

C++

PENG

October 16, 2026

Contents

1	Introduction	1
2	Types	2
2.1	Compound Types	3
2.2	Qualifier	3
2.3	Data Structure	4
3	string	4
4	vector	5
4.1	Iterators	6
5	Array	6
5.1	Arrays of Arrays	7
6	Exprs	7
6.1	Type conversion	9
7	Statement	10
7.1	Try and Exception	11
8	Function	11
8.1	Default Arguments	13
8.2	inline and constexpr Func	13
8.3	Debugging	13
8.4	main	14
8.5	return	14
8.6	Recursion	15
8.7	Overloaded Funcs	15
8.8	Pointers to Funcs	15
9	class	16

1 Introduction

Definition 1.1 (C++). *C++ is a statically typed programming language [1].*

Definition 1.2 (IO objects).

```
#include <iostream>
// using std::cout, std::cin;
std::cout << "str"; // return left operand: std::cout
std::cin >> v;      // 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;
```

```

while(std::cin >> v)
    ...
// istream become invalid when
// 1. reading a non-integer val: 1 2 3.1
// 2. end-of-file: Ctrl-z RET (Windows), Ctrl-d RET (Linux)
Definition 1.3 (comment).

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

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

```

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, ..., $2^{16} - 1, -2^{16}, \dots, -1$

1 byte = 8 bits

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 object i lvalue
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 different 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 data elements.*

```

// ordinarily defined outside functions in ClassName.h
struct ClassName
{
    statements;
}; // must end with ;

```

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

```

4.1 Iterators

Definition 4.2 (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
```

5 Array

Definition 5.1 (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 to array
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)
    ...

// pointer to array elem
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);

```

5.1 Arrays of Arrays

Definition 5.2 (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)
    ...

```

6 Exprs

Definition 6.1 (expression). *Composed of operators and operands.*

```

// unary operator: act one 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:
// operands of operators with higher precedence group tightly than
// operands of operators at lower precedence
// associativity: group operands with same precedence
// left associative: operators with same precedence group from left-most to right
// right associative: operators with same precedence group from right-most to left
// () force grouping

// arithmetic operator
// left associative
// return rvalue
// reminder: int % int
// return same sign as m
(-m) % n // -(m % n)
m % (-n) // -(m % n)

// logical and relational operator
// return bool val
// 0 return false
// non-zero return true
// guarantee eval order
expr1 && expr2 // eval expr2 if and only if expr1 is true
expr1 || expr2 // eval expr2 if and only if expr1 is false
// precedence:
// right associative
// 1. !
// left associative
// 2. <=, >=
// 3. ==, !=
// 4. &&
// 5. ||
i != j < k // i != (j < k)

// assignment operator
// low precedence
// right associative
// left-hand operand: a modifiable lvalue
// return left-hand operand lvalue
// compound assignment operator
// *=, /=, %=
a *= b; // compound assignment: a eval only once
a = a * b; // ordinary assignment: a eval twice

// increment and decrement operator
// require lvalue operand
// prefix: increment operand and return the changed object as lvalue
// postfix: increment operand and return object's original value as rvalue
// use postfix only when necessary
// precedence: ++-- > dereference
*b++ // *(b++)

// evaluation order

```

```

// member access operator
p->mem => (*p).mem
// precedence: . > dereference
pointer-> return lvalue
.mem return lvalue or rvalue same as mem

// conditional operator
cond ? expr1 : expr2;
// return lvalue if both expr1 and expr2 are lvalue
// otherwise return rvalue
cond ? expr1 : cond ? expr2 : expr3
// right associative
// low precedence, need ()

// bitwise operator
// hardware

// shift operator (IO operator)
// left associative
// precedence:
1. arithmetic operator
2. IO operator
3. relational, assignment, conditional operator

// sizeof operator
// right associative
// return constexpr size_t in byte
// 1 byte = 8 bits
sizeof(type)
sizeof(expr)
// array
sizeof(a) // sizeof(*a) * size

// comma operator
// guarantee eval order from left to right
// lowest precedence
// expr1, expr2
// return expr2 lvalue or rvalue
for (int v1=..., v2=...; cond; expr1, expr2)
    ...

