Chapter 8

Arithmetic in C++

8.1 The C++ Vocabulary

8.2 Variables and Types
  8.2.1 Data Objects
  8.2.2 Variables
  8.2.3 Declaration of Variables

8.3 Elementary C++ Data Types
  8.3.1 Integers
  8.3.2 Floating Point Numbers
  8.3.3 Type Conversions in Expressions
  8.3.4 Qualifiers for Variable Declarations
8.4 Arithmetic Expressions
   8.4.1 Standard Arithmetic Operators
   8.4.2 Assignments and Compound Assignments
   8.4.3 Operator Priorities

8.5 Symbolic Constants
   8.5.1 Declaration of Symbolic Constants
   8.5.2 Hexadecimal and Octal Constants

8.6 The char Data Type

8.7 References
### 8.1 The C++ Vocabulary

<table>
<thead>
<tr>
<th>asm</th>
<th>do</th>
<th>if</th>
<th>reinterpret_cast</th>
<th>try</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>double</td>
<td>inline</td>
<td>return</td>
<td>typedef</td>
</tr>
<tr>
<td>bool</td>
<td>dynamic_cast</td>
<td>int</td>
<td>short</td>
<td>typeid</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>signed</td>
<td>typename</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>main</td>
<td>sizeof</td>
<td>union</td>
</tr>
<tr>
<td>catch</td>
<td>explicit</td>
<td>mutable</td>
<td>static</td>
<td>unsigned</td>
</tr>
<tr>
<td>char</td>
<td>export</td>
<td>namespace</td>
<td>static_cast</td>
<td>using</td>
</tr>
<tr>
<td>class</td>
<td>extern</td>
<td>new</td>
<td>struct</td>
<td>virtual</td>
</tr>
<tr>
<td>const</td>
<td>false</td>
<td>operator</td>
<td>switch</td>
<td>void</td>
</tr>
<tr>
<td>const_cast</td>
<td>float</td>
<td>private</td>
<td>template</td>
<td>volatile</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>protected</td>
<td>this</td>
<td>wchar_t</td>
</tr>
<tr>
<td>default</td>
<td>friend</td>
<td>public</td>
<td>throw</td>
<td>while</td>
</tr>
<tr>
<td>delete</td>
<td>goto</td>
<td>register</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

*types — type qualifiers — execution control structures*
8.2 Variables and Types

8.2.1 Data Objects

The **operational model** of computing / programming considers a program as

a sequence of instructions for **manipulating data objects**.

Data objects can be distinguished by their **reference levels**:

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Constant</td>
<td>an actual value (a number, a character, ...)</td>
</tr>
<tr>
<td>1</td>
<td>Variable</td>
<td>contains a value (= constant)</td>
</tr>
<tr>
<td>2</td>
<td>Level-1 pointer</td>
<td>refers to a variable, points to a variable</td>
</tr>
<tr>
<td>3</td>
<td>Level-2 pointer</td>
<td>refers to a level-1 pointer</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Constant:

| Const_Name | | internal object |

### Example:

| 3.14159 | | internal representation |

### Variable:

| Name | | reference (address) |

| Name | | internal representation |

A data object has...
- a name
- an internal object
  (determined by its data type)
8.2.2 Variables

Variables are data objects with the following properties:

- **Identifier, name**
  - Defined in its declaration (example: `int size = 12`)

- **Type**
  - Defined in its declaration (example: `int size = 12`)

- **Internal object** = address of the manipulable memory section (lvalue)

- **Value** = alterable contents of the addressed memory section
  - Its interpretation depends on its type.
  - Its value can be changed by assignments (=) or specific operators (++)

- **Lifespan:**
  - Dynamically defined by the program structure (see: functions).
8.2.3 Declaration of Variables

\[ \text{basic\_variable\_declaration} ::= \text{type} \{\text{name,}\}^* \text{name}; \]

bool correct_input;  // true or false
int number_of_items; // number of items
float width, height; // width and height

Initializing Variables

\[ \text{basic\_variable\_declaration} ::= \text{type} \{\text{name\_init,}\}^* \text{name\_init}; \]

\[ \text{name\_init} ::= \{\text{name} | \text{name} (\text{init}) | \text{name} = \text{init}\} \]

int counter(0);  // C++ initialization
int counter = 0; // older C style syntax
8.3 Elementary C++ Data Types

- **Integers**
  
  ```
  int normal_integer;
  long int long_integer;
  short int short_integer;
  unsigned short int unsigned_short_int;
  ```

- **Floating Point Numbers**
  
  ```
  float normal_float;
  double double_float;
  long double high_precision_float;
  ```

- **Characters**
  
  ```
  char single_character;
  ```
8.3.1 Integers

Integers (= whole numbers) have no fractional part or decimal point.

