4a. Numeric data types

**Topic: Numeric data types**
- The int data type
- Arithmetic expressions and operators
- Smaller and larger-capacity integer types
- Using floating-point data
- Standard numeric constants and functions
- Type conversions
- Formatting output

**The data type int**
- Data type: a category that defines the storage and meaning or interpretation given to a pattern of bits
- int is a signed integer type
- Usual range of values: \(-2^G \ldots 2^G\) (32 bits: \(2^{32} = 4G\))
- 2586 is an expression of type int
- After int x = 9; x is an expression of type int

**Arithmetic operators**
- Binary operators have 2 operands:
  - + -- * /
    (Note: division of integers produces an int value; division by 0 is undefined)
  - % modulo or remainder operator; e.g.:
    - 7 % 3 = 1
    - The clock time 3 hours after 11:00 is (11 + 3) % 12
    - The clock time h hours after time t is (t + h) % 12, except a result of 0 should change to 12
  - Unary negation: \(-2\) \(-9\) \(-5 + 9\)

**Converting cents to dollars**
```
cout << "Cents: ";
int cents;
cin >> cents;
int dollars = cents / 100;
int change = cents % 100;
cout << dollars << " dollars, " << change << " cents" << endl;
```
- integer division yields an integer
- the modulo operator can truncate a number

**Math notation and C/C++**
```
\[
\begin{align*}
\frac{a + 1}{b - 2} & \quad \text{C/C++ } (a+1) / (b-2) \\
(\text{a}^2) & \quad \text{C/C++ a * a or pow(a, 2)} \\
\text{c modulo d} & \quad \text{C/C++ c % d} \\
\sqrt{2} & \quad \text{C/C++ sqrt(2)}
\end{align*}
\]
```

**Extended assignment operators**
- \(x += y\) means \(x = x + y\)
- \(x -= y\) means \(x = x - y\)
- \(x *= y\) means \(x = x * y\)
- \(x /= y\) means \(x = x / y\)
- \(x %= y\) means \(x = x \% y\)
- \(++x\) or \(x++\) means \(x = x + 1\)
- \(--x\) or \(x--\) means \(x = x - 1\)
4a. Numeric data types

Pre- and post-increment operators

```c
int n = 4;
cout << "n++ = " << n++ << endl
<< "now n = " << n << endl
<< "++n = " << ++n << endl
<< "n = " << n << endl;
```

- Both operators add 1 to value of variable
- Pre-increment operates before evaluation

Operator precedence

- Parenthesized operations come first
- Unary minus has high precedence
- Multiplication and division precede addition and subtraction
- Operations of same precedence proceed left to right

```c
Examples:
8 - 2 + 5 8 - (2 + 5)
3 * 2 + 4 3 * (2 + 4) 3 + 2 * 4
3 + 6 / 2 (3 + 6) / 2 3 - 2 + 3
1 + 3 % 2 25 % 5 * 2
```

Small and large integer types

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage (bytes)</th>
<th>Min. value*</th>
<th>Max. value*</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2G</td>
<td>+2G</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>0</td>
<td>+4G</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>-32K</td>
<td>+32K</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>-2G</td>
<td>+2G</td>
</tr>
</tbody>
</table>

- Type qualifiers: short, long, unsigned, const
- Use the `sizeof` operator; e.g.: `cout << sizeof (char);`

Data types `float and double`

- May store fractional values and a great range of numbers
- Storage: sign bit, fraction, exponent, based on scientific-notation concept
- `float` occupies 32 bits in MSVC 5.0 implementation
- Floating-point storage entails representational error
- Qualifier `double` extends precision (64 bits in MSVC)

Using `double`

```c
cout << "Enter 2 #s: ";
double a,b;
cin >> a >> b;
double quotient = a / b;
cout << "quotient = "
<< quotient << endl;
```

