CS2310 Computer Programming

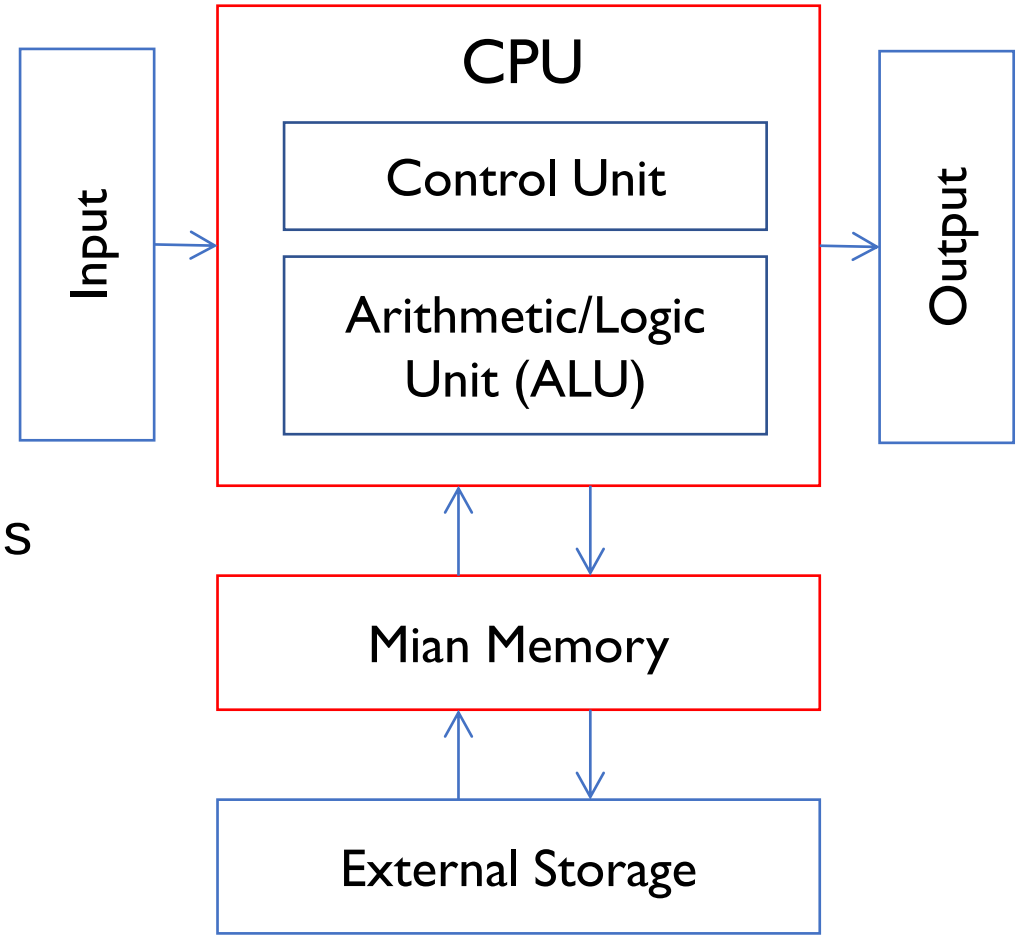
LT02: Data, Operators, and BasicIO

Computer Science, City University of Hong Kong

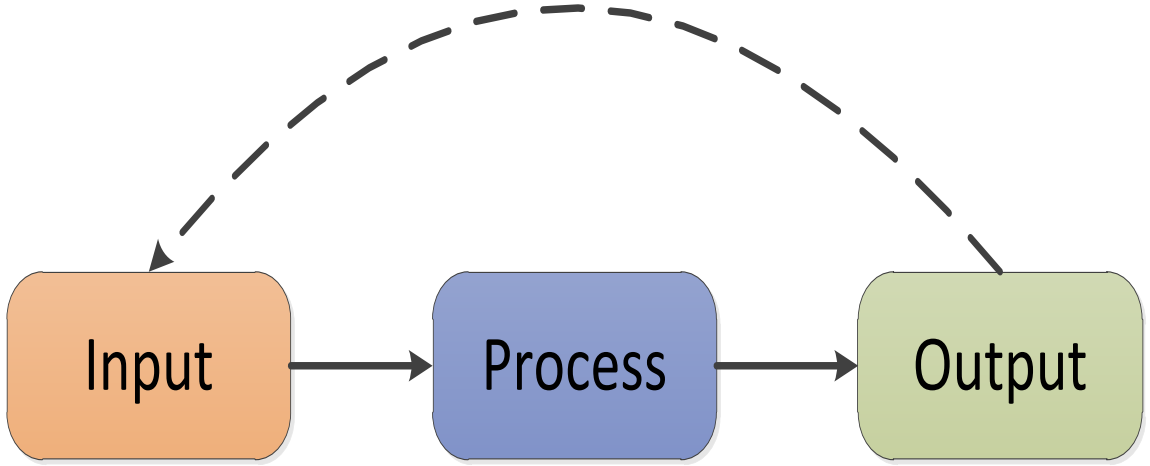
Semester A 2023-24

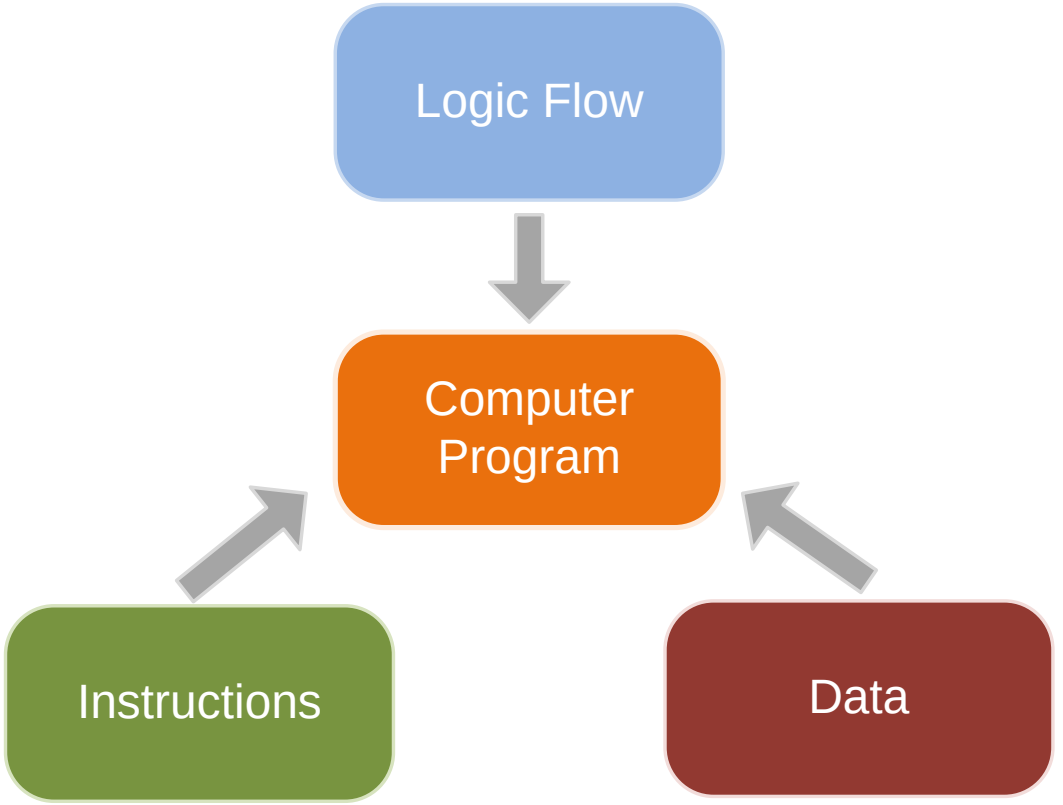
Quick Review: What's a Computer

- von Neumann machine (stored program computer)
- *Main Memory*: stores both data and program, i.e., a list of instructions
- *CPU (Central Processing Unit)*:
  - *ALU*: performs arithmetic and bitwise operations
  - *Control Unit*: read instructions from memory, direct ALU to execute instructions
- *External storage:* (slow) mass storage
- *Input/output:* keyboard, display …

