CS2311 Computer Programming

LT2: Basic Syntax

Part I: Variables and Constants

Outline

- C++ syntax
- Variable type, scope, and declaration
- Constants

A General C++ Program

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

int main() {
  /* Place your code here! */

  return 0;
}
```

Syntax

- Like any language, C++ has an *alphabet* and *rules* for putting together words and punctuation 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***

Syntax – A Simple Program

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

int main() {

  cout << "Hello, world! " << endl;
  return 0;
}
```

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

int main() {

    cout << "Hello World!" << endl;
    return 0;
}
```

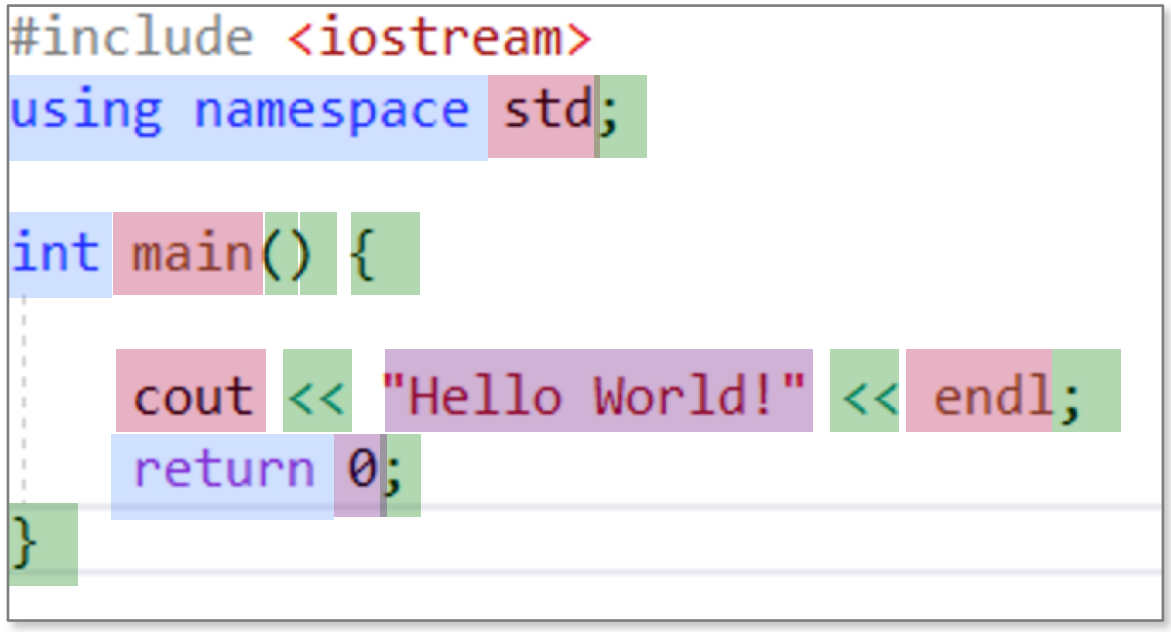
Syntax – Tokens

- Tokens can be categorized into:
  - keywords
    - e.g., **namespace**, **return**, **int**
  - identifiers
    - e.g., user-defined **variables**, **objects**, **functions**, etc.
  - string constants
    - e.g., "Hello"
  - numeric constants
    - e.g., 7, 11, 3.14
  - operators
    - e.g., **+**, **/**
  - punctuators
    - e.g., **;** and **,**

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

int main() {

    cout << "Hello World!" << endl;
    return 0;
}
```