Range of values

<table>
<thead>
<tr>
<th>Integers</th>
<th>Type</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int, short int</td>
<td>-2^31 to 2^31 - 1</td>
</tr>
<tr>
<td>long int</td>
<td>long int</td>
<td>-2^15 to 2^15 - 1</td>
</tr>
<tr>
<td>short int</td>
<td>short int</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td>-32,768 to 32,767</td>
</tr>
</tbody>
</table>

Types of Integers

- **int**: the most efficient size for the machine is chosen (2 or 4 bytes)
- **long int**: extra storage for integer (4 bytes guaranteed)
- **short int**: reduced storage (2 bytes)
- **unsigned**: uses all bits for the number (for 1 byte: range 0 to 65,535)
8.3.2 Floating Point Numbers

Floating point numbers have a decimal point or mantissa and exponent specification.

- 3.1419
- 0.5
- 1.2e34 (\(= 1.2 \times 10^{34}\))

Range of values

The range of floating point numbers (which are always signed) and their accuracy varies widely among different computer and operating systems.\(^1\)

Types of Floats

- **float**: normal precision (usually 4 bytes = 32 bits)
- **double**: twice the range and precision of float (usually 8 bytes = 64 bits)
- **long double**: extended precision (at least 8 bytes)
8.3.3 Type Conversions in Expressions

If an expression (see chapter 8.4) contains operands of different types, an **automatic type conversion** (to the type which is highest in the following hierarchy) is performed.

```
  double
    ↘
    float
      ↘
      long int
        ↘
        int
          ↘
          char
          ↘
          short int
```

This diagram is a simplified version of all possible type conversions.
Examples of binary operators and implicit type conversions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Integer example</th>
<th>Floating-point example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>5 + 2 \Rightarrow 7</td>
<td>5.5 + 2.2 \Rightarrow 7.7</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>5 - 2 \Rightarrow 3</td>
<td>5.5 - 2.2 \Rightarrow 3.3</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>5 \times 2 \Rightarrow 10</td>
<td>5.5 \times 2.2 \Rightarrow 12.1</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>5 / 2 \Rightarrow 2</td>
<td>5.5 / 2.2 \Rightarrow 1.375</td>
</tr>
<tr>
<td>%</td>
<td>Modulus, or remainder</td>
<td>5 % 2 \Rightarrow 1</td>
<td>5.5 % 2.2 \Rightarrow \text{type error}</td>
</tr>
</tbody>
</table>
### Chapter 8: Arithmetic in C++

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Integer / floating-point example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>$5 + 2.2 \Rightarrow 7.7$</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>$5.0 - 2 \Rightarrow 3.0$</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>$4 \times 2.2 \Rightarrow 8.8$</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>$6.0 / 2 \Rightarrow 3.0$</td>
</tr>
<tr>
<td>%</td>
<td>Modulus, or remainder</td>
<td>$5.0 % 2.0 \Rightarrow \text{type error}$</td>
</tr>
</tbody>
</table>
8.3.4 Qualifiers for Variable Declarations

```
variable_declaration ::= [ special ] [ class ] [ size ] [ sign ] type
{ name_init, }* name_init;
```

<table>
<thead>
<tr>
<th>Special</th>
<th>Class</th>
<th>Size</th>
<th>Sign</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile</td>
<td>register</td>
<td>short</td>
<td>signed</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>static</td>
<td>long</td>
<td>unsigned</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>extern</td>
<td>&lt;blank&gt;</td>
<td>&lt;blank&gt;</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>auto</td>
<td>&lt;blank&gt;</td>
<td>&lt;blank&gt;</td>
<td>(bool)</td>
</tr>
<tr>
<td></td>
<td>&lt;blank&gt;</td>
<td>&lt;blank&gt;</td>
<td>&lt;blank&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Special:

- volatile: special variable whose value may change at any time
- <blank>: normal variable

Class:

- register: frequently used variable, kept in a machine register
- static: meaning depends on context
- extern: defined in another file
- auto: variable allocated from the stack
- <blank>: auto default class is selected

Size:

- long: larger than normal integer or very large float
- short: smaller than normal integer
- <blank>: normal size number (integer or float)
Sign:

- signed: signed integer or character
- unsigned: use all integer or character bits (no sign bit)

Type:

- int: integer
- float: floating point number
- double: double-size float
- char: single character, also used for very short integer
- bool: Boolean value, true or false
8.4 Arithmetic Expressions

**expression**

- simple value
- function call
- unary expression
- binary expression
- ...

**simple value**

- number constant
- string constant
- character constant
- identifier (declared)
- ...

© Christian Jacob
unary operator

binary operator

For assignments the simple value must be an identifier!
8.4.1 Standard Arithmetic Operators

**Arithmetics**

\[
\begin{align*}
\text{op} & \quad \text{int} \Rightarrow \text{int} & \quad +, - \\
\text{op} & \quad \text{float} \Rightarrow \text{float} & \quad +, - \\
\text{int} & \quad \text{op} \quad \text{int} & \Rightarrow \text{int} & \quad +, -, *, /, \% \\
\text{float} & \quad \text{op} \quad \text{float} & \Rightarrow \text{float} & \quad +, -, *, / \\
\end{align*}
\]

**Comparisons**

\[
\begin{align*}
\text{int} & \quad \text{op} \quad \text{int} & \Rightarrow \text{int} & \quad ==, !=, <=, <, >, >= \\
\text{float} & \quad \text{op} \quad \text{float} & \Rightarrow \text{float} & \quad ==, !=, <=, <, >, >= \\
\end{align*}
\]
Chapter 8: Arithmetic in C++

Logical Operators

\[
\text{int } \text{op} \text{ int } \Rightarrow \text{ int}
\]

\[
\text{op} \quad \text{int} \quad \Rightarrow \quad \text{int} \quad ! \quad (\text{not})
\]

\[
\text{int } \text{op} \text{ int } \Rightarrow \text{ int} \quad \& \& \quad (\text{and}), \quad | \quad (\text{or})
\]

Bitwise Logical Operators

\[
\text{op} \quad \text{int} \quad \Rightarrow \quad \text{int}
\]

\[
\text{op} \quad \text{int} \quad \Rightarrow \quad \text{int} \quad \sim \quad (\text{not})
\]

\[
\text{int } \text{op} \text{ int } \Rightarrow \text{ int} \quad \& \quad (\text{and}), \quad | \quad (\text{or}), \quad ^{\text{xor}},
\]

\[
>> \quad (\text{right shift}), \quad << \quad (\text{left shift})
\]

\[
\begin{array}{cccccccc}
\text{x} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\text{~x} & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{x} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\text{x << 2} & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]
The ? Operator

(condition ? true_expression : false_expression)

Example:

```cpp
int x = 15;
cout << (x < 7 ? 4 : 2); // Output: 2
```
The Comma Operator: Concatenation of Two Expressions

deadline 1, deadline 2

evaluates deadline 1 and then deadline 2, and finally returns the value of deadline 2 as the result.

Example:

```c++
int x = 0, y = 1;
cout << (x = x + y, y = y * 2);
```

More useful in loops (see later):

```c++
for (i=1, j=9; ...; i = i+2, j = j-3)
{ ... // loop body }
```
8.4.2 Assignments and Compound Assignments

Syntax Diagram for Assignments

```
int pounds = 0;
float kilos = pounds / 2.2;

int x, y, z; int f(int n);
...
x = -y * (z++ - f(4));
```
Compound Assignments

A common operation in programming is to apply an operator to a variable, and then store the result in the same variable.