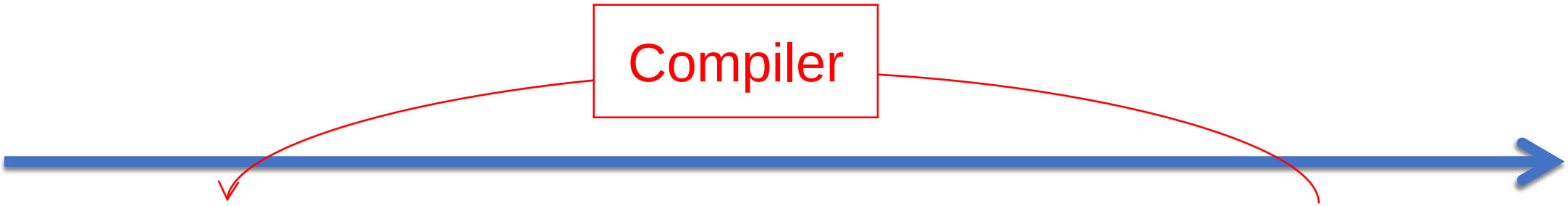


Quick Review: What's a Computer Program





Quick Review: Programming Languages



Machine Language

Language directly understood by the computer

Defined by ISA

x86, RISC …

Symbolic Language

English-like *abbreviations* representing elementary computer operations

Assembly language

High-level Language

Close to human language.

Example: $a = a + b$

[add vales of *a* and *b*, and store the result in *a*, replacing the previous value]

C/C++, Java, Python

Quick Review: Basic Syntax and Program

```
/* What's wrong with the following program? */


using namespace std;
int main()
{
        cout < Hello world! < endl
        return 0;
}
```

Today's Outline

- C++ basic syntax
- Variable and constant
- Operators
- Basic I/O

A Simple C++ Program

```
#include <iostream>
using namespace std;
int main() {
    float r, area;                          // the radius and area of the circle
    cout << “Input circle radius ”;         // print prompt on screen
    cin >> r;                               // let user input r from keyboard
    area = 3.1415926 * r * r;               // calculate circle area
    cout << “Area is ” << area << endl;     // print result on screen
    return 0;
}
```

Syntax of C++

- Like any language, C++ has an alphabet and rules for putting together words and punctuations to make a legal program.

  This is called *syntax* of the language.

- C++ compilers detect any violation of the syntax rules in a program

- C++ compiler collects the characters of the program into *tokens*, which form the basic vocabulary of the language

- Tokens are separated by space

Tokens in C++

- Tokens in C++ can be categorized into:

Keywords

Identifiers

Operators

String constants

Numeric constants

Punctuators

Tokens in C++: An Example

```
#include <iostream>
using namespace std ;
int main ( ) {
    float r , area ;
    cout << “input circle radius ” ;
    cin >> r ;
    area = 3.1415926 * r * r ;
    cout << “area is ” << area << endl ;
    return 0 ;
}
```

preprocessor

keywords

identifiers

operators

string constants

numeric constants

punctuators

Keywords

- Words reserved by the programming language
- Each keyword in C++ has a reserved meaning and cannot be used for other purpose

```
#include <iostream>
using namespace std;
int main() {
        float r, area;
        cout << “input circle radius ”;
        cin >> r;
        area = 3.1415926 * r * r;
        cout << “area is ” << area << endl;
        return 0;
}
```

Keywords (cont'd)

Data type	char	double	float	int	bool
	long	short	signed	unsigned	void

Keywords (cont'd)

Data type	char	double	float	int	bool
	long	short	signed	unsigned	void
Flow control	if	else	switch	case	while
	break	default	for	do	continue

Keywords (cont'd)

Data type	char	double	float	int	bool
	long	short	signed	unsigned	void
Flow control	if	else	switch	case	while
	break	default	for	do	continue
Others	using	namespace	true	false	sizeof
	return	const	class	new	delete
	operator	public	protected	private	friend
	this	try	catch	throw	struct
	typedef	enum	union		

Identifiers

- Identifiers give unique names to various objects in a program like the name of variables, functions, libraries, and namespace
- Keywords cannot be used as identifiers

```
#include <iostream>
using namespace std;
int main() {
        float r, area;
        cout << “input circle radius ”;
        cin >> r;
        area = 3.1415926 * r * r;
        cout << “area is ” << area << endl;
        return 0;
}
```

Identifiers (cont'd)

- An identifier is composed of a sequence of letters, digits and underscore
  - E.g., myRecord, point3D, last_file
- An identifier must begin with either an underscore or a letter
  - Valid identifiers: _income, _today_record, record1
  - Invalid identifiers: 3D_point, 2ppl_login, -right-
  - Identifier is ***case sensitive***
- Always use meaningful names for identifiers

Identifiers (cont'd)

```
float a (float b, float c, float d)
{
        return (b + c) * d / 2;
}
```

Identifiers (cont'd)

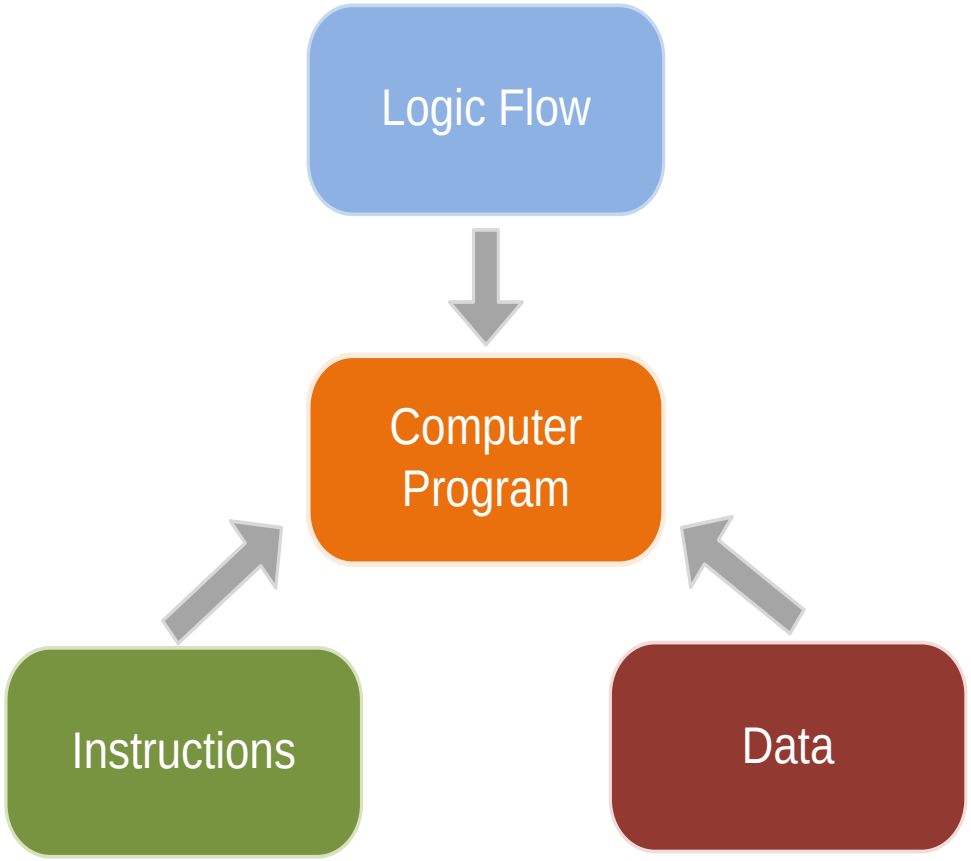
```
float a (float b, float c, float d)
{
        return (b + c) * d / 2;
}

float trapezoid_area (float upper_edge, float lower_edge, float height)
{
        return (upper_edge + lower_edge) * height / 2;
}
```