Syntax – A Simple Program

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

int main() {

  cout << "Hello, world! " << endl;
  return 0;
}
```



keywords

punctuators

identifiers

string constants/literals

numeric literals

Keywords (reserved words) – covered in this course

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

Keywords

- Each keyword has a reserved meaning and cannot be used as identifiers
  - Can we have a variable called "`main`"?

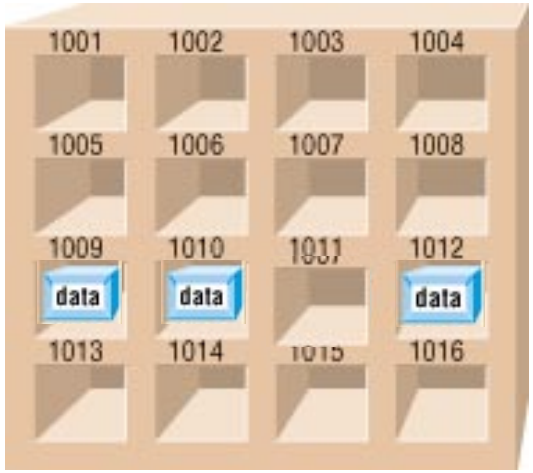
Identifiers

- Identifiers give unique names to objects, variables, functions, etc. with respect to their scopes (detail later)
- Keywords cannot be used as identifiers
- An identifier is composed of a sequence of letters, digits and underscores
  - No hyphen!
  - E.g. `myRecord`, `point3D`, `last_file`
- An identifier **must** begin with either a letter or an underscore (not recommended)
  - valid identifiers: `_income`, `record1`, `my_income`, `My_income`
  - Invalid identifiers: `3D_Point`, `my-income`
- Always use meaningful names for identifiers
  - Bad examples: x, xx, a, b, …

Variables and Constants

Variables and Constants

- Data stored in memory, in binary format
  - They do not exist after the program execution.



- A variable: its value may be changed during program execution

- A constant: its value will NOT be changed during program execution

Variables and Constants

- Every variable/constant have 3 attributes: **name**, **type**, and **scope**
  - **Name**: identifier of the variable
  - **Type**: variables/constants must belong to a data type, either ***predefined*** or ***user-defined***.
  - **Scope**: it defines ***where*** the variable can be accessed, and also the ***conflict domain*** for identifiers

Variable Declaration Format

- Format

  data_type variable/constant_identifier;

- Variables and constants must be *declared before use*
  - `int age;`

- Variable names
  - 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 _

Variable Names

- **Capitalized** and **lower case** letters are different

- Examples:

```
int age;
int age1, age2;
```

*Their values are **undefined** at this point*

- Optionally, the initial value of a variable may be set with declaration.

```
int age = 18;
int age1=18, age2=23;
int age1 = 18, age2 = 23; // Space is okay
```

Variable Names

Which of these are **valid** variable names?

A. `you`
B. `U2`
C. `CityU_CS`
D. `$Cake`
E. `\you`
F. `CityU-CS`

C++ predefined data types

- Numerical
  - `int`: Integers (`1`, `1743`, `0`, `-45`)
  - `float`, `double`: real numbers (`0.25`, `6.45`, `3.01e-5`)

```
float x;
double z=1.0;
```

- Character
  - `char`: a single ASCII character ('a', 'e', 'o', '\n', '\\', '\"')

```
char c;
```

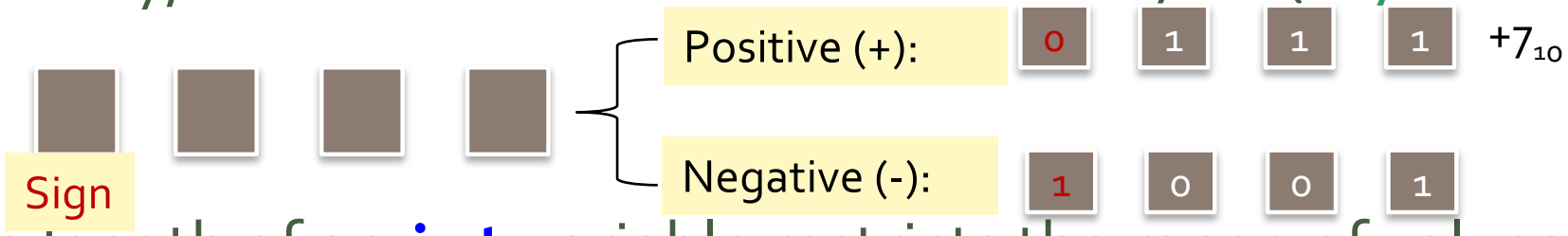
- Logical
  - `bool`: boolean (`true`, `false`)

