C++ Programming

Week 3:

Operators and Flow Control

Dr. Owen Chen
Cary Chinese School
Director of Math and Computer Science

2023 Summer

Week 3: Agenda

- Continue C++ Basic Types (integer, char, bool, float)
- C++ operators
- Flow Control

Variables and Basic Types

int

- **int is the most frequently used integer type**

```
int i; //declare a variable
int j = 10; //declare and initialize
int k;
k = 20; //assign a value
```

- **Remember to initialize a variable!**
- **Will the compiler give an error?**

```
int i;
cout << i; //what is i's value?
```

How to initialize a variable

```
int num;
num = 10;//do not forget this line

int num = 10;

int num (10);

int num {10};
```

Arithmetic Types

Type	Bytes	Range	Fixed width types
bool	1	true, false	
char †	1	-127 to 127	
signed char	1	-128 to 127	int8_t
unsigned char	1	0 to 255	uint8_t
short	2	-2^{15} to $2^{15}-1$	int16_t
unsigned short	2	0 to $2^{16}-1$	uint16_t
int	4	-2^{31} to $2^{31}-1$	int32_t
unsigned int	4	0 to $2^{32}-1$	uint32_t
long int	4/8		int32_t/int64_t
long unsigned int	4/8*		uint32_t/uint64_t
long long int	8	-2^{63} to $2^{63}-1$	int64_t
long long unsigned int	8	0 to $2^{64}-1$	uint64_t
float (IEEE 754)	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	
double (IEEE 754)	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{+308}$	

* 4 bytes on Windows64 systems, † one-complement

Arithmetic Types - Short Name

Signed Type	short name
signed char	/
signed short int	short
signed int	int
signed long int	long
signed long long int	long long

Unsigned Type	short name
unsigned char	/
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long
unsigned long long int	unsigned long long

Arithmetic Types - Suffix and Prefix

Type	SUFFIX	example
int	/	2
unsigned int	u	3u
long int	l	8l
long unsigned	ul	2ul
long long int	ll	4ll
long long unsigned int	ull	7ull
float	f	3.0f
double		3.0

Representation	PREFIX	example
Binary C++14	0b	0b010101
Octal	0	0308
Hexadecimal	0x or 0X	0xFFA010

C++14 allows also *digit separators* for improving the readability

`1'000'000`

int*_t <cstdint>

C++ provides fixed width integer types.

They have the same size on any architecture:

int8_t, uint8_t
int16_t, uint16_t
int32_t, uint32_t
int64_t, uint64_t

Good practice: Prefer fixed-width integers instead of native types. `int` and `unsigned` can be directly used as they are widely accepted by C++ data models `int*_t` types are <u>not</u> "real" types, they are merely *typedefs* to appropriate fundamental types

C++ standard does not ensure a one-to-one mapping:

- There are **five** distinct *fundamental types* (`char`, `short`, `int`, `long`, `long long`)
- There are **four** `int*_t` *overloads* (`int8_t`, `int16_t`, `int32_t`, and `int64_t`)

ithare.com/c-on-using-int_t-as-overload-and-template-parameters

Warning: I/O Stream interprets `uint8_t` and `int8_t` as `char` and not as integer values

```
int8_t var;
cin >> var;  // read '2'
cout << var; // print '2'
int a = var * 2;
cout << a;   // print '100' !!
```

char

- `char`: **type for character, 8-bit integer indeed!**

- `signed char`: **signed 8-bit integer**
- `unsinged char`: **unsigned 8-bit integer**
- `char`: **either** `signed char` **or** `unsinged char`

Integers and characters

char.cpp

How we represent a character?

- Use an 8-bit integer

```
char c1 = 'C’;  //its ASCII code is 80
char c2 = 80;  //in decimal
char c3 = 0x50; //in hexadecimal
```

Chinese characters?

```
char16_t   c = u'于'; //c++11
char32_t   c = U'于'; //c++11
```

bool

A C++ keyword, but not a C keyword
bool width: 1 byte (8 bits), NOT 1 bit!
Value: true **(1) or** false **(0)**

What is the output?

bool.cpp

```
bool b = true;
int i = b;
cout << "i=" << i << endl;
cout << "b=" << b << endl;
```

bool

Boolean data conversion

```
bool b = true;
int i = b; // the value of i is 1.

bool b = -256; //  not recommended - the value of b is true

bool b = (-256 != 0); // better choice
```

size_t

Computer memory keeps increasing
32-bit int used to be large enough
But it is not sufficient now.