Today's Outline

- C++ language syntax
- Variable and constant
- Operators
- Basic I/O

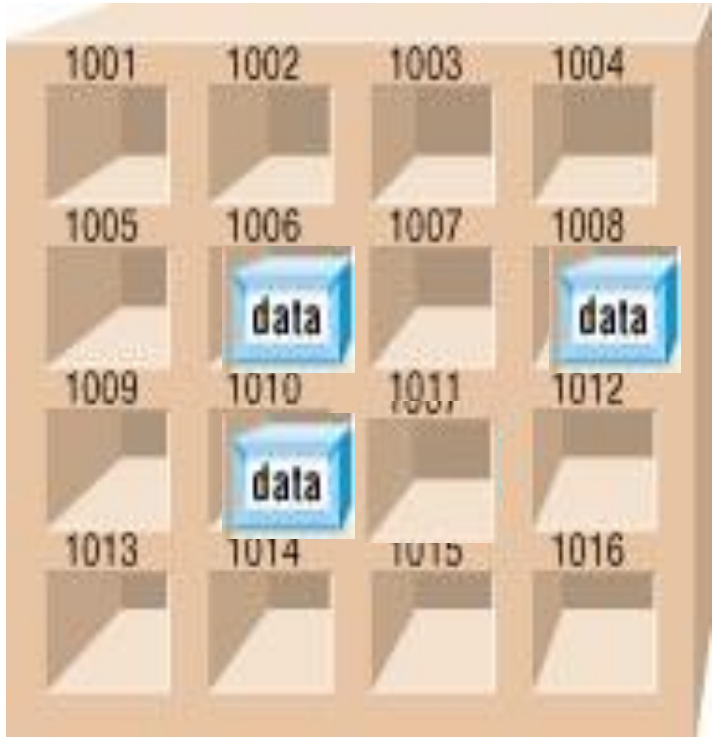
Variable and Constant

- Computer programs typically involve data access
- Two categories of data in C++ program
- *Variable*: memory storage whose value can be changed during program execution
- *Constant*: memory storage whose value does NOT change during program execution



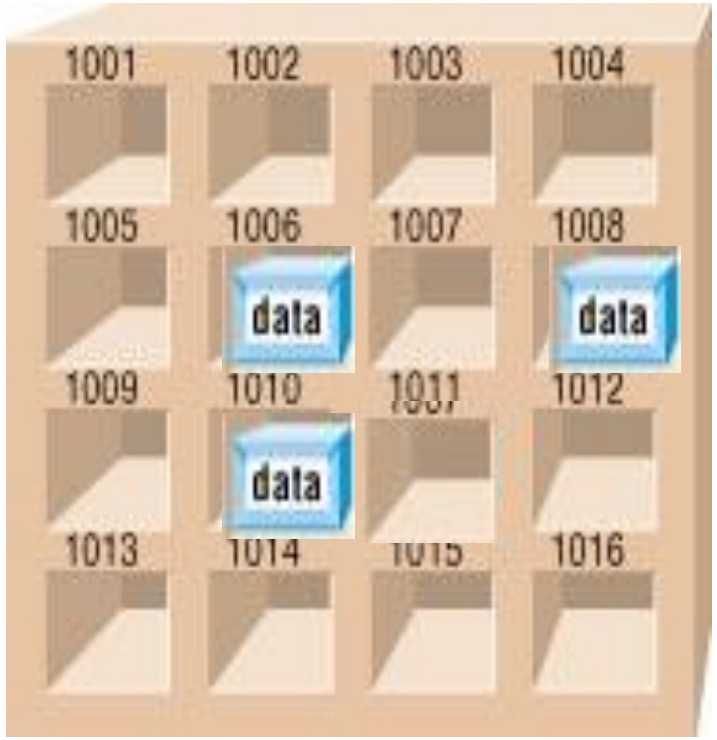
Variable and Constant

- Every variable/constant has ***5*** attributes
- *Address*: location of data in memory storage
- *Value*: content in memory storage

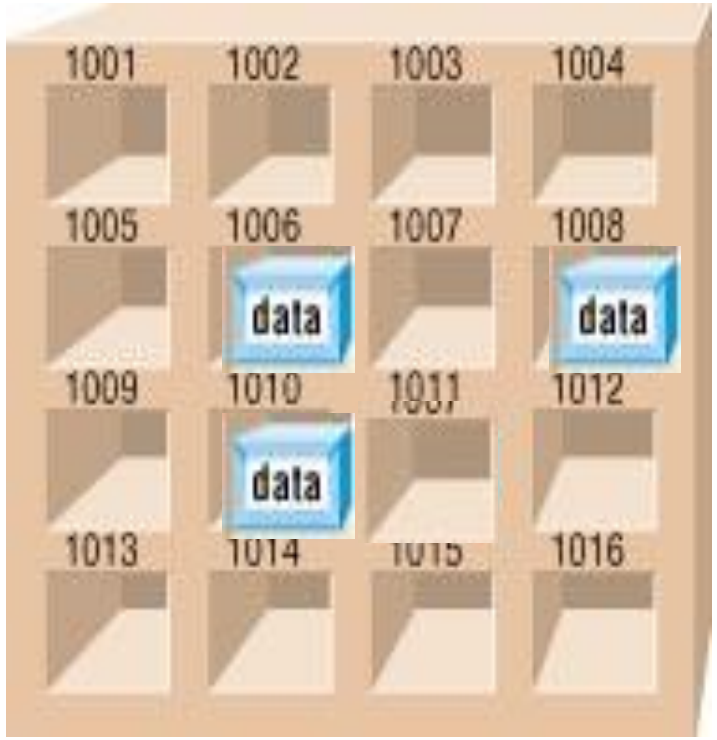


Variable and Constant

- Every variable/constant has ***5*** attributes
- *Address*: location of data in memory storage
- *Value*: content in memory storage
- *Name*: identifier of the variable



Variable and Constant



- Every variable/constant has ***5*** attributes
- *Address*: location of data in memory storage
- *Value*: content in memory storage
- *Name*: identifier of the variable
- *Type*: C++ is a strictly typed language, variables and constants must belong to a data type
  - E.g., numerical, character, logic, other…
- *Scope*: it defines the region within a program where the variable/constant can be accessed, and also the *conflict domain*
  - more detail soon

Variable Naming

- Hard rules
  - Variable names are composed of the characters:

    a,b,c,..,z,A,B,C,…,Z,0,1,2,…,9 and _
  - Variables names must begin with:

    a,b,c,..,z,A,B,C,…,Z or _
- Naming conventions
  - CamelCase: Resembles the humps of a camel,
    - e.g., myVariableName
  - snake_case: Mimics the form of a snake with underscores,
    - e.g., my_variable_name

Variable/Constant Name

- Which of the following are valid variable/constant names?

[A] you

[B] CityU_CS

[C] 2U

[D] $cake

[E] \you

[F] CityU-CS

Variable Declaration

- Variable and constants must be declared before use
- Variable declaration format

  data_type variable_identifier ;

- *Optionally*, you can set the initial value of variable during declaration
- Examples

  int age ;

  float bathroom_temperature = 28, bedroom_temperature = 30 ;

  char initial ;

  char student_name[20] ;

Constant Declaration

- *Variable and constants must be declared before use*
- Constant declaration format

  const data_type variable_identifier = value ;

- You *MUST* assign an initial value to a constant during declaration
- The value of a const *CANNOT* be changed after declaration
- Examples

```
const int days_per_year = 365;
days_per_year = 366; // error
const int hrs_per_day; // error
```

C++ Predefined Data Types

- Numerical
  - int, short, long: integer number
  - float, double: real number
- Character
  - char: ASCII character (a, e, o, \n)
- Logic (next lecture)
  - bool: Boolean (true, false)
- Other
  - void: empty values (e.g., void main() {…})

int

- Typically, an int is stored in *4 bytes* (1 byte = 8 bits)
- The most significant bit of an int data type is the *sign bit*
  - 0: positive
  - 1: negative
- For example

  `00000000 00000000 00000000 00001111` = 15

- What's the decimal value of the following integer?

