

C++

PENG

October 13, 2026

Contents

1	Introduction	1
1.1	Flow of Control	2
1.2	Class	2
2	Types	2
2.1	Compound Types	3
2.2	Qualifier	3
2.3	Data Structure	5
3	string	5
4	vector	6
5	Iterators	6
6	Array	6
6.1	Pointers and Arrays	7
6.2	Arrays of Arrays	7
7	Exprs	8
8	Statement	8
9	Function	8

1 Introduction

C++ is a statically typed programming language [1].

Definition 1.1 (IO objects).

```
#include <iostream>
// using std::cout, std::cin;
std::cout << "str"; // return left operand: std::cout
std::cin >> v1;    // return left operand: std::cin
// IO operators: left associative
std::cout << "str" << "\n"; // (std::cout << "str") << "\n";
std::cin >> v1 >> v2;      // (std::cin >> v1) >> v2;
```

Definition 1.2 (comment).

```
// single line
/* paired can not nest */

std::cout << /* */ /* */ /* */;
// std::cout << " /* ";
// remove everything between */ and /*
```

1.1 Flow of Control

Definition 1.3 (while).

```
while(cond)
{
    statements;
}

while(std::cin >> v)
    ...
// istream become invalid:
// reading a non-integer val: 1 2 3.1
// end-of-file: Ctrl-Z then RET (windows), or Ctrl-D then RET (linux)
```

Definition 1.4 (for).

```
for (init-statement; cond; expr) // loop var used locally inside for body
{
    statements;
}
```

Definition 1.5 (if).

```
if (cond)
{
    statements;
}
else if (cond)
{
    statements;
}
else
{
    statements;
}
```

1.2 Class

Definition 1.6 (class). *Define data structures.*

2 Types

Definition 2.1 (arithmetic types).

```
char // '\0': null, '\t': horizontal tab, '\n': newline
#include <string>
std::string
int // unsigned int >= 0
bool // true 1 and false 0
double
```

Definition 2.2 (object). *Memory region that contains data with type.*

Definition 2.3 (decl and def).

```
// def: allocate storage and define only once
// decl: specify type and var names
// headers should have guards to be safe to be included multiple times
// source.cpp
int i = 10;
// source.h
```

```

#ifndef SOURCE_H
#define SOURCE_H
extern int i;
#endif
// main.cpp
#include "header.h"
i ...

```

Definition 2.4 (bit).

<i>bit</i>	<i>combination</i>	<i>total</i>	<i>unsigned</i>	<i>int</i>
1	0, 1	2^1	0, 1	0, 1
2	00, 01, 10, 11	2^2	0, 1, 2, 3	0, 1, -2, -1
...				
32		2^{32}	0, 1, ..., $2^{32} - 1$	0, ..., $2^{16} - 1, -2^{16}, \dots, -1$

Definition 2.5 (type conversion).

```

// int => unsigned
unsigned i = -1;           // -1 = 2^32 - 1 for 32-bit unsigned int
#include <cmath>
unsigned i = std::pow(2, 32); // 2^32 = 0 for 32-bit unsigned int
unsigned a = 1; int b = -1;
a + b // 1 + (2^32 - 1) = 2^32 = 0
a < b // 1 < (2^32 - 1) true

```

2.1 Compound Types

Definition 2.6 (reference). *A ref defines an alias for an object.*

```

int i = 0;
int &ref = i;

```

Definition 2.7 (pointer). *A pointer is an object that holds the address of another object.*

```

int i = 0;
int *p = &i; // p holds the address of i
*p          // return the object i
p = nullptr; // null pointer: not bound to any object
p = 0;

```

```

// pointer to pointer
int **pp = &p; // * indicate pointer level

```

```

// ref to pointer
int *&ref = p; // refer to int*

```

2.2 Qualifier

Definition 2.8 (const). *The compiler will replace the const var with its value.*

```

// const must be initialized
const int i = 1;

// const var is local to file by default
// must specify extern to be used in other files
// source.cpp
extern const i = 10;
// header.h
extern const i;
// main.cpp

```

```

#include "header.h"
i ...

// ref to const
// cannot change the object to which it bounds
const int i = 1;
const int &r = i;
const int &r2 = 2; // compiler: const int temp = 2; const int &r2 = temp;

// pointer to const
// cannot change the object to which it points
const int i = 1;
const int *p = &i;

// const pointer
// the address it holds cannot be changed
int i = 1;
int *const p = &i; // p always points to i

```

Definition 2.9 (constant expr). *A constant expression is an expr whose value cannot change at compile time.*

```

constexpr int i = size(); // ok only if size is a constexpr function
constexpr int *p = nullptr; // const pointer, constexpr applies to p

```

Definition 2.10 (type alias). *A synonym for a type.*

```

typedef int base;
using base = int;
typedef base *p; // int*: pointer to int
p p1 = &i;

// const p: const modify the given type: int *const: constant pointer to int
const p p2 = 0; // int *const p2 = 0
const p *p3; // int *const *p3

```

Definition 2.11 (const level).

```

// top-level const: an object itself is const, i.e. const pointer
const int i = 1;
int *const p1 = &i;

// low-level const: point or refer to const object
const int *p2 = &i;
const int &r = i;

// when copying, top-level const is ignored,
// low-level const is never ignored
const int *const p3 = &i;
p2 = p3;

```

Definition 2.12 (auto). *Deduce the type from the initializer at compile time.*

```

auto i = 1, *p = &i; // must have initializer
auto i = 0, pi = 3.14; // error, types must be consistent

// auto may differ from initializer's type
// auto ignores top-level const

```

Definition 2.13 (decltype). *Return the operand type without evaluating the expr.*

```

decltype(expr) var