`size_t`:

- Unsigned integer
- Type of the result of `sizeof` operator
- Can store the maximum size of a theoretically possible object of any type
- 32-bit, or 64-bit

`size_t`

`size_t` <cstddef>

`size_t` is an *alias* data type capable of storing the biggest representable value on the current architecture

- `size_t` is an unsigned integer type (of at least 16-bit)
- In common C++ implementations:
  - `size_t` is 4 bytes on 32-bit architectures
  - `size_t` is 8 bytes on 64-bit architectures
- `size_t` is commonly used to represent size measures

Arithmetic Type Limits

Query properties of arithmetic types in C++11:

```
#include <limits>

std::numeric_limits<int>::max();      // 2^31 - 1
std::numeric_limits<uint16_t>::max(); // 65,535

std::numeric_limits<int>::min();      // -2^31
std::numeric_limits<unsigned>::min(); // 0
```

* this syntax will be explained in the next lectures

Promotion and Truncation

Promotion to a larger type keeps the sign

```
int16_t x = -1;
int     y = x; // sign extend
cout << y;     // print -1
```

Truncation to a smaller type is implemented as a modulo operation with respect to the number of bits of the smaller type

```
int     x = 65537; // 2^16 + 1
int16_t y = x;     // x %2^16
cout << y;         // print 1

int     z = 32769; // 2^15 + 1 (does not fit in a int16_t)
int16_t w = z;     // (int16_t) (x %2^16 = 32769)
cout << w;         // print -32767
```

Fixed width integer types (since C++11)

Defined in <cstdint>

```
int8_t
int16_t
int32_t
int64_t
uint8_t
uint16_t
uint32_t
uint64_t
...
```

Some useful macros

```
INT8_MIN
INT16_MIN
INT32_MIN
INT64_MIN
INT8_MAX
INT16_MAX
INT32_MAX
INT64_MAX
...
```

intmax.cpp

```cpp
#include <iostream>
#include <cstdint>
using namespace std;
int main()
{
    cout << "INT8_MAX=" << INT8_MAX << endl;
}
```

Floating-point Types and Arithmetic

Floating-point Numbers

How many numbers in range [0, 1]?

Infinite!

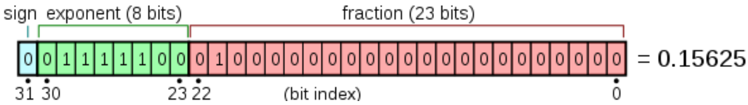
How many numbers can 32 bits represent?

2^{32}

You want 1.2, but `float` can only provide you 1.200000047683716...

Floating-point types

- `float`: **single precision floating-point type, 32 bits**



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2-127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

- `double`: **double precision floating-point type, 64 bits**
- `long double`: **extended precision floating-point type**
    - 128 bits if supported
    - 64 bits otherwise
- **half precision floating-point, 16 bits**
  (popular in deep learning, but not a C++ standard)

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Precision

- **Will f2 be greater than f1?**

precision.cpp

```
float f1 = 23400000000;
float f2 = f1 + 10; // but f2 = f1
```

- **Why?**

- **Can we use == operator to compare two floating point numbers?**

```
if (f1 == f2) //bad
if (fabs(f1 - f2) < FLT_EPSILON) // good
```

32/64-bit Floating-Point

- **IEEE764 Single precision** (32-bit) `float`

Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	8-bit	23-bit

- **IEEE764 Double precision** (64-bit) `double`

Sign	**Exponent** (or base)	**Mantissa** (or significant)
1-bit	11-bit	52-bit

Floating-point number:

- *Radix* (or base): β
- *Precision* (or digits): p
- *Exponent* (magnitude): e
- *Mantissa*: M

$$n = \underbrace{M}_{p} \times \beta^e \quad \rightarrow \quad \text{IEEE754: } 1.M \times 2^e$$

```
float  f1 = 1.3f;    // 1.3
float  f2 = 1.1e2f;  // 1.1 ·10^2
float  f3 = 3.7E4f;  // 3.7 ·10^4
float  f4 = .3f;     // 0.3
double d1 = 1.3;     // without "f"
double d2 = 5E3;     // 5 ·10^3
```

```
cout << 0 / 0;           // undefined behavior
cout << 0.0 / 0.0;       // print "nan"
cout << 5.0 / 0.0;       // print "inf"
cout << -5.0 / 0.0;      // print "-inf"

auto inf = std::numeric_limits<float>::infinity;
cout << (-0.0 == 0.0);                        // true, 0 == 0
cout << ((5.0f / inf) == ((-5.0f / inf));  // true, 0 == 0
cout << (10e40f) == (10e40f + 9999999.0f); // true, inf == inf
cout << (10e40)  == (10e40f + 9999999.0f); // false, 10e40 != inf
```