  `10000000 00000000 00000000 00000001` = ?

int

- C++ uses *two's complement* to encode negative numbers
- E.g., for -11
  - reverse the sign

```
00000000 00000000 00000000 00001011
```

int

- C++ uses *two's complement* to encode negative numbers
- E.g., for -11
  - reverse the sign

```
00000000 00000000 00000000 00001011
```

  - invert the bits (0 goes to 1, and 1 to 0)

```
11111111 11111111 11111111 11110100
```

int

- C++ uses *two's complement* to encode negative numbers
- E.g., for -11
  - reverse the sign

```
00000000 00000000 00000000 00001011
```

  - invert the bits (0 goes to 1, and 1 to 0)

```
11111111 11111111 11111111 11110100
```

  - add 1 to the resulting number

```
11111111 11111111 11111111 11110101
```

int

- A 32-bit int can store any integer in the range of -2^{31} and $2^{31}-1$
  - i.e., -2147483648 to 2147483647
  - max int: `01111111 11111111 11111111 11111111`
  - min int: `10000000 00000000 00000000 00000000`

int

- A 32-bit int can store any integer in the range of -2^{31} and $2^{31}-1$
  - i.e., -2147483648 to 2147483647
  - max int: `01111111 11111111 11111111 11111111`
  - min int: `10000000 00000000 00000000 00000000`
- When an int is assigned a value greater than its maximum value, overflow occurs
- Similarly, underflow occurs when a value smaller than the minimum value is assigned
- However, C++ does not inform you the errors

Example

```
#include <iostream>
using namespace std;
int main() {
	int num1 = 2147483647;
	int num2 = num1 + 1;
	cout << num2 << endl;
	return 0;
}
```

short, long and unsigned

- long is used for large integers (8 bytes)
- short is used for small integers (2 bytes)
  - For example: short minute_of_day;

short, long and unsigned

- long is used for large integers (8 bytes)
- short is used for small integers (2 bytes)
  - For example: short minute_of_day;
- unsigned is used to declare that the integer data type is *non-negative*
  - For example

    unsigned short a;

    unsigned long b;

    unsigned int c;
  - unsigned integers has no sign bit
    - i.e., the range of unsigned int is 0 to $2^{32}-1$

char

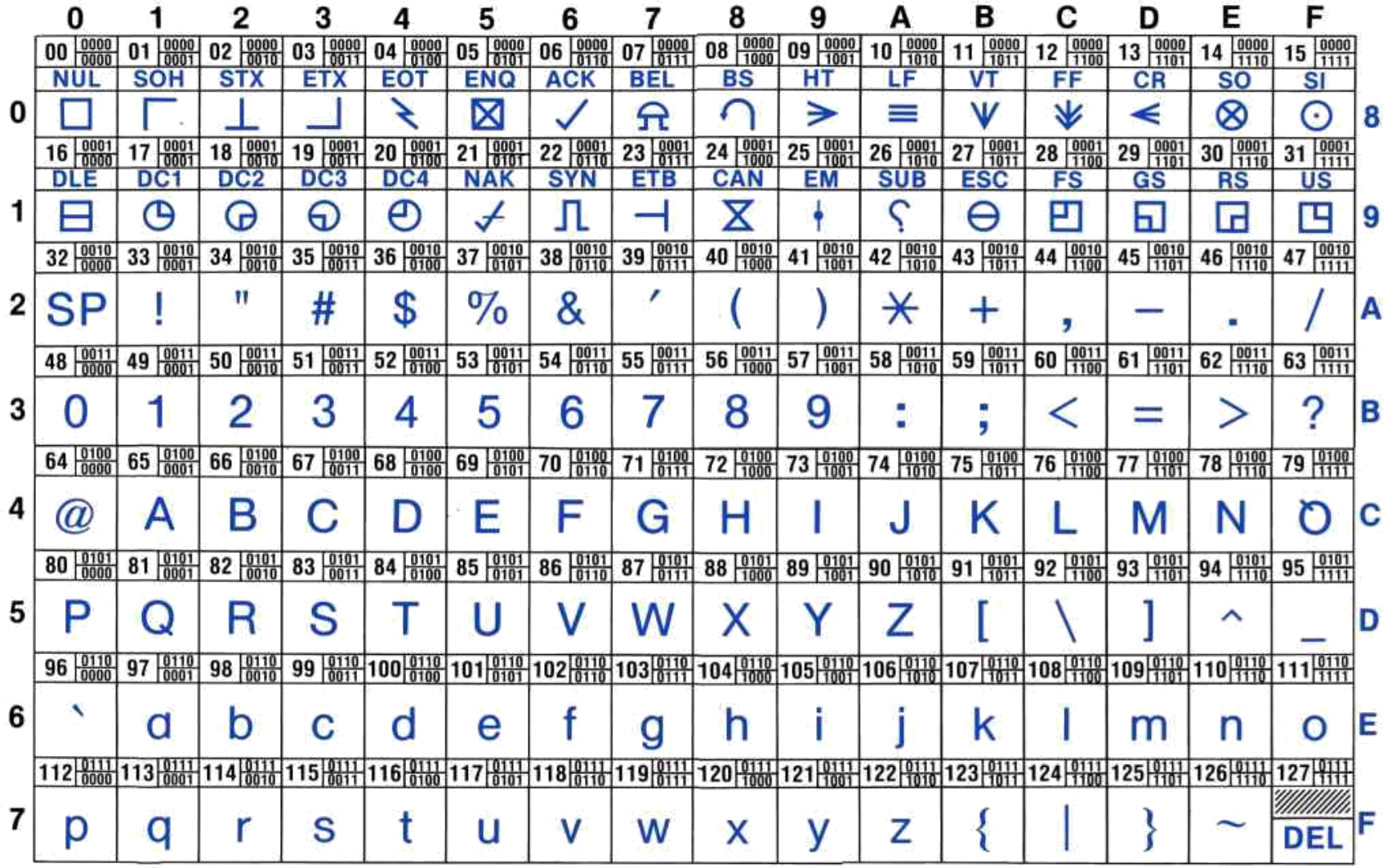
- Used to store a single character, enclosed by the single quotation mark

  char c = ‘a’;

  char c = ‘\n’;

- Characters are treated as small integers (and vice versa)
  - A char type takes 1 byte (8 bits, representing up to 256 characters)
  - `a’ is stored as `01100001`, equivalent to an integer 97
  - `b’ is stored as `01100010`, equivalent to an integer 98
  - …