```
bool valid;
```

- Other
  - `void` : empty values

int

- Typically, an **int** variable is stored in four bytes (**1 byte = 8 bits**).



- The length of an **int** variable restricts the range of values it can store, e.g., a 32-bit **int** can store any integer in the range of -2^{31} and $2^{31}-1$, i.e. -2147483648 to 2147483647

$-(2^{31}-1)$	11111111 11111111 11111111 11111111 - 01111111 11111111 11111111 11111111	$2^{31}-1$
	00000000 00000000 00000000 00000000	0
	10000000 00000000 00000000 00000000	-2^{31}

- When an **int** is assigned a value greater than its maximum value, overflow occurs and this gives illogical results; similarly underflow may occur when a value smaller than the minimum value is assigned. However, C++ does **NOT** inform you the errors.

short, long and unsigned

- **short**, **long** and **unsigned** are special data types for integers.

```
short x;
long x;
```

- **short** is used for small integers to conserve space (**2 bytes**).
- **long** is used for large integers (**8 bytes**).

- **unsigned** is of the same size as **int** (**4 bytes**) except it assumes the value to be stored is positive or zero. Thus the sign bit can be conserved it can store a positive integer larger than the maximum value of **int** (which is $2^{31}-1$).
- The range of an **unsigned int** is from **0** to $2^{32}-1$

00000000 00000000 00000000 00000000 - 11111111 11111111 11111111 11111111

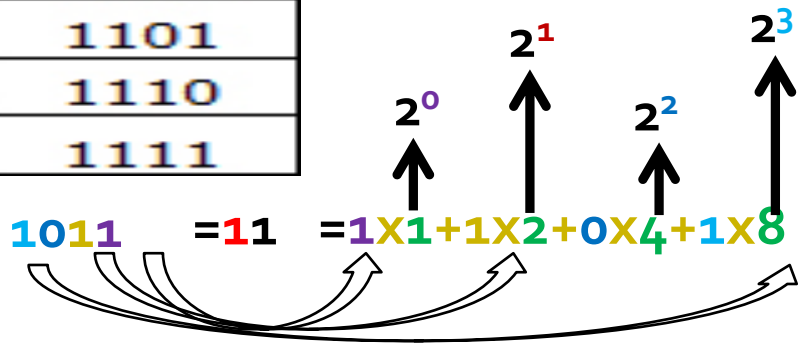
Data Type `char`

- Used to store a single ASCII character, enclosed by the single quotation mark
  - `char c = 'a';`
  - `char c = '\n';`
- ASCII
  - A character takes **one byte**, that is 8 bits, `0` or `1`