```

2.3 Data Structure

Definition 2.14 (structure). *A data structure is a way to group together related data elements and for using those data.*

```
// class is ordinarily defined outside functions, usually in className.h
struct className
{
    statements;
}; // must end with ;

className x, *y;
```

3 string

Definition 3.1 (string). *Variable-length sequence of characters.*

```
#include <string>
// using std::string;

// class defines how objects can be initialized
std::string s = "A"; // copy ['A', '\0'], drop null char
std::string s(n, 'c'); // "c...c"

// read string
std::cin >> s1 >> s2;

// read line
std::getline(std::cin, s)

// compare
// at least one should be std::string
std::string s = "Hello";
s < "Hiya" // compare first diff char ASCII: i > e
s < "Hello World"

// concat
// cant add two string literals, i.e. "A" + "B"
s + " World\n" // "Hello World\n"

// for range statement: iterate through elems in a given sequence
for (char c : s)
    ...
for (char &r : s) // change char in string
    r = toupper(r); // uppercase

// ispunct(c): check if char is punctuation, i.e. ',', '!', ...
// isspace(c): check if char is space

// subscript operator: s[n]
// !s.empty()
// s.size(): unsigned std::string::size_type
// s.size() < -1 => 2^32 - 1 for 32-bit int
// n: unsigned std::string::size_type
// 0 <= n < s.size()
for(decltype(s.size()) i = 0; i != s.size(); i = i + 1)
    ...

// iterator
```

4 vector

Definition 4.1 (vector). *Variable-length sequence of objects of a given type.*

```
#include <vector> // class template
// using std::vector;

std::vector<int> v; // empty vector
std::vector<int> v = {val_1, ..., val_n};
std::vector<int> v(size, val); // {val, ..., val}
v.push_back(elem) // append elem

// for range statement
for(auto &r : v)
    ...

// compare

// subscript operator
// !v.empty()
// v.size(): unsigned std::vector<int>::size_type
// 0, ..., v.size() - 1
```

5 Iterators

Definition 5.1 (iterator). *Access the elem in container. General mechanism, all lib containers have iterator.*

```
auto b = v.begin(); // return iterator: denotes first elem
auto e = v.end();   // return iterator: denotes one after the last elem

// deref
*b // return ref to elem obj
b->mem // arrow operator: (*b).mem, i.e. b->empty()

// operation
++b // next elem
--e // previous elem
b != e

// for vector and string iterators
auto mid = b + (e - b) / 2;
b <= e
```

6 Array

Definition 6.1 (array).

```
constexpr size_t size = ...;
int a[size] = {};
// array type cannot be: auto
// any object size type: unsigned size_t
// dimension must be known at compile time: constexpr

// cannot copy
// error: a2 = a1;

// pointers and ref
```

```

int *p[size]; // array of n pointers to int
int (*p)[size] = &a; // pointer to array of n int
int (&r)[size] = a; // refer to array of n int
int *(&r)[size] = p; // refer to array of n pointers to int
// () for def -> right for array size -> left for elem type

// for range statement
for (auto &r : a)
    ...

// subscript operator

```

6.1 Pointers and Arrays

```

int *p = a; // a automatically convert to &a[0]
auto p = a; // p = &a[0]
decltype(a) a2 = {0, 1, 2};

// iterator
int *b = std::begin(a); // pointer to first elem
int *e = std::end(a); // pointer to one after last elem

// subscript
int *p = &a[m]; // p = a + m
p[n] // *(p + n) => a[m + n]

// C-style char string
char c[] = "str"; // c[] = {'s', 't', 'r', '\0'}
strcmp(c1, c2) // 0 for c1 == c2; 1 for c1 > c2; -1 for c1 < c2

// initialize vector with array {a[i], ..., a[j]}
std::vector<int> v(a + i, a + j);

```

6.2 Arrays of Arrays

Definition 6.2 (multidimensional array).

```

constexpr size_t rows=..., cols=...;
int a[rows][cols] = {0};

// subscript operator
a[n] // nth inner-array elem

// for range statement
// control vars except innermost array must be ref
for (auto &row : a) // for every elem in outer array
    for(auto col : row) // for every elem in inner array
        ...

// pointers
int (*p)[cols] = a; // p = &a[0]

for (auto p = a; p != a + rows; ++p) // int (*p)[cols] = &a[0]
    for (auto q = *p; q != *p + cols; ++q) // int *q = &a[0][0]
        ...

// type alias
using int_array = int[rows];
// typedef int int_array[rows];

```

```
for (int_array *p = a; p != a + rows; ++p)
    ...
```

7 Exprs

Definition 7.1 (expression). *Composed of operators and operands.*

```
// unary operator: act on one operand
// & address-of
// * dereference

// binary operator: act on two operands1
// == equality
// * multiplication

// overloaded operator
// define operator in class

// lvalue and rvalue
// lvalue: can be assignment
// rvalue: cannot be assignment
// lvalue = ...           return left-hand operand as lvalue
// address-of &lvalue      return a pointer to lvalue as rvalue
// deref * and subscript operator return lvalue
// ++iterator or --iterator return lvalue
// decltype(lvalue)        return ref
// decltype(rvalue)        return the exact type of rvalue
decltype(*p) // int&
decltype(&i) // int*
decltype(&p) // int**

// compound expr with two or more operators
// precedence and associativity define how operands grouped
// () force grouping
```

8 Statement

9 Function

Definition 9.1 (func). *Block of code with name.*

References

- [1] S.B. Lippman, J. Lajoie, and B.E. Moo. *C++ Primer*. Addison-Wesley Professional, 2012. ISBN: 978-0-321-71411-4.