ASCII Code



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	00 0000 0000	01 0000 0001	02 0000 0010	03 0000 0011	04 0000 0100	05 0000 0101	06 0000 0110	07 0000 0111	08 0000 1000	09 0000 1001	10 0000 1010	11 0000 1011	12 0000 1100	13 0000 1101	14 0000 1110	15 0000 1111	
	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	
0	□	⌜	⊥	⌟	↘	⊠	✓	𐤀	↶	➢	≡	⩔	⤋	⇐	⊗	⊙	8
	16 0001 0000	17 0001 0001	18 0001 0010	19 0001 0011	20 0001 0100	21 0001 0101	22 0001 0110	23 0001 0111	24 0001 1000	25 0001 1001	26 0001 1010	27 0001 1011	28 0001 1100	29 0001 1101	30 0001 1110	31 0001 1111	
	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
1	⊟	◷	◶	◵	◴	✓	⎍	⊣	⧖	⍿	⍁	⊖	◱	◱	◱	◳	9
	32 0010 0000	33 0010 0001	34 0010 0010	35 0010 0011	36 0010 0100	37 0010 0101	38 0010 0110	39 0010 0111	40 0010 1000	41 0010 1001	42 0010 1010	43 0010 1011	44 0010 1100	45 0010 1101	46 0010 1110	47 0010 1111	
2	SP	!	"	#	$	%	&	′	(	)	⋇	+	,	−	.	/	A
	48 0011 0000	49 0011 0001	50 0011 0010	51 0011 0011	52 0011 0100	53 0011 0101	54 0011 0110	55 0011 0111	56 0011 1000	57 0011 1001	58 0011 1010	59 0011 1011	60 0011 1100	61 0011 1101	62 0011 1110	63 0011 1111	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	B
	64 0100 0000	65 0100 0001	66 0100 0010	67 0100 0011	68 0100 0100	69 0100 0101	70 0100 0110	71 0100 0111	72 0100 1000	73 0100 1001	74 0100 1010	75 0100 1011	76 0100 1100	77 0100 1101	78 0100 1110	79 0100 1111	
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	C
	80 0101 0000	81 0101 0001	82 0101 0010	83 0101 0011	84 0101 0100	85 0101 0101	86 0101 0110	87 0101 0111	88 0101 1000	89 0101 1001	90 0101 1010	91 0101 1011	92 0101 1100	93 0101 1101	94 0101 1110	95 0101 1111	
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	D
	96 0110 0000	97 0110 0001	98 0110 0010	99 0110 0011	100 0110 0100	101 0110 0101	102 0110 0110	103 0110 0111	104 0110 1000	105 0110 1001	106 0110 1010	107 0110 1011	108 0110 1100	109 0110 1101	110 0110 1110	111 0110 1111	
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	E
	112 0111 0000	113 0111 0001	114 0111 0010	115 0111 0011	116 0111 0100	117 0111 0101	118 0111 0110	119 0111 0111	120 0111 1000	121 0111 1001	122 0111 1010	123 0111 1011	124 0111 1100	125 0111 1101	126 0111 1110	127 0111 1111	
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL	F

string

- A string is an array of characters
  - Both array and string will be introduced in detail in future lectures
- Strings are delimited by double quotation marks “”, and the identifier must be followed with [] or begin with *

```
char course_name[] = “Computer Programming";

char *course_name= “Computer Programming";

char initial[] = "C"; vs. char initial = ‘C’;
```

Floating Types

- Represent real numbers using floating point representation

```
float height;
double weight = 120.8;
long double number;
```

Floating Types

- Represent real numbers using floating point representation

  ```
  float height;
  double weight = 120.8;
  long double number;
  ```

- float takes 4 bytes, but is less accurate (7 digits after decimal point)
- double takes 8 bytes, but more accurate (15 digits after decimal point)
  - the default type for floating type in C++
- long double is even more precise, but rarely used

Floating Types

- Represent real numbers using floating point representation

  ```
  float height;
  double weight = 120.8;
  long double number;
  ```

- float takes 4 bytes, but is less accurate (7 digits after decimal point)
- double takes 8 bytes, but more accurate (15 digits after decimal point)
  - the default type for floating type in C++
- long double is even more precise, but rarely used
- Exponent representation is acceptable, e.g.,

  ```
  double a = 1.23e3;
  ```

sizeof

- sizeof can be used to find the number of bytes needed to store an object (which can be a variable or a data type)
- Its result is typically returned as an unsigned integer

```
#include <iostream>
using namespace std;
int main() {
        int a = 4;
        cout << sizeof(a) << endl;
        cout << sizeof(int) << endl;
        cout << sizeof(double) << endl;
        cout << sizeof(long double) << endl;
        return 0;
}
```

Type Conversion

- Very often, we need to convert data from one type to another

For example:

Each pig weighs 280.3 lbs (float)

Each boat can carry 615.2 lbs (float)

How many pigs a boat can carry? (int)

Type Conversion

- Very often, we need to convert data from one type to another

For example:

Each pig weighs 280.3 lbs (float)

Each boat can carry 615.2 lbs (float)

How many pigs a boat can carry? (int)

```
float pig_weight = 280.3, boat_load = 615.2;
int n_pig = boat_load/pig_weight;
```

int float

Type Conversion

- *Implicit* type conversion
  - *Binary expression*: lower-ranked operand is promoted to higher-ranked operand, e.g.,

```
int r = 2;
double pi = 3.14159;
cout << pi * r * r << "\n";
```

9. long double
8. double
7. float

6. long long
5. long
4. int
3. short

2. char

1. bool

Type Conversion

- *Implicit* type conversion
  - *Binary expression*: lower-ranked operand is promoted to higher-ranked operand, e.g.,

```
int r = 2;
double pi = 3.14159;
cout << pi * r * r << "\n";
```

  - *Assignment*: right operand is promoted/demoted to match the variable type on the left, e.g.,

```
double a = 1.23;
int b = a;
```

9. long double
8. double
7. float

6. long long
5. long
4. int
3. short

2. char

1. bool

Type Conversion

- *Explicit* type conversion (type-casting)

```
int a = 3;
double b = (double)a;
```

9. long double
8. double
7. float

6. long long
5. long
4. int
3. short

2. char

1. bool

Type Conversion

- *Explicit* type conversion (type-casting)

```
int a = 3;
double b = (double)a;
```

- *Demoted values might change or become invalid*

```
double a = 3.1;
int b = (int)a;
cout << b << endl;
```

9. long double
8. double
7. float

6. long long
5. long
4. int
3. short

2. char

1. bool

Type Conversion

- *Explicit* type conversion (type-casting)

```
int a = 3;
double b = (double)a;
```

- *Demoted values might change or become invalid*

```
double a = 3.1;
int b = (int)a;
cout << b << endl;
```

```
double a = 3.9;
int b = (int)a;
cout << b << endl;
```

9. long double
8. double
7. float

6. long long
5. long
4. int
3. short

2. char

1. bool

Example

```
#include <iostream>
using namespace std;
int main() {
      int i = 5; char a = 'B';
      double x = 1.57;
      i = i + x; cout << i << endl;
      x = x * a; cout << x << endl;
      return 0;
}
```

Variable and Constant

- Every variable/constant has ***5*** attributes
- *Address*: location of data in memory storage
- *Value*: content in memory storage
- *Name*: identifier of the variable, needs to be declared
- *Type*: C++ is a strictly typed language, variables and constants must belong to a data type
  - E.g., numerical, character, logic, other…
- *Scope*: it defines the region within a program where the variable/constant can be accessed, and also the *conflict domain*

Scope

- Scope of a variable/constant refers to the region of a program where the variable/constant is visible (can be accessed)

Example I:

- The accessibility of variable '*a*' is within function '*foo*'
- Trying to access '*a*' in '*main*' will cause an error

```
#include <iostream>
using namespace std;
void foo() {
        int a = 0;
        cout << "a in foo: " << a << endl;
}
int main() {
        foo();
        cout << "a in main: " << a << endl;
        return 0;
}
```

Scope

- Scope of a variable/constant refers to the region of a program where the variable/constant is visible (can be accessed)

Example II:

- Defined two '*a*' within '*main*'
- Will cause an error due to conflict domain

```
#include <iostream>
using namespace std;
int main() {
        int a = 0;
        int a = 1;
        cout << "a in main: " << a << endl;
        return 0;
}
```

Scope

- Scope of a variable/constant refers to the region of a program where the variable/constant is visible (can be accessed)

Example III:

- Defined two variables with the same name '*a*'
- Their accessibilities are within '*foo*' and '*main*', respectively

```
#include <iostream>
using namespace std;
void foo() {
        int a = 0;
        cout << "a in foo: " << a << endl;
}
int main() {
        int a = 1;
        foo();
        cout << "a in main: " << a << endl;
        return 0;
}
```

Scope

- Scope of a variable/constant refers to the region of a program where the variable/constant is visible (can be accessed)

Example IV:

- Defined a *global* variable '*a*'
- Its accessibility is the *entire* program

```
#include <iostream>
using namespace std;
int a = 0;
void foo() {
        cout << "a in foo: " << a << endl;
}
int main() {
        foo();
        cout << "a in main: " << a << endl;
        return 0;
}
```