    - 'a' is stored as the following bit pattern **0 1 1 0 0 0 0 1**
    - It is equivalent to an integer 97 (= $2^6 + 2^5 + 2^0$)

Table of Numeric Conversions	
Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

$11 = 1 \times 10^0 + 1 \times 10^1$

Decimal: base is 10!

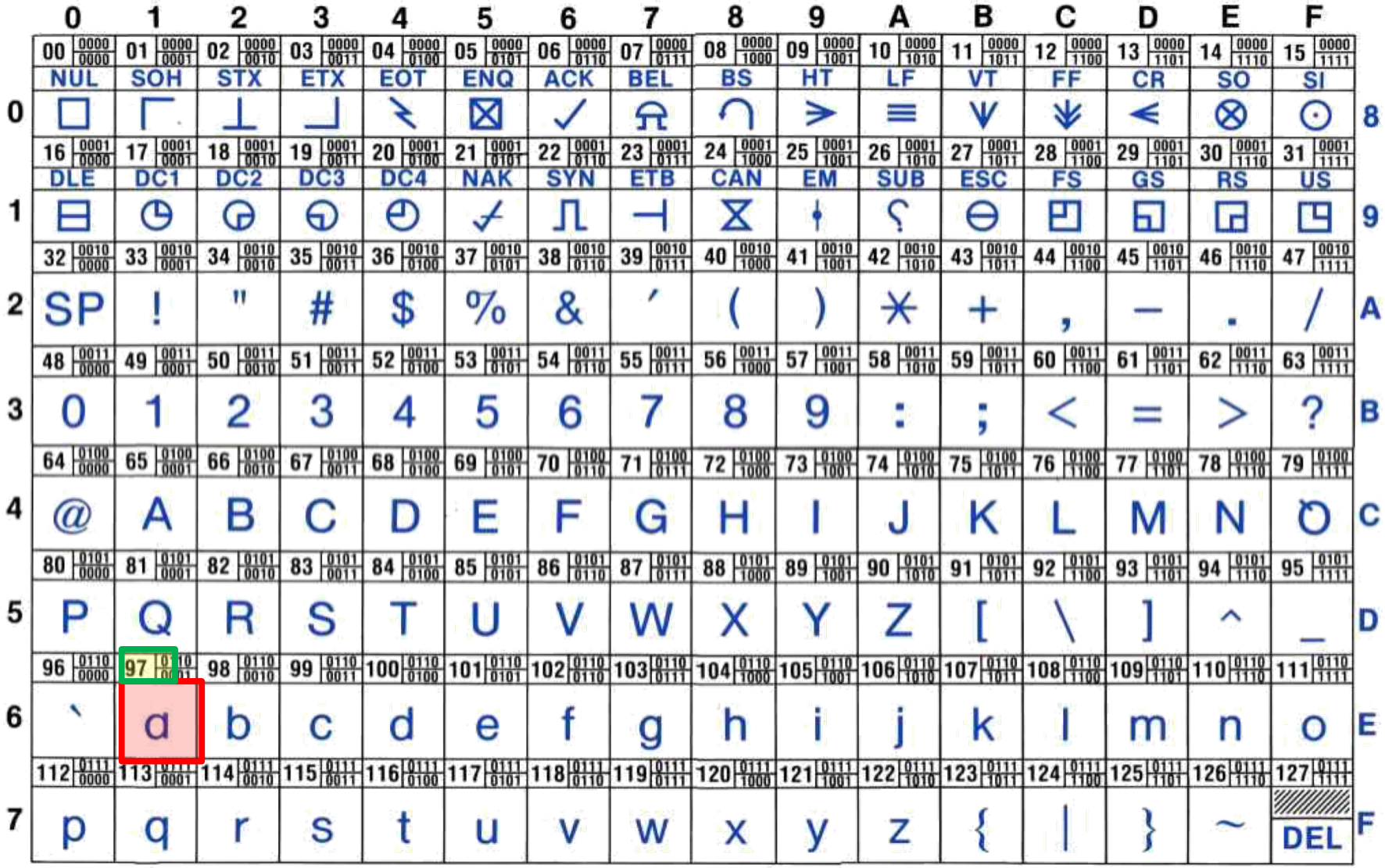


Binary: base is 2!

Data Type `char`

- Used to store a single ASCII character, enclosed by the single quotation mark
  - `char c = 'a';`
  - `char c = '\n';`
- ASCII
  - A character takes **one byte**, that is 8 bits, `0` or `1`
    - 'a' is stored as the following bit pattern **0 1 1 0 0 0 0 1**
    - It is equivalent to an integer 97 (= $2^6 + 2^5 + 2^0$)
- Characters are (*almost the same as*) integers
  - Characters are treated as small integers, and conversely, small integers can be treated as characters
  - $2^8 = 256$, it can represent up to 256 integers

ASCII Code


0 1 2 3 4 5 6 7 8 9 A B C D E F
00 NUL 01 SOH 02 STX 03 ETX 04 EOT 05 ENQ 06 ACK 07 BEL 08 BS 09 HT 10 LF 11 VT 12 FF 13 CR 14 SO 15 SI
16 DLE 17 DC1 18 DC2 19 DC3 20 DC4 21 NAK 22 SYN 23 ETB 24 CAN 25 EM 26 SUB 27 ESC 28 FS 29 GS 30 RS 31 US
32 SP 33 ! 34 " 35 # 36 $ 37 % 38 & 39 ' 40 (41) 42 * 43 + 44 , 45 - 46 . 47 /
48 0 49 1 50 2 51 3 52 4 53 5 54 6 55 7 56 8 57 9 58 : 59 ; 60 < 61 = 62 > 63 ?
64 @ 65 A 66 B 67 C 68 D 69 E 70 F 71 G 72 H 73 I 74 J 75 K 76 L 77 M 78 N 79 O
80 P 81 Q 82 R 83 S 84 T 85 U 86 V 87 W 88 X 89 Y 90 Z 91 [92 \ 93] 94 ^ 95 _
96 ` 97 a 98 b 99 c 100 d 101 e 102 f 103 g 104 h 105 i 106 j 107 k 108 l 109 m 110 n 111 o
112 p 113 q 114 r 115 s 116 t 117 u 118 v 119 w 120 x 121 y 122 z 123 { 124 | 125 } 126 ~ 127 DEL

char as integers

- Any integer expression can be applied to `char` type variables

