

C++

PENG

October 30, 2026

Contents

1	Introduction	2
2	Types and Variables	2
2.1	Variable Initialization	2
2.2	Compound Types	3
2.3	const	4
3	String	6
4	vector	7
4.1	Iterators	7
5	Array	7
5.1	Array Declarations	8
5.2	Access Elements of Array	8
5.3	Arrays of Arrays	8
6	Exprs	9
6.1	Type conversion	11
7	Statement	12
7.1	Try and Exception	13
8	Function	13
8.1	Func Declaration	14
8.2	Parameter Initialization	14
8.3	Return Statement	15
8.4	Recursion	16
8.5	Pointers to Funcs	16
8.6	Overloaded Funcs	16
8.7	Default Arguments	17
8.8	inline and constexpr Func	17
8.9	Command-Line Options to main	17
8.10	Debugging	17
9	class	18
9.1	Additional Class Features	19
9.2	Constructor	20
9.3	Operator	22
9.4	OOP	22
9.5	Template	22
10	Sequential Containers	23
11	Generic Algorithms	23
12	Associative Containers	23

13 Dynamic Memory	23
A Tools	24

1 Introduction

Definition 1.1 (C++).

- *C is based on functions.*
- *C++ is object-oriented: separation of abstract mathematical concepts and black-box implementation [1, 2].*

2 Types and Variables

Definition 2.1 (arithmetic type).

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

Definition 2.2 (bit). *1 byte = 8 bits*

<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, ..., $2^{16} - 1, -2^{16}, ..., -1$

Definition 2.3 (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 Variable Initialization

Definition 2.4 (object). *An object is a memory region that contains data with type.*

Definition 2.5 (initialization and assignment).

- *initialization happens when a variable is given a value when it is created*
 - *initialize at compile time*
 - *initialize at run time*
- *assignment replace an object's current value with a new one*

Definition 2.6 (list initialization).

```
// initialization form
// = for copy initialize
// omit = for direct initialize
```

```
type var = value;
```

```

type var = {value}; // list initialization
type var{value};    // narrowing conversion, i