Scope

- Scope of a variable/constant refers to the region of a program where the variable/constant is visible (can be accessed)

Example V:

- Defined a *global* variable '*a*' and a *local* variable '*a*' within '*main*'
- What's the output of the program??

```
#include <iostream>
using namespace std;
int a = 0;
int main() {
        int a = 1;
        cout << "a in main: " << a << endl;
        return 0;
}
```

Define Scope using Namespace

- A scope can be defined in many ways: by {}, functions, classes, and namespaces
- *Namespace* is used to *explicitly* define the scope. A namespace can ONLY be defined in global or namespace scope.
- The *scope operator* **::** is used to resolve scope for variables of the same name.

```
#include <iostream>
using namespace std;
int a = 0;
namespace level1 {
        int a = 1;
        namespace level2 {
                int a = 2;
        }
}
int main() {

        cout <<    a << endl;
        cout << level1::a << endl;
        cout << level1::level2::a << endl;
        return 0;
}
```

```
#include <iostream>
using namespace std;
int a = 0;
namespace level1 {
	int a = 1;
	namespace level2 {
		int a = 2;
	}
}
int main() {
	int a = 3;
	cout << a << endl;
	cout << level1::a << endl;
	cout << level1::level2::a << endl;
	return 0;
}
```

```
#include <iostream>
using namespace std;
int a = 0;
namespace level1 {
        int a = 1;
        namespace level2 {
                int a = 2;
        }
}
int main() {
        int a = 3;
        cout << ::a << endl;
        cout << level1::a << endl;
        cout << level1::level2::a << endl;
        return 0;
}
```

Today's Outline

- C++ basic syntax
- Variable and constant
- Operators
- Basic I/O

Operators

- An operator specifies an operation to be performed on some values
  - These values are called the operands of the operator

- Some examples: +, -, *, /, %, ++, --, >>, <<

Operators

- An operator specifies an operation to be performed on some values
  - These values are called the operands of the operator

- Some examples: +, -, *, /, %, ++, --, >>, <<

- Some of these have meanings that depend on the context
  - e.g., << means different operations in

```
cout << a << endl;
int b = a << 1;
```

Assignment Operator =

- Generic form: *variable* = *expression* **;**
- Write the value of *expression* into the memory storage of *variable*
- An *expression* is a combination of constants, variables, and function calls that evaluate to a result
- Examples:

```
float a = 2.0 * 4.0 * 8.0;
float b = a – sqrt(a);
char x = 'a';
```

Assignment Operator =

- 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

Assignment Operator =

- 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

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

Examples of Assignment Statements

```
/* Invalid: left hand side must be a variable */
x + 10 = y;
```

Examples of Assignment Statements

```
/* Invalid: left hand side must be a variable */
x + 10 = y;

/* Assignment to constant is not allowed */
const int a = 2;
a = 3;
```

Examples of Assignment Statements

```
/* Invalid: left hand side must be a variable */
x + 10 = y;

/* Assignment to constant is not allowed */
const int a = 2;
a = 3;

/* Valid but not easy to understand */
a = (b = 2) + (c = 3);
```

Examples of Assignment Statements

```
/* Invalid: left hand side must be a variable */
x + 10 = y;

/* Assignment to constant is not allowed */
const int a = 2;
a = 3;

/* Valid but not easy to understand */
a = (b = 2) + (c = 3);

/* Avoid complex expressions */
b = 2, c = 3;
a = b + c;
```

Swapping the Values

- If we want to swap the content of two variables, ***a*** and ***b***
- What's the problem of the following program?

```
int a = 3;
int b = 4;
a = b;
b = a;
```

Swapping the Values

- We need to make use of a temporary variable:

```
int a = 3;
int b = 4;
int c;       // a buffer for value swapping
```

Swapping the Values

- We need to make use of a temporary variable:

```
int a = 3;
int b = 4;
int c;      // a buffer for value swapping
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 Assignment Operators

- Generic form of efficient assignment operators

  variable op = *expression* ;

  where *op* is operator; the meaning is

  variable = *variable op* (*expression*) ;

- Efficient assignment operators include

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

- Examples

```
a += 5;        // is the same as a = a+5;
a %= 5;        // is the same as a = a%5;
a *= b+c;      // is the same as a = a*(b+c)
```

Increment & Decrement Operators

- Increment and decrement operators: `++` and `--`
  - `k++` and `++k` is equivalent to `k=k+1;`
  - `k--` and `--k` is equivalent to `k=k-1;`

Increment & Decrement Operators

- Increment and decrement operators: `++` and `--`
  - `k++` and `++k` is equivalent to `k=k+1;`
  - `k--` and `--k` is equivalent to `k=k-1;`
- Post-increment and post-decrement: `k++` and `k--`
  - k's value is altered **AFTER** the expression is evaluated

  e.g., `a = k++` is equivalent to (1) `a = k`, (2) `k = k+1`
- Pre-increment and pre-decrement: `++k` and `--k`
  - k's value is altered **BEFORE** the expression is evaluated

  e.g., `a = ++k` is equivalent to (1) `k = k+1`, (2) `a = k`

Example

```
#include <iostream>
using namespace std;
int main() {
      int x = 3;
      cout << ++x; // (1) x = x+1 (2) cout << x
      cout << x++; // (1) cout << x (2) x = x+1
      cout << x;
      return 0;
}
```

Example

```
#include <iostream>
using namespace std;
int main() {
      int x = 3;
      cout << ++x;
      cout << x++;
      cout << x;
      return 0;
}
```

	old x	new x	cout
int x = 3;		3	
cout << ++x;	3	4	4
cout << x++;	4	5	4
cout << x;			5

What values are printed?

```
int a = 0, b = 0;
cout << "b = " << b << endl;

a = 0;
b = 1+(a++);
cout << "b= " << b << endl;
cout << "a= " << a << endl;

a = 0;
b = 1+(++a);
cout << "b= " << b << endl;
cout << "a= " << a << endl;
```

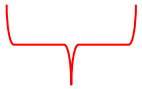
```
a = 0;
b = 1 + (a++);
```

0

1. Evaluates a++, (value:0)
2. Computes a = a+1
3. b = 1 + 0

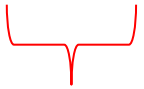
   = 1

```
a = 0;
b = 1 + (a++);
```



1. Evaluates a++, (value:0)
2. Computes a = a+1
3. b = 1 + 0
   = 1

```
a = 0;
b = 1 + (++a);
```



1. Computes a = a+1
2. Evaluates ++a (value:1)
3. b = 1 + 1
   = 2

What values are printed?

```
int a = 0, b = 0;
cout << "b = " << b << endl;

a = 0;
b = 1+(a++); // (1) b=1+a (2) a=a+1
cout << "b = " << b << endl;
cout << "a = " << a << endl;

a = 0;
b = 1+(++a); // (1) a=a+1 (2) b=1+a
cout << "b = " << b << endl;
cout << "a = " << a << endl;
```

Output:

```
b = 0
b = 1
a = 1
b = 2
a = 1
```

Precedence & Associativity of Operators

- An expression may have more than one operators and its precise meaning depends on the *precedence* and *associativity* of the involved operators
- Precedence: order of evaluation for different operators
  - determines how an expression like x R y S z should be evaluated, where R and S are different operators, e.g., x + y / z.
- Associativity: order of evaluation for operators with the same precedence
  - means whether an expression like x R y R z, where R is an 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

- What's the value of *a*, *b*, *c* after executing the following statements?

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

- Which of the following interpretation is right?

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

Precedence & Associativity of Operators

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

Precedence & Associativity of Operators

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

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

Precedence & Associativity of Operators

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

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

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

Bitwise Operators

- Bitwise AND **&**
  - Compute AND on every bit of two numbers
  - The result of AND is 1 only if both bits are 1