```
char c = 'a';      // c = 97
c = c + 1;         // c = 98
cout << "variable c+1 has the character " << c;
```

  - The output is "`variable c+1 has the character b`".

Two's complement

- The way that computers represent integers
  - For a positive integer, two's complement is the same as the integer

Decimal number 11 in memory

  - For a negative integer, e.g. -11
    - Reserve the sign: 11

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

    - Add 1 to the resulting number

Strings

- A string is a sequence of characters.
  - A **string** is treated as an array of characters. We call it **cstring**. (Another type of string is a **string** object)

- Strings are delimited by double quotation marks "", and the identifier must be followed with []
  - `char lecture[] = "CS2311 Lecture02"; or`
  - `char * lecture = "CS2311 Lecture02";`
  - `char lecture[] = "C"; vs. char lecture = 'C';`

- How to display "hello"?
  - Remember escape sequences?

```
char name[] = "\"hello\"";
```

`float` types

- Represent real numbers using the floating point representation
  - `float height;`
  - `double weight = 120.82;`
  - `long double number;`
- `float` uses less memory (4 bytes), but is less accurate (7 digits after decimal point); `double` uses more memory (8 bytes) but more accurate (15 digits after decimal point)
- We use `double` most of the time. It's also the default type for floating numbers in C++.
- Exponent representation is also acceptable,
  - e.g., `1.23e2` (which is 1.23×10^2) and `3.367e-4` (awhich is 3.367×10^{-4})

```
double weight = 1.23e2; // weight = 123.0
```

The `sizeof` operator

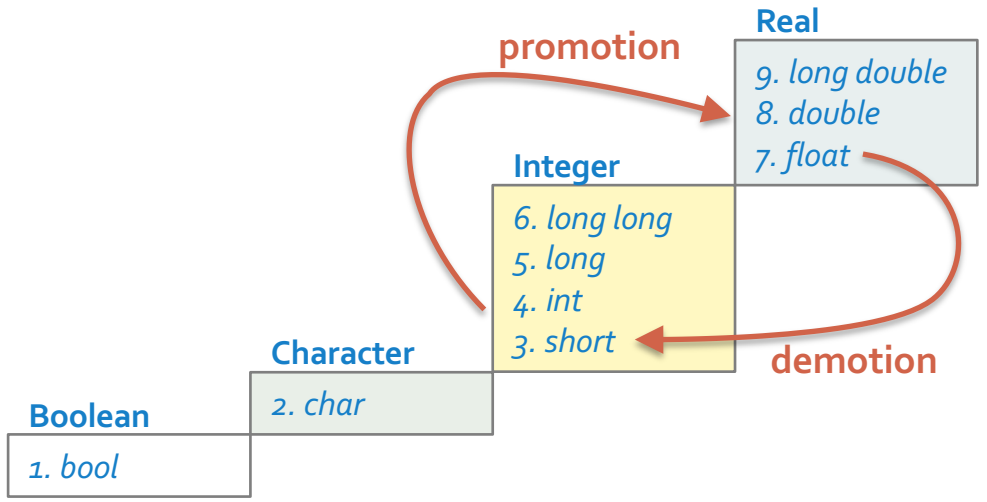
- **`sizeof`** can be used to find *the number of bytes needed to store an object* (which can be a **variable** or a **data type**)
- Its result is typically returned as an unsigned integer, e.g.,