```

6.1 Type conversion

Definition 6.2 (type conversion).

```

// implicit conversion
// arithmetic conversion
// integer promotion
// unsigned type
// array to pointer conversion
// conversion to bool
// conversion to const

// explicit conversion
// static_cast<type>(expr)
int i = 1, j = 2;
// double k = i / j; // 0
double k = static_cast<double>(i) / j; // 0.5
// const_cast: remove low-level const

```

```
// reinterpret_cast: bit-pattern of operands
// old-style cast
// type (expr);
// (type) expr;
```

7 Statement

Definition 7.1 (statement).

```
// simple statement
// expr statement
; // null statement
// compound statement (block, scope)
// {} empty block is equivalent to ; null statement

// flow-of-control statement
// conditional statements
// if statement
if (cond)
    statement;
else if (cond)
    statement;
else
    statement;

// switch statement
switch (v)
{
    case expr1: // constexpr
        statement;
        break;
    case expr2:
        statement;
        break;
    ...
    default: // no case match
        statement;
        break;
}

// iterative statement
while(cond)
    statement;

// traditional for statement
for (init-statement; cond; expr) // loop var used locally inside for body
    statement;

// range for statement
// iterate through elems of a container or sequence
for (declaration : sequence) // ref to write elem
    statement;

// do while statement
do
    statement
while (cond);
```

```

// jump statements:
// break statement
break; // terminate nearest loop or switch

// continue statement
// terminate current iteration of nearest loop and begin next iteration
continue;

// goto statement
label:
    ...
goto label;

// statement scope
// vars defined in control structure are only visible within scope

```

7.1 Try and Exception

Definition 7.2 (exception).

```

// try block
try
{
    throw std::runtime_error("ERROR MESSAGE");
}
catch (std::runtime_error e)
{
    std::cout << e.what() << std::endl; // ERROR MESSAGE
}

// exceptions

```

8 Function

Definition 8.1 (func). *Block of code with name.*

```

type func(params) // each param with single declarator: int v1, int v2, ...
{
    statement;
    return r;
}

// automatic objects
// params destroyed after function call

// static object
// lifetime continues after function call
static type v = ...;

// func decl
// a function must be declared before use
// semicolon replace the func body in declaration
// func.h
type func(params);
// func.cpp
// header that decls func should be included in source file that defines func
#include "func.h"
type func(params) {...}
// main.cpp

```

```

#include "func.h"
func(arguments);

// separate compilation
// g++ -c compile and assemble but do not link, generate object file
g++ -c main.cpp // main.o
g++ -c func.cpp // func.o
g++ main.o func.o // link object files, generate a.out

// param and argument
// param are initialized in the same way that vars are initialized:
// type param = argument;
param is ref => called by ref, param is an alias for argument to which it is bound
param is not ref => called by value, argument value is copied
// pointer param
// copy pointer value
void reset(int* p)
{
    *p = 2; // change value of object to which p point
    p = 0; // only change the local copy of p, argument is unchanged
}

int i = 1;
reset(&i); // i = 2

// C use pointer param
// C++ often use reference param instead
// ref param
void reset(int& r)
{
    r = 2;
}

int i = 1;
reset(i); // i = 2

// ref param for large containers
void func(const std::string &r);
// ref param for multiple results return

// const param
// argument -> param: top-level const of argument is ignored
// argument -> pointer or ref param: low-level const of argument cannot be ignored
// when a func do not need write access to argument object,
// use ref or pointer to const param for std::string, std::vector, and array

// array param
// cannot copy array
// array converted to a pointer to the first elem in array
void func(const int*);
// size of array
// C-style char string: stop when seeing null char
void func(const char*);
// std lib
void func(const int *b, const int *e);
// define a param that indicates the size of array
void func(const int *, size_t size);

```

```

// array ref param
void func(int (&a)[dim]); // param is ref to array of n ints

// multidimensional array argument
// subsequent dim must be specified
void print(int (*matrix)[dim], int rows, ...)
// matrix points to first elem in array whose elems are array of 10 ints

// varying number of params
// std::initializer_list: template type
void func(params, std::initializer_list<T> il);
// number of arguments of a single type in il
// elems in il are always const
void error_msg(std::initializer_list<std::string> il)
{
    for (auto b = il.begin(); b != il.end(); ++b)
        std::cout << *b << " ";
}

error_msg({"FUNC", s1, s2});

// ellipsis params
void func(params, ...)

```

8.1 Default Arguments

Definition 8.2 (default argument). *Func with default argument can be called with without that argument.*

```

// func.h
// default arguments ordinarily should be specified with func declaration in header
type func(params, def_defaults_for_params);

```

8.2 inline and constexpr Func

Definition 8.3 (inline func). *Calling a small func is slower than equivalent expr. A inline function is expanded in line at each call to avoid func call overhead.*

```

inline type func(params);
// inline is only a request to compiler.
// The compiler may choose to this request.

// inline and constexpr func defined in header

```

Definition 8.4 (constexpr func). *Can be used in constant expression.*

```

constexpr type func(params);
// implicitly inline
// replace a call with return value
// return type must be literal type
// func body contains exactly one return statement

```

8.3 Debugging

Definition 8.5 (assert and NDEBUG). *Debugging code is executed only during developing.*

```

// #define NDEBUG // turn off assert, defined before #include <cassert>
#include <cassert>
assert(expr); // eval expr. if expr is false, write a message and terminate program.

// preprocessor macro
// preprocessor names are managed by preprocessor not compiler

```

```

// use preprocessor name directly and do not provide using decl

// conditional debugging using NDEBUG
void func(int i)
{
#ifdef NDEBUG
    std::cerr << __func__ << ": i is " << i << std::endl;
#endif
}
// local static defined by compiler
// __func__: func name
// __FILE__: file name
// __LINE__: current line number
// __TIME__: compiled time
// __DATE__: compiled date

```

8.4 main

Definition 8.6 (command-line options to main).

```

int main(int argc, char *argv[]) {...}
int main(int argc, char **argv) {...}

// argc: size of argv
// argv[0]: program name
// argv[1]: option1
// argv[2]: option2
// ...

```

8.5 return

Definition 8.7 (return). *Terminate func and return control to the point from which func was called. Analogous to the use of break statement to exit loop.*

```

return; // for void func()
return expr; // type tmp = expr;
// values are returned in the same way as vars and parameters are initialized

// return ref: call is lvalue
// other return type: rvalue
int& func(int& r)
{
    return r; // int &tmp = r
}
int i = 0;
func(i) = 1; // call is lvalue

// never return ref or pointer to func local var

// return list
std::vector<int> func()
{
    return {1, 2, 3};
}

// return pointer to array
type (*func(params)) [dim]; // return pointer to array of n type elems
// like int (*p)[3] = &a;
// use typedef

```

```

typedef int arrayT[dim];    // synonym to array of n ints
using arrayT = int[dim];
arrayT* func(params);
// use decltype
int a[] = {1,2,3};
decltype(a) *func(params);
// use trailing return type
auto func(params) -> int(*)[dim];

// call operator
// precedence: same as dot and arrow operator
// left associative
func().mem

```

8.6 Recursion

Definition 8.8 (recursion). *A function that calls itself is a recursive function.*

8.7 Overloaded Funcs

Definition 8.9 (overload). *Same name funcs with different params list in one scope.*

```

// error for two funcs to differ only in terms of return types

// top-level const is indistinguishable from one without a top-level const
// except low-level const
// const_cast

// func matching
// 1. candidate funcs: all funcs
// 2. viable funcs: arguments match params
// 3. best match: exact match is better than argument type conversion
// argument type conversion:

```

8.8 Pointers to Funcs

Definition 8.10 (func pointer). *Function pointer denotes func rather than object. It points to a function type determined by return type and params types.*

```

type func(params);
type (*p)(params);

p = func;          // func is automatically converted to pointer
// p = &func;     // equivalent assignment
p(params);        // call func
// (*p)(params);  // equivalent call
p = nullptr;      // point to no func
// p = 0;

// func pointer as param
type func(params, type p(params));
// type func(params, type (*p)(params)); // equivalent decl

// typedef
typedef type F(params);    // func type
using F = type(params);    // equivalent func type
typedef decltype(func) F;  // equivalent func type

typedef type (*PF)(params); // pointer to func type

```

```

using PF = type(*) (params); // equivalent pointer to func type
typedef decltype(func) *PF; // equivalent pointer to func type

PF p;
F *p;

// def func return pointer to func
PF func(params);
F *func(parms);
type (*func(params)) (params); // directly decl
auto func(params) -> type (*) (params);
decltype(func2) *func(params);

```

9 class

Definition 9.1 (class). *Define abstract data type relying on the separation of interface and implementation.*

```

// every member must be declared inside class
// const mem func

```

References

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