```
short a = 3, b = 5, c = a & b;
cout << c << endl; // 1
// a = 00000011
// b = 00001001
// c = 00000001
```

- Bitwise OR **|**
  - Compute OR on every bit of two numbers
  - The result of OR is 1 as long as one of the bits is 1

```
short a = 3, b = 5, c = a | b;
cout << c << endl; // 7
// a = 00000011
// b = 00001001
// c = 00001011
```

Bitwise Operators (cont'd)

- Bitwise XOR **^**
  - Compute XOR on every bit of two numbers
  - The result of XOR is 1 if the two bits are different

```
short a = 3, b = 5, c = a ^ b;
cout << c << endl; // 6
// a = 00000011
// b = 00001001
// c = 00001010
```

- Bitwise NOT **~**
  - Takes one number and inverts all of its bits

```
char a = 254; int b = ~a;
cout << b << endl; // 1
// a = 11111110
// b = 00000001
```

Bitwise Operators (cont'd)

- Left shift **<<** and right shift **>>**
  - *a* << *n*: left shifts the bits of *a* for *n* digits
  - *a* >> *n*: right shifts the bits of *a* for *n* digits
  - Note that whether << or >> is explained as bit shift depends on context
    - e.g., in cout << x, << is the output operator
      in cout >> x, >> is the input operator

```
int a = 3, b = 1;

int c = a << b, d = a >> b;
cout << c << endl; // 6
cout << d << endl; // 1
```

Example I

- What's the output of the following statements?

```
char x = 6;

int a = (x >> 1) & 1;
cout << a << endl;

int b = (x >> 3) & 1;
cout << b << endl;
```

Example II

- Print a char type in binary format

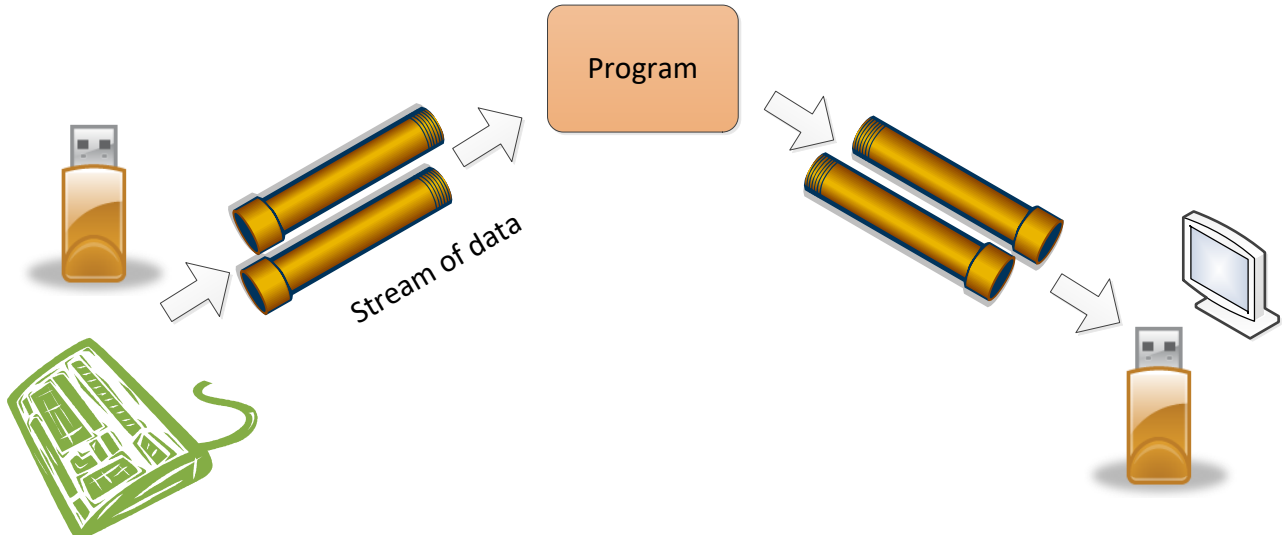
```
char x = 112;

int b0 = (x >> 0) & 1;  int b1 = (x >> 1) & 1;
int b2 = (x >> 2) & 1;  int b3 = (x >> 3) & 1;
int b4 = (x >> 4) & 1;  int b5 = (x >> 5) & 1;
int b6 = (x >> 6) & 1;  int b7 = (x >> 7) & 1;

