulators*" can be used to change the default *format* of arguments



cout: Output Operator <<

Type	Expression	Output
Integer	`cout << 21`	`21`
Float	`cout << 14.5`	`14.5`
Character	`cout << 'a';` `cout << 'H' << 'i'`	`a` `Hi`
Bool	`cout << true` `cout << false`	`1` `0`
String	`cout << "hello"`	`hello`
New line (endl)	`cout << 'a' << endl << 'b';`	`a` `b`
Tab	`cout << 'a' << '\t' << 'b';`	`a  b`
Special characters	`cout << '\"' << "Hello" << '\"' << endl;`	`"Hello"`
Expression	`int x = 1;` `cout << 3 + 4 + x;`	`8`

cout – Change the width of output

- Change the width of output
  - Calling member function width(*width*) or using `setw` manipulator
    - requires iomanip library: `#include <iomanip>` for `setw`
  - Leading blanks are added to any value fewer than width
  - If formatted output exceeds the width, the entire value is printed
  - Effect last for one field only

Approach	Example	Output (◆ for space)

`cout.width(width)`	`cout.width(5);`
	`cout << 56 << endl;`
	`cout.width(6);`
	`cout << 5768 << endl;`
		◆◆◆56
		◆◆5768

`setw(width)`	`cout << setw(5) << 18;`
	`cout << setw(5) << 123 << endl;`
	`cout << setw(5) << 1234567 << endl;`
		◆◆◆18◆◆123
		1234567

`cout` – Set the Precision and Format of Floating Point Output

- Must `#include <iomanip>`
- Floating-point precision is six by default, i.e. 6 digits in total
- Use `setprecision`, fixed and scientific manipulators to change the *precision value* and *printing format*
- Effect is **permanent**

Default behavior

Example	Output
`cout << 1.34 << endl;`	`1.34`
`cout << 1.340 << endl;`	`1.34`
`cout << 1.3401234 << endl;`	`1.34012`
`cout << 0.0000000134 << endl;`	`1.34e-008`

fixed and scientific Manipulators

- `cout << fixed`: always uses the fixed point notation
  - 6 significant digits after decimal point
- `cout << scientific`: always uses the scientific notation
- They change the meaning of precision (see the example)

Example	Output
`cout << fixed;`	
`cout << 1.34 << endl;`	`1.340000`
`cout << 1.340 << endl;`	`1.340000`
`cout << 0.0000000134 << endl;`	`0.000000`

`cout << scientific;`	
`cout << 1.34 << endl;`	`1.340000e+00`
`cout << 1.340 << endl;`	`1.340000e+00`
`cout << 0.000000012345671 << endl;`	`1.234567e-08`
`cout << 0.000000012345675 << endl;`	`1.234568e-08`

cout setprecision

- Normally, `setprecision(n)` means output `n` significant digits in total
- But with "`fixed`" or "`scientific`", `setprecision(n)` means output `n` significant digits **after** the decimal point

Example	Output
`cout << setprecision(2);`	`1.3`
`cout << 1.34 <<endl;`	`1.3e-08`
`cout << 0.0000000134 << endl;`	`0.00`
`cout << fixed;`	`5.00e-04`
`cout << 0.0000000134 << endl;`	
`cout << scientific << 0.0005 << endl;`	

cout – Other Manipulators

Manipulators	Example	Output

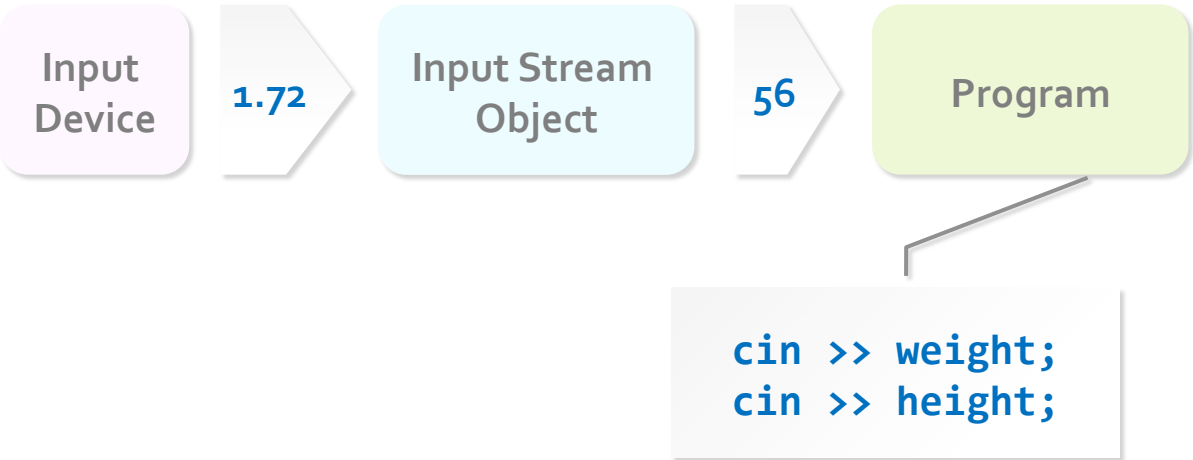
fill	`cout << setfill('*');`
	`cout << setw(10);`
	`cout << 5.6  << endl;`
	`cout << setw(10);`
	`cout << 57.68 << endl;`
		*******5.6
		*****57.68

radix	`cout << oct << 11 << endl;// octal`
	`cout << hex << 11 << endl;// hexadecimal`
	`cout << dec << 11 << endl;`
		13
		b
		11

alignment	`cout << setiosflags(ios::left);`
	`cout << setw(10);`
	`cout << 5.6 << endl;`
		5.6

`cin`: Extraction Operators (>>)

- Preprogrammed for **all** standard C++ data types
- Get bytes from an input stream object
- Depend on ***white space*** to separate incoming data values



Input Operator >>

Type	Variable	Expression	Input	x	y
Integer	int x, y;	`cin >> x;`	21	21	
		`cin >> x >> y;`	5 3	5	3
Float	float x, y;	`cin >> x;`	14.5	14.5	
Character	char x, y;	`cin >> x;`	a	a	
		`cin >> x >> y;`	Hi	H	i
String	char x[20]; char y[20];	`cin >> x;`	hello	hello	
		`cin >> x >> y`	Hello World	Hello	World

Programming Styles

- Programmers should write code that is understandable to other people as well
- **Meaningful** variable names / literals
- Which is more meaningful

```
tax = temp1 * temp2;    // not meaningful
tax = price * tax_rate; // good
```

- **Meaningful** Comments
  - Write comments as you write the program
- Indentation

Indentation Styles

```
int main()
{
    int x, y;
    x = y++;
    return 0;
}
```

```
int main() {
    int x, y;
    x = y++;
    return 0;
}
```

Both are good. Choose one and stick with it.

```
int main()
{
int x, y;
 x= y++;
return 0;}
```

BAD!! Avoid this!!

Use of Comments

- Top of the program
  - Include information such as the name of organization, programmer's name, date and purpose of program
- What is achieved by the function, the meaning of the arguments and the return value of the function
- Short comments should occur to the right of the statements when the effect of the statement is not obvious and you want to illuminate what the program is doing
- Which one of the following is more meaningful?

```
tax = price * rate;  // sales tax formula
tax = price * rate;  // multiply price by rate
```

Summary

- Basic Operators
  - Assignment
  - Arithmetic
  - Compound : Increment & Decrement
  - Expression
  - Operator precedence
- I/O Operators
  - `cin >>`
  - `cout <<`
    - Format
    - Precision
- Comments
- Programming Style