- Result of dividing floating-point values is of floating-point type

Floating-point data

- Data types `float` and `double` represent numbers with possible fraction parts
- The numeral floats because the exponent part compensates for a shift to eliminate 0’s on the left
- `double` has twice the precision of `float`
### Standard numeric identifiers
- Header file `math.h` (#include `<math.h>`):
  - absolute value: `abs`, `fabs`
  - trigonometry: `cos`, `acos`, `sin`, `asin`, `tan`, `atan`
  - other: `exp`, `log`, `pow`, `sqrt`
- Example:
  ```cpp
  void main()
  {
    cout << abs(-3.5) << endl
    << fabs(-3.5) << endl
    << sqrt(2) << endl
    << log(16) / log(2) << endl
    << cos(3.1416 / 360) << endl;
  }
  ```
  [math.cpp]
  Output:
  3
  3.5
  1.41421
  4
  81
  0.999962

### Numeric type conversions
- Automatic: occurs by promotion or truncation. Examples:
  ```cpp
  int y = 4.3; cout << y;
  double z = 2;
  ```
- High types to low: `double`, `float`, `long`, `int`, `short`, `char`
- Type casts coerce types.
  E.g., `(double(1) / 2)` yields 0.5
  ```cpp
  cout << (int) 4.2;
  ```
  outputs 4
- Danger: overflow (watch warnings)

### Type casts and integer division
```cpp
void main()
{
  int a=5,b=3;
  cout << a " / " << b << " = " << a / b << endl;
  cout << "double(" << a " ) / " << b << " = " << double(a) / b << endl;
}
```

---
**Output:**
5 / 3 = 1  
double(5) / 3 = 1.66667

- Without type cast, integer division occurs here, yielding integer result
- Type casting forces a data item to have a specified type

### C++ promotes type for compatibility
- Lower type is promoted to higher
- High to low: `double`, `float`, `long`, `int`, `short`, `char`
- Examples:
  ```cpp
  double f = 2;
  double sum = 1 + 2.2;
  ```
  ```cpp
  (3.6 * 4)  
  ```
  is `double`

### Integer overflow
```cpp
void main()
{
  short int product = 2 * 20000;
  cout << "2 * 20000 = " << product << endl;
}
```

---
**Output:**
2 * 20000 = -25536

- Overflow occurs when a value assigned exceeds the capacity of a data type

### Operator overloading
- In C and C++, the meaning of some operators depends on the context
- Examples: `=`, `+`, `-`, `*`, `/`, `<<`, `>>`
- Recall that in machine language, different instructions are used for operations on different types of data
Manipulators

- Library: iomanip.h
- Used to format stream output
- A manipulator changes the state of an output stream object
- Areas of control: field width, justification, precision
- Manipulator functions: setw, setprecision, setiosflags

Manipulator **setw**: example

```cpp
#include <iostream.h>
#include <iomanip.h>

const int FW = 6; // field width
void main()
{
    int n1 = 32, n2 = 110, n3 = 7;
    cout << setw(FW) << n1 << endl
    << setw(FW) << n2 << endl
    << setw(FW) << n3 << endl;
}
```

Output:

```
3 2
1 1 0
7
```

Setting width and precision of fractions

```cpp
#include <iostream.h>
#include <iomanip.h>

void main()
{
    double f = 18.558;
    cout << setprecision(2)
    << setiosflags(ios::showpoint | ios::fixed)
    << setw(6) << f << endl;
}
```

Output:

```
18.56
```

Discussion problems

- Write a program that prompts for a unit price, discount rate, and quantity and displays a subtotal without tax; a tax amount, given a constant tax rate of 5%; and a total amount due. Format to two places.
- Write a program that prompts for the height and diameter of an oil drum and displays the area of its surface, given that the area of a circle is \( \pi r^2 \) and \( \pi = 3.14159 \). Format to two places.

Formatting tabular output

```cpp
char name1[80] = "Wang",
    name2[80] = "Fuentes";
int sal1 = 25000, sal2 = 37820;

cout << setw(12) << "Name" << setw(8)
    << "Salary" << endl;
cout << setw(12) << name1 <<
    setw(8) << sal1 << endl;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang</td>
<td>25000</td>
</tr>
<tr>
<td>Fuentes</td>
<td>37820</td>
</tr>
</tbody>
</table>

Using field width and precision

```cpp
char name[80] = "book";
double price = 25.95, qty = 37;

cout << setprecision(2)
    << setiosflags(ios::showpoint | ios::fixed)
    << setw(12) << "Name" << "Price" <<"Qty" << endl;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td>25.95</td>
<td>37</td>
</tr>
</tbody>
</table>