cout << b0 << b1 << b2 << b3 << b4 << b5 << b6 << b7 << endl;
```

Today's Outline

- C++ basic syntax
- Variable and constant
- Operators (and punctuators)
- Basic I/O

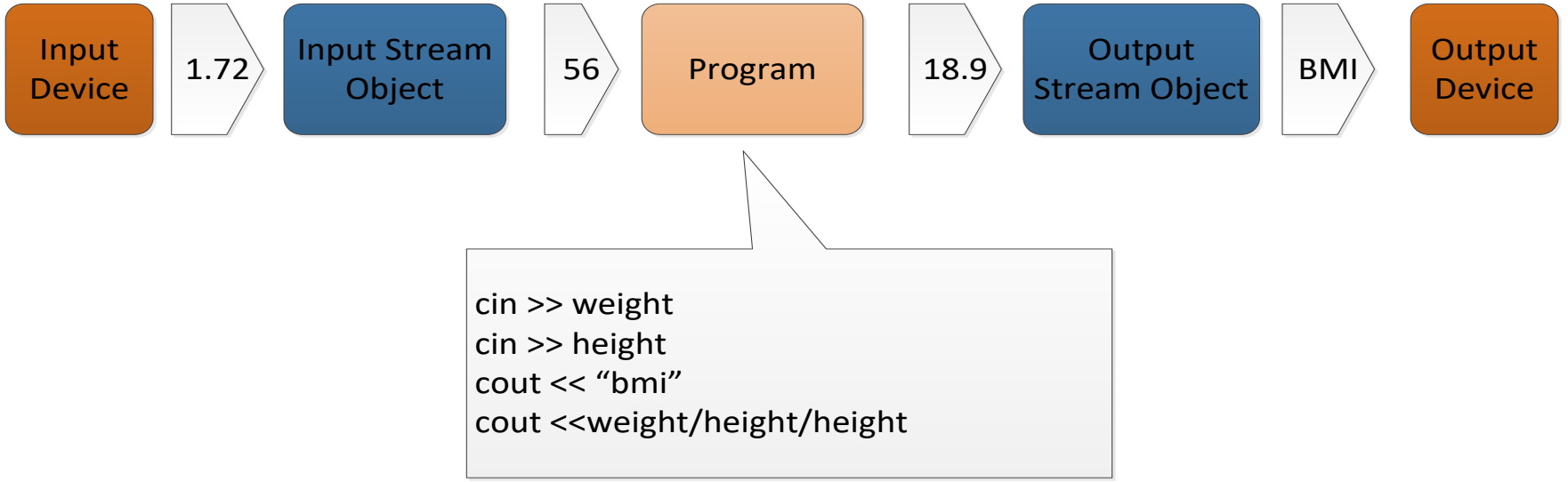
Basic I/O – Keyboard and Screen

- A program can do little if it cannot take input and produce output
- Most programs read user input from keyboard and secondary storage
- After process 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 input operator **(>>)** and output operator **(<<)**

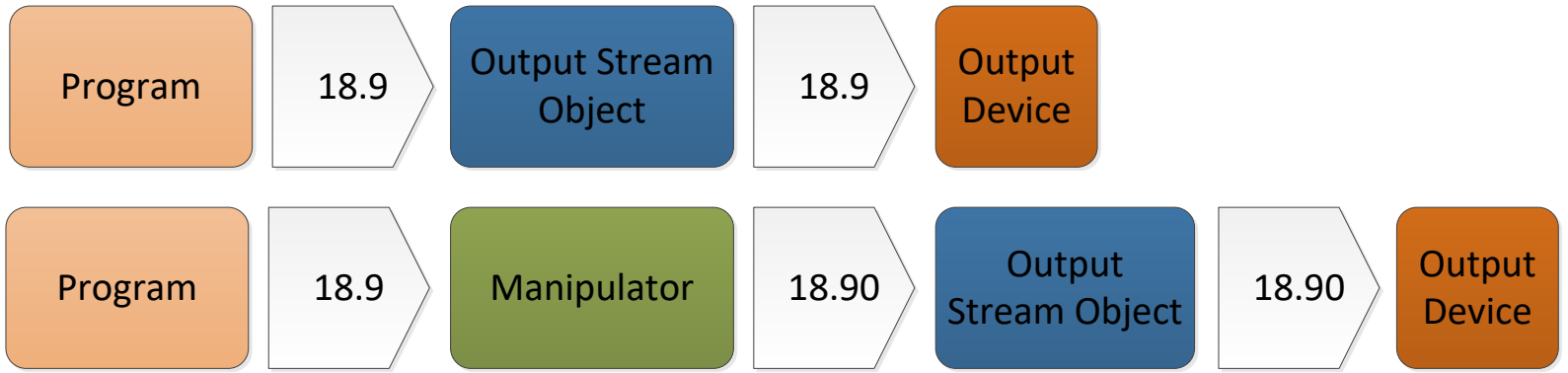


cout: Output Operator (<<)

- Preprogrammed for all standard C++ data types
- It sends bytes to an output stream object, e.g. cout

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

- Calling member function width(*width*) or using setw manipulator
  - setw requires "ipmanip", i.e., #include <iomanip>
- Leading blanks are added to any value that has fewer characters than '*width*'
- If formatted output exceeds the width, the entire value prints
- Effect lasts for one field only

Approach	Example	Output (◆: space key)

1. **cout.width**(*width*)
 	
2. **setw**(*width*)
	cout.width(5); //or cout << setw(5);
	cout << 123 << endl;
	cout << 123 << endl;
	
	cout << setw(5); //or cout.width(5);
	cout << 1234567 << endl;
		
		◆◆123
		123
		
		1234567

cout: Floating-Point Precision and Format

- You can control the *precision* and *format* of floating point print
- Must #include <iomanip>
- Floating-point precision is 6 digits by default
- Use fixed, scientific and setprecision manipulators to change the precision value and printing format
- Effect is *permanent*

cout: Floating-Point Precision and Format

- Default precision (*6* digits, *5* digits after decimal points) and format

Example	Default output
cout << 1.23 << endl; cout << 1.230 << endl; cout << 1.2345678 << endl; cout << 0.000012345678 << endl;	1.23 1.23 1.23457 1.23457e-05

cout: Floating-Point Precision and Format

- Default precision (*6* digits, *5* digits after decimal points) and format
- cout << **fixed**: always uses the fixed-point notation (*6 significant digits after the decimal point*)

Example	Default output	After cout << fixed;
cout << 1.23 << endl; cout << 1.230 << endl; cout << 1.2345678 << endl; cout << 0.000012345678 << endl;	1.23 1.23 1.23457 1.23457e-05	1.230000 1.230000 1.234568 0.000012

cout: Floating-Point Precision and Format

- Default precision (*6* digits, *5* digits after decimal points) and format
- cout << **fixed**: always uses the fixed-point notation (*6 significant digits after the decimal point*)
- cout<< **scientific**: always uses the scientific notation

Example	Default output	After cout << fixed;	After cout << scientific;
cout << 1.23 << endl; cout << 1.230 << endl; cout << 1.2345678 << endl; cout << 0.000012345678 << endl;	1.23 1.23 1.23457 1.23457e-05	1.230000 1.230000 1.234568 0.000012	1.230000e+00 1.230000e+00 1.234568e+00 1.234568e-05

cout: Floating-Point Precision and Format

- Normally, setprecision(n) means output n significant digits

Example	Output
cout << setprecision(2); cout << 1.234 << endl; cout << 0.0000001234 << endl;	 1.2 1.2e-07

cout: Floating-Point Precision and Format

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

Example	Output
cout << setprecision(2); cout << 1.234 << endl; cout << 0.0000001234 << endl; cout << fixed; cout << 1.234 << endl; cout << 0.0000001234 << endl; cout << scientific << 1.234 << endl; cout << 0.0000001234 << endl;	 1.2 1.2e-07 1.23 0.00 1.23e+00 1.23e-07

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

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; *// hexidecimal* cout << dec << 11 << endl;	13 b 11

cout: Other Manipulators

Manipulators	Example	Output

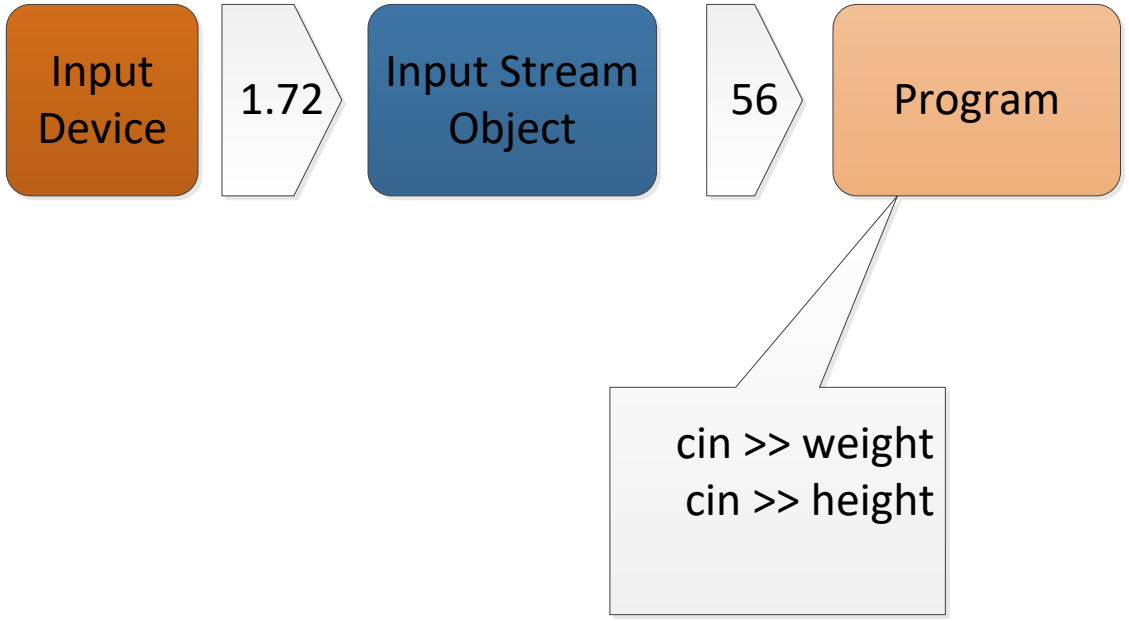
fill	cout << setfill('*');
	cout << setw(10);
	cout << 5.6 << endl;
	cout << setw(10);
	cout << 57.68 << endl;
		
		*******5.6
		
		*****57.68

radix	cout << oct << 11 << endl; // *octal*
	cout << hex << 11 << endl; // *hexidecimal*
	cout << dec << 11 << endl;
		13
		b
		11

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

cin: Input Operator (>>)

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



cin: 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 Hi	a H	
		cin >> x >> y;	Hi	H	i
String	char x[20]; char y[20];	cin >> x;	hello	hello	
		cin >> x >> y	Hello World	Hello	World

cin: 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 Hi	a H	
		cin >> x >> y;	Hi	H	i
String	char x[20]; char y[20];	cin >> x;	hello	hello	
		cin >> x >> y	Hello World	Hello	World

cin: 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 Hi	a H	
		cin >> x >> y;	Hi	H	i
String	char x[20]; char y[20];	cin >> x;	hello	hello	
		cin >> x >> y	Hello World	Hello	World

Expected Outcome

- Describe the basic syntax and data types of C++ language
- Explain the concepts of variable, constant, and their scope
- Declare variable and constant under different scopes
- Perform update on variables via different operators
- Able to output variables' values to screen with different precision and format
- Able to read value from keyboard and assign to variable