```
int length1, length2;
double x;
length1 = sizeof(int);
cout << length1 << endl;
// same as       length1 = sizeof(length1);
// or            length1 = sizeof(length2);
length2 = sizeof(x);  // same as sizeof(double);
cout << length2 << endl;
```

Data Type Conversion

- Arithmetic conversions occur if necessary for the operands of a **binary operator**

- A `char` can be used in any expression where an `int` may be used, e.g., 'a' + 1 is equal to 97 + 1



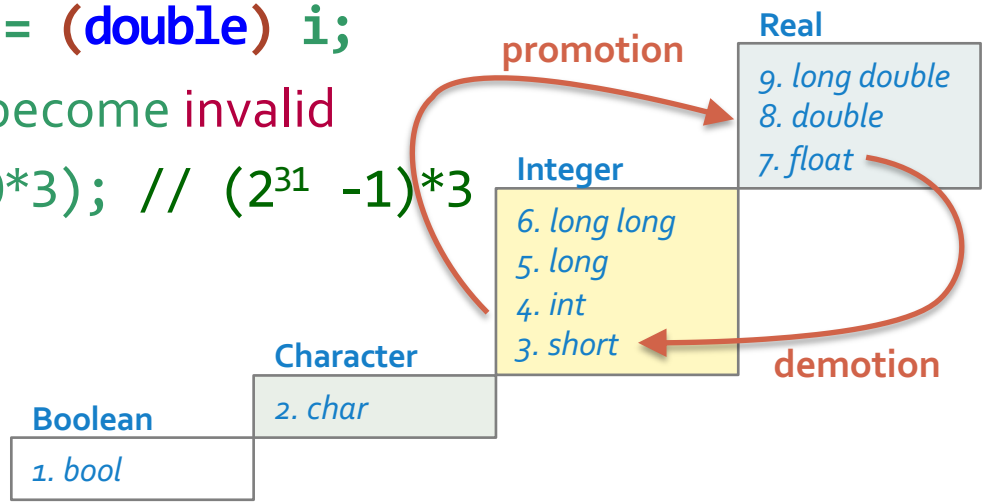
Type Conversion

- **Implicit** type conversion
  - binary expressions (e.g. `x + y`): lower-ranked operand is promoted to higher-ranked operand `// int x = 1; double y = 2.2;`
  - assignment (e.g. `x = y`): right operand is *promoted*/*demoted* to match the variable type on the left,
    - e.g., `int x = 1.8; // x will be integer 1`
- **Explicit** type conversion (type-casting)

  example: `int i = 10; double j = (double) i;`

  Demoted values might **change** or become invalid

  e.g., `int b = (int)(2147483647.0*3); // (2^31 -1)*3`



Constants

- Everything we covered before for variables apply for constants
  - type, name, scope
- Declaration format:

```
data_type  variable/constant identifier = value;
const data_type  variable/constant identifier = value;
```

- Examples:

```
const float pi = 3.14159;
const int maxValue = 500;
const char initial = 'D';
const char student_name[] = "John Chan";
```

[Optional] **Scope and namespace**

- A scope can be defined in many ways: by **{}**, **function**s, **class**es, and **namespace**s
- 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

[Optional] **Scope and namespace**

```
int a = 90; // this a is defined in global namespace
namespace level1 {
     int a = 0;
  namespace level2 {
    int a = 1;
  }
}
```

[Optional] **Resolving scope**

- Inside the main function, we can then resolve the variable's scope

```
// :: resolves to global namespace
cout << ::a << "\n";
cout << level1::a << "\n";
cout << level1::level2::a << "\n";
```

Summary

- Syntax
  - identifiers
  - reserved words
  - constants
  - operators
  - punctuators
- Variables & Constants
  - data types
    - numerical, character, logical
  - data type conversion
- Scope & Namespace (Optional)