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 – Tokens

- Tokens can be categorized into:
  - keywords
    - e.g., **main**, **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 **,**

Keywords

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				

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
  - Invalid identifiers: 3D_Point, my-income
- Always use meaningful names for identifiers
  - Bad examples: x, xx, a, b, temp, temp1, …

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

- Variables and constants must be declared before use
- Format:
  data_type *variable/constant identifier*;
- Examples:
  ```
  int age;
  int age1, age2;
  ```
  Their values are undefined at this point
- 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 Scope – Local vs. Global

- Scope of a variable refers to the accessibility/visibility boundary of a variable
  - We need to be able to "see" a variable in order to access it
- **Local** variables
  - Declared in a code block {} and can be only accessed within the code block
  - Try to access a local variable outside the block will produce unpredictable result

Declaration – Variable (Local)

```
#include <iostream>
using namespace std;
int main() { /* Any code after this line can access these local variables */
    int number1=12;
    int number2=3;
    int result;
    result = number1 + number2;
    return 0;
} /* Any code after this line cannot access these local variables */
int result; // This is legal, though meaningless...
result = number1 + number 2; // This is illegal
```


Global Variables

- **Global** variable
  - Defined in the global declaration sections of a program, i.e. defined outside a function block
  - Can be seen and accessed by all functions

Declaration – Variable (Global)

```
#include <iostream>
using namespace std;
int number1=12;
int number2=3;
int result;
/* Any code after this line can access these global variables */
int main() {
    result = number1 + number2; // Legal
    return 0;
}
int result;     // Illegal, identifier not unique
```


Mixed Use of Local and Global Variables

```
#include <iostream>
using namespace std;
int number1=12;
int number2=3;
/* Any code after this line can access these global variables */
int main() {
   int result;
   result = number1+number2;
   return 0;
} /* You can't access "result" beyond this line */
```

global

local

Global and Local Variables

- What if a global variable and a local one have the same name?

```
int x = 1;

int main() {
   int x = 0;
   cout << "The value of the variable is " << x << ".\n";
   return 0;
}
```


Global and Local Variables

- The local variable makes the global variable with the same name out-of-scope inside the function – it "hides" the global variable
- The same applies to local variables with different scopes

```
int x = 0;
cout << x << "\n";
{
   int x = 10;
   cout << x << "\n";
}
```

Global Variables are BAD

- Since every function/object has access to global variables, it becomes difficult to figure out who actually read and write these variables
- Even more difficult when you have millions of lines of code
- A recommended practice: minimize variable scope

C++ predefined data types

- Numerical
  - **int**: Integers (1, 3, 8 , 3222, 421, 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 b;
```

- 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
- 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.

[Optional] 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 large positive integer.
- The range of an **unsigned int** is from **0** to $2^{32}-1$

Floating Point Types

- **float**, **double** and **long double** are used to represent real numbers using the floating point representation.

```
double weight = 120.82;
```

- **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 and 3.367e-4 (1.23×10^2, and 3.367×10^{-4})

```
double weight = 1.23e2;
```


Data Type **char**

- Every character is represented by a code
  - American Standard Code for Information Interchange (ASCII)
- Used to store a **single** ASCII character, enclosed by the **single** quotation mark

```
char c = 'a';
char c = '\n';
```

- Characters are (***almost the same as***) integers
- Characters are treated as small integers, and conversely, small integers can be treated as characters
- A character takes one byte
  - $2^8 = 256$, **small** integers
  - Internally, 'a' is stored as the following bit pattern 0 1 1 0 0 0 0 1
  - It is equivalent to an integer 97

char as integers

- Any integer expression can be applied to char type variables

```
char c = 'c';
c = c + 1;
cout << "variable c has the character " << c;
```

  - The output is "variable c has the character d"

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	□	┌	⊥	┘	↘	⊠	✓	⍾	↶	➢	≡	⩛	⤋	⩚	⊗	⊙	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	⊟	◷	◶	◵	◴	✓	⎍	⊣	⧖	↕	ʕ	⊖	◰	◱	◲	◳	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

Strings (**cstring**)

- 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 name[] = "John Doe";
```

- Remember escape sequences?

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

- To extend a string beyond one line, use backslash ****

```
char name[] = "Alex looooooong \
      John";
```

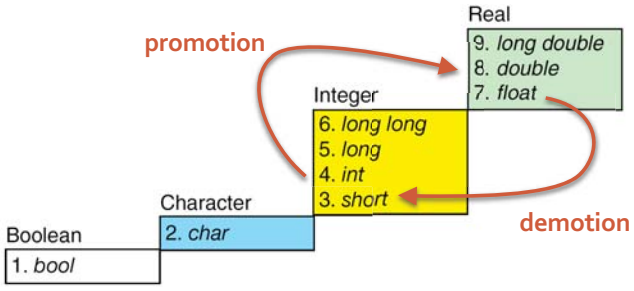

Type Conversion

- A **char** can be used in any expression where an **int** may be used
- Arithmetic conversions occur if necessary as the operands of a binary operator are evaluated (see the next few slides)

Type Conversion

- **Implicit** type conversion
  - binary expressions (e.g. **x + y**): lower-ranked operand is promoted to higher-ranked operand
  - assignment (e.g. **x = y**): right operand is promoted/demoted to match the variable type on the left
- **Explicit** type conversion (type-casting)

  example: **int i = 10; double j = (double) i;**
  - Demoted values might change or become invalid



Constants

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

  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";
```

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 len1, len2;
float x;
len1 = sizeof(int);
len2 = sizeof(x);
```


[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 resolving 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";
```