For example, the following assignment doubles the value of j:

\[ j = j \times 2; \]

This can be rewritten as:

\[ j *= 2; \]

This works for the following operators in C++:

\[ +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |= \]
Special Case: Increments and Decrements by One

For increasing or decreasing a variable value by 1, there is even a shorter notation in C++.

Instead of writing

\[ k = k + 1; \]

one can write

\[ k++; // \text{return value of } k, \text{ then increment} \]

or

\[ ++k; // \text{increment, then return value of } k \]

This also works for the decrement operator ‘--’:

\[ k--; // \text{or } --k; \text{ equivalent to } k = k - 1; \]
Examples:

\[
\begin{align*}
a &= 1; & // a &= 1 \\
b &= ++a; & // a &= 2, \ b &= 2 \\
c &= a++; & // a &= 3, \ c &= 2 \\
\end{align*}
\]

\[
\begin{align*}
a &= 5; \ b &= 3; \\
n &= ++a + b--; & // a &= 6, \ b &= 2, \ n &= 9 \\
\end{align*}
\]

\[
\begin{align*}
a &= 5; \ b &= 3; \\
n &= ++a \times ++b; & // a &= 6, \ b &= 4, \ n &= 24 \\
n &= a++ \times b++; & // a &= 6, \ b &= 4, \ n &= 15 \\
\end{align*}
\]

\[
\begin{align*}
a &= 5; \ b &= 3; \\
n &= a++ \times --b; & // a &= 6, \ b &= 2, \ n &= 10 \\
\end{align*}
\]
### 8.4.3 Operator Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operators</th>
<th>Ass.</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>! ~ ++ -- + -</td>
<td>⇐</td>
<td><strong>right</strong> to left</td>
</tr>
<tr>
<td></td>
<td>* / %</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt; &gt;&gt;</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>== !=</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>⇒</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>? :</td>
<td>⇐</td>
<td><strong>right</strong> to left</td>
</tr>
<tr>
<td>low</td>
<td>,</td>
<td>⇒</td>
<td><strong>left</strong> to right</td>
</tr>
</tbody>
</table>
8.5 Symbolic Constants

8.5.1 Declaration of Symbolic Constants

constant_declaration ::= const type {name_init,}*name_init;

const float PI = 3.1415926; // magic Pi

The values of constants can not be changed by further assignments:

PI = 3.0  // Doesn’t work!
8.5.2 Hexadecimal and Octal Constants

The C++ language has conventions for representing octal and hexadecimal values:

- **Octal** number: leading zeros
- **Hexadecimal** number: leading “0x” (zero + ‘x’):

**Examples:**

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 8</th>
<th>Base 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>06</td>
<td>0x6</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>0x9</td>
</tr>
<tr>
<td>15</td>
<td>017</td>
<td>0xF</td>
</tr>
</tbody>
</table>
8.6 The char Data Type

The type char represents a single character, enclosed in single quotation marks (‘A’, ‘a’, ‘!’, ‘\’).

```cpp
cchar capitalA = 'A';
cchar smallB = 'b';
cchar lookOutChar = '!';
cchar capitalB = 66; cout << capitalB;
```

The backslash character (\) is called the escape character, signalling that a special character follows.

```cpp
cchar backspace = '\b';
cchar newline = '\n';
cchar backslash = '\\';
cchar quote = '\''
cchar doubleQuote = '"'
```
## A small selection of special characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>Move the cursor one character to the left</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
<td>Go to the top of a new page</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td>Go to the next line</td>
</tr>
<tr>
<td>\r</td>
<td>Return</td>
<td>Go to the beginning of the current line</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
<td>Advance to the next tab stop</td>
</tr>
<tr>
<td>\</td>
<td>Single quote</td>
<td>The character ‘</td>
</tr>
<tr>
<td>\</td>
<td>Double quote</td>
<td>The character “</td>
</tr>
<tr>
<td>\nnn</td>
<td>The character with ASCII code nnn</td>
<td>The ASCII character with number nnn (octal)</td>
</tr>
<tr>
<td>\NN</td>
<td>The character with ASCII code NN</td>
<td>The ASCII character with number NN (hexadecimal)</td>
</tr>
</tbody>
</table>
8.7 References