Floating-point - Summary 1/3

- NaN (mantissa/= 0)

*	11111111	***********************

- ± infinity

*	11111111	00000000000000000000000

- Lowest/Largest ($\pm 3.40282 * 10^{+38}$)

*	11111110	11111111111111111111111

- Minimum (normal) ($\pm 1.17549 * 10^{-38}$)

*	00000001	00000000000000000000000

- Denormal number ($< 2^{-126}$)(minimum: $1.4 * 10^{-45}$)

*	00000000	***********************

- ± 0

*	00000000	00000000000000000000000

Floating-point - Summary

	bfloat16	float	double
Exponent	8-bit [0*-254]		11-bit [0*-2046]
Bias	127		1023
Mantissa	7-bit	23-bit	52-bit
Largest ($\pm$)	2^{128} $3.4 \cdot 10^{38}$		2^{1024} $1.8 \cdot 10^{308}$
Smallest ($\pm$)	2^{-126} $1.2 \cdot 10^{-38}$		2^{-1022} $2.2 \cdot 10^{-308}$
Smallest (denormal*)	/	2^{-149} $1.4 \cdot 10^{-45}$	2^{-1074} $4.9 \cdot 10^{-324}$
Epsilon	2^{-7} 0.0078	2^{-23} $1.2 \cdot 10^{-7}$	2^{-52} $2.2 \cdot 10^{-16}$

Floating-point - Limits

```
#include <limits>
// T: float or double

std::numeric_limits<T>::max();          // largest value

std::numeric_limits<T>::lowest();       // lowest value (C++11)

std::numeric_limits<T>::min();          // smallest value

std::numeric_limits<T>::denorm_min()   // smallest (denormal) value

std::numeric_limits<T>::epsilon();      // epsilon value

std::numeric_limits<T>::infinity()      // infinity

std::numeric_limits<T>::quiet_NaN()    // NaN
```

Floating-point - Useful Functions

```
#include <cmath> // C++11

bool std::isnan(T value)      // check if value is NaN
bool std::isinf(T value)      // check if value is ±infinity
bool std::isfinite(T value)   // check if value is not NaN
                              // and not ±infinity

bool std::isnormal(T value); // check if value is Normal

T    std::ldexp(T x, p)       // exponent shift x*2^p
int  std::ilogb(T value)      // extracts the exponent of value
```

Conversion Rules

Conversion Rules

Implicit type conversion rules, applied in order, before any operation:

⊗: any operation (*, +, /, -, %, etc.)

(A) Floating point promotion

floating_type ⊗ integer_type → floating_type

(B) Implicit integer promotion

small_integral_type := any signed/unsigned integral type smaller than `int`

small_integral_type ⊗ small_integral_type → `int`

(C) Size promotion

small_type ⊗ large_type → large_type

(D) Sign promotion

signed_type ⊗ unsigned_type → unsigned_type

Examples and Common Errors

```
float    f = 1.0f;
unsigned u = 2;
int      i = 3;
short    s = 4;
uint8_t  c = 5; // unsigned char

f * u; // float × unsigned → float: 2.0f
s * c; // short × unsigned char → int: 20
u * i; // unsigned × int → unsigned: 6u
+c;    // unsigned char → int: 5
```

Integers are not floating points!

```
int   b = 7;
float a = b / 2;   // a = 3 not 3.5!!
int   c = b / 2.0; // again c = 3 not 3.5!!
```

Implicit Promotion

Integral data types smaller than 32-bit are *implicitly* promoted to `int`, independently if they are *signed* or *unsigned*

- Unary `+, -, ~` and Binary `+, -, &, etc.` promotion:

```
char a = 48;        // '0'
cout << a;          // print '0'
cout << +a;         // print '48'
cout << (a + 0);    // print '48'

uint8_t a1 = 255;
uint8_t b1 = 255;
cout << (a1 + b1);  // print '510' (no overflow)
```

auto Declaration

C++11 The auto keyword specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;    // 1 is int, 2 is int, 1 + 2 is int!
//      -> 'a' is "int"
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double
//      -> 'b' is "double"
```

`auto` can be very useful for maintainability and for hiding complex type definitions

```
for (auto i = k; i < size; i++)
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Example: `auto x = 0;` in general makes no sense (`x` is `int`) `auto` (as well as `decltype`) can be used for defining both function input and output types C++11/C++14 C++20

```
auto g(int x) -> int { return x * 2; } // C++11
// "-> int" is the deduction type
// a better way to express it is:
auto g2(int x) -> decltype(x * 2) { return x * 2; }

auto h(int x) { return x * 2; }          // C++14

void f(auto x) {}                         // C++20
// equivalent to templates but less expensive at compile-time
//--------------------------------------------------------------
int x = g(3); // C++11

f(3);          // C++20
f(3.0);        // C++20
```

C++ Operators

Operators Precedence

Precedence	Operator	Description	Associativity
1	a++ a--	Suffix/postfix increment and decrement	Left-to-right
2	++a --a ! ~	Prefix increment/decrement, Logical/Bitwise Not	Right-to-left
3	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
4	a+b a-b	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	Left-to-right
6	< <= > >=	Relational operators	Left-to-right
7	== !=	Equality operators	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR	Left-to-right
10	|	Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

- **Unary** operators have <u>higher</u> precedence than **binary operators**
- **Standard math operators** (`+`, `*`, `etc.`) have <u>higher</u> precedence than **comparison**, **bitwise**, and **logic** operators
- **Comparison** operators have <u>higher</u> precedence than **bitwise** and **logic operators**
- **Bitwise** operators have <u>higher</u> precedence than **logic** operators
- **Compound assignment** operators `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `!=`, `&=`, `>>=`, `<<=` have <u>lower</u> priority
- The **comma** operator has the <u>lowest</u> precedence (see next slides)

---

en.cppreference.com/w/cpp/language/operator_precedence

Examples:

```
a + b * 4;               // a + (b * 4)

a * b / c % d;           // ((a * b) / c) %
                         d
a + b < 3 >> 4;          // (a + b) < (3 >> 4)

a && b && c || d;        // (a && b && c) || d

a | b & c || e && d;     // ((a | (b & c)) || (e && d)
```

Important: sometimes parenthesis can make expression worldly... but they can help!

Prefix/Postfix Increment Semantic

Prefix Increment/Decrement `++i`, `--i`

(1) Update the value
(2) Return the new (updated) value

Postfix Increment/Decrement `i++`, `i--`

(1) Save the old value (temporary)
(2) Update the value
(3) Return the old (original) value

Prefix/Postfix increment/decrement semantic applies not only to built-in types but also to objects

Operation Ordering Undefined Behavior ⋆

Expressions with undefined (implementation-defined) behavior:

```
int i = 0;
i = ++i + 2;          // until C++11: undefined behavior
                      // since C++11:  i = 3
i = 0;
i = i++ + 2;          // until C++17: undefined behavior
                      // since C++17: i = 3

f(i = 2, i =
1);                   // until C++17: undefined behavior
                      // since C++17:  i = 2
i = 0;
a[i] = i++;           // until C++17: undefined behavior
                      // since C++17: a[1] = 1

f(++i, ++i);          // undefined behavior
i = ++i + i++;        // undefined behavior
```

Assignment, Compound, and Comma Operators

Assignment and **compound assignment** operators have *right-to-left associativity* and their expressions return the assigned value

```
int y = 2;
int x = y = 3; // y=3, then x=3
               // the same of x = (y = 3)
if (x = 4)     // assign x=4 and evaluate to true
```

The **comma** operator has *left-to-right associativity*. It evaluates the left expression, discards its result, and returns the right expression

```
int a = 5, b = 7;
int x = (3, 4); // discards 3, then x=4
int y = 0;
int z;
z = y, x;       // z=y (0), then returns x (4)
```

Spaceship Operator <=>

C++20 provides the **three-way comparison operator** `<=>`, also called *spaceship operator*, which allows comparing two objects in a similar way of `strcmp`. The operator returns an object that can be directly compared with a positive, 0, or negative integer value

```
(3 <=> 5)       == 0; // false
('a' <=> 'a') == 0; // true

(3 <=> 5)       < 0; // true
(7 <=> 5)       < 0; // false
```

The semantic of the *spaceship operator* can be extended to any object and can greatly simplify the comparison operators overloading

Safe Comparison Operators

C++20 introduces a set of `<utility>` functions different to safely compare integers of types (`signed`, `unsigned`)

```
bool cmp_equal(T1 a, T2 b)
bool cmp_not_equal(T1 a, T2 b)
bool cmp_less(T1 a, T2 b)
bool cmp_greater(T1 a, T2 b)
bool cmp_less_equal(T1 a, T2 b)
bool cmp_greater_equal(T1 a, T2 b)
```

example:

```
#include <utility>
unsigned a  = 4;
int      b  = -3;
bool     v1 = (a > b);              // false!!!, see next slides
bool     v2 = cmp_greater(a, b);  // true
```

Control Flow

`if` Statement

if and if-else

Statements are executed conditionally

```
int num = 10;

if (num < 5)
        cout << "The number is less than 5. " << endl;
else
        cout << “else condition”;

if (num == 5 ) {
  cout << "The number is 5." << endl;
} else {
  cout << "The number is not 5." << endl;
```

if-else if-else

```
if (num < 5)
    cout << "The number is less than 5." << endl;
else if (num > 10)
    cout << "The number is greater than 10." << endl;
else
    cout << "The number is in range [5, 10]." << endl;
```

A little more complex

When will "Where I'm?" **be printed?**
How to make the code easier to understand?

```
if(num < 10)
if(num < 5)
cout << "The number is less than 5" << endl;
else
cout << "Where I'm?" << endl;
```

? : operator

When can we use the ternary conditional operator?

```
bool isPositive = true;
int factor = 0;
//some operations may change isPositive's value
if(isPositive)
    factor = 1;
else
    factor = -1;
```

```
factor = isPositive ? 1 : -1;
```

- *Ternary operator*

  `<cond> ? <expression1> : <expression2>`

  `<expression1>` and `<expression2>` must return a value of the same or convertible type

```
int value = (a == b) ? a : (b == c ? b : 3); // nested
```

Condition

```
int num = 10;

If  (num < 5)
    cout << "The number is less than 5. " << endl;
```

Condition value can be `bool`, `char`, `int`, `float`

Relational Expressions

The condition can be a relational expression

The 6 relational/comparison operators

Operator name	Example
equal to	`a == b`
not equal to	`a != b`
less than	`a < b`
greater than	`a > b`
less than or equal to	`a <= b`
greater than or equal to	`a >= b`

Logical Expressions

- **An operand is not `bool`, it will be converted to `bool` implicitly.**

Operator name	Symbol-like operator	Keyword-like operator	Example
negation	`!`	`not`	`!a`
AND	`&&`	`and`	`a && b`
Inclusive OR	`||`	`or`	`a || b`

- **Precedence:** `!` **>** `&&` **>** `||`

- **What's the value of the follow expressions?**

```
if(-2 && true)
   cout << "The condition is true." << endl;
if(not -2)
   cout << " (!-2) is true, really?" << endl;
```

Non-Boolean Expressions

- **They will be converted to `bool` implicitly if it is feasible.**

```
float count = 0.2f;
if (count) //not recommend to use a float-point number
    cout << "There are some." << endl;
```

- **Pointers are also frequently used as conditions**

```
int * p = new int[1024];
if (!p) // if(p == NULL)
    cout << "Memory allocation failed." << endl;
```

`for` and `while` **loops**

for and while Loops

- for

```
for ([init]; [cond]; [increment]) {
    ...
}
```

To use when number of iterations <u>is known</u>

- while

```
while (cond) {
    ...
}
```

To use when number of iterations <u>is not known</u>

- do while

```
do {
...
} while (cond);
```

To use when number of iterations is not known, but there is <u>at least one iteration</u>

`while` **loop**

while loop

while.cpp

Syntax :

```
while ( expression ) {
    //...
}
```

If the condition is true, the statement (loop body) will be executed.

```
int num = 10;
While (num > 0) {
    cout << "num = " << num << endl;
    num--;
}
```

do-while **loop**

while.cpp

The test takes place after each iteration in a do-while loop.
The test takes place before each iteration in a while loop.

```
int num = 10;
do {
    cout << "num = " << num << endl;
    num--;
} while (num > 0);
```

break statement

while.cpp

Terminate a loop

```
int num = 10;
while (num > 0)
{
    if (num == 5)
        break;
    cout << "num = " << num << endl;
    num--;
}
```

continue **statement**

while.cpp

Skip the remaining part of the loop body and continue the next iteration.

```
int num = 10;
while (num > 0)
{
   if (num == 5)
      continue;
   cout << "num = " << num << endl;
   num--;
}
```

The Condition, Be Careful!

Can you find any problem from the code?

```
size_t num = 10;
while(num >= 0)
{
    cout << "num = " << num << endl;
    num--;
}
```

The Condition, Be Careful!

```
bool flag = true;
int count = 0;
while(flag = true)
{
    cout << "count = " << count++ << endl;
    // and do sth
    if (count == 10) //meet a condition
    flag = false; //set flag to false to break the loop
}
```

Why?

Expression `3+4` **has a value;**
Expression `a+b` **has a value;**
Expression `(a==b)` **has value (true or false);**
`a=b` **is an assignment, also an expression and has a value**

The follow code can be compiled successfully!

```
int b = 0;
int m = (b = 8);
cout << "m="<< m << endl;
```

`for` **loop**

`for` loop

for.cpp

- **Syntax:**

```
for (init-clause; cond-expression; iteration-expression)
    loop-statement
```

- **Example:**

```
int sum = 0;
for(int i = 0; i < 10; i++)
{
    sum += i;
    cout << "Line " << i << endl;
}
cout << "sum = " << sum << endl;
```

`for` loop VS `while` loop

```
int sum = 0;
for(int i = 0; i < 10; i++)
{
    sum += i;
    cout << "Line " << i << endl;
}
```

```
int sum = 0;
int i = 0;
while (i < 10)
{
    sum += i;
    cout << "Line " << i << endl;
    i++;
}
```

`for` loop VS `while` loop

```
int num = 10;
while (num > 0)
{
    cout << "num = " << num << endl;
    num--;
}
```

```
for (int num=10; num > 0; num-- )
{
    cout << "num = " << num << endl;
}
```

Endless loop

Sometimes we need it

```
for(;;)
{
    // some statements
    cout << "endless loop!" << endl;
}

while(true)
{
    // some statements
    cout << "endless loop!" << endl;
}
```

`break/continue` statement

break and continue statements behavior the same way in both loops:

- **while loop.**
- **for loop**

`goto` and `switch` Statements

goto Statement

goto is a legacy feature - not recommended to use any more

goto.cpp

```
float mysquare(float value)
{
    float result = 0.0f;

    if(value >= 1.0f || value <= 0)
    {
        cerr << "The input is out of range." << endl;
        goto EXIT_ERROR;
    }
    result = value * value;
    return result;

  EXIT_ERROR:
    //do sth such as closing files here
    return 0.0f;
}
```

`switch` Statement

Execute one of several statements, depending on the value of an expression.
`break` **prevents executing some following statements. Don't forget** `break`**!**
More similar to `goto`**, not** `if-else if-else`

```
switch (input_char)
{
   case 'a’:
                  x = ‘a’;
                  break;
   case 'A':
      cout << "Move left." << endl;
      break;
   case 'd':
   case 'D':
      cout << "Move right." << endl;
      break;
   default:
      cout << "Undefined key." << endl;
      break;
}
```

switch.cpp

C++11 introduces the **range-based `for` loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but **safer**

The range-based `for` loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";     // print: 3 2 1

int values[] = { 3, 2, 1 };
for (int v : values)      // ARRAY OF VALUES
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")     // RAW STRING
    cout << c << " ";     // print: a b c d
```

Range-based for loop can be applied in three cases:

- Fixed-size array `int array[3]`, `"abcd"`
- Branch Initializer List `{1, 2, 3}`
- Any object with `begin()` and `end()` methods

```
std::vector vec{1, 2, 3, 4};

for (auto x : vec) {
    cout << x << ", ";
// print: "1, 2, 3, 4"
```

```
int matrix[2][4];
for (auto& row : matrix) {
    for (auto element : row)
        cout << "@";
    cout << "\n";
}
// print: @@@@
//        @@@@
```

C++17 extends the concept of **range-based loop** for *structure binding*

```
struct A {
    int x;
    int y;
};

A array[] = { {1,2}, {5,6}, {7,1} };
for (auto [x1, y1] : array)
    cout << x1 << "," << y1 << "  "; // print: 1,2  5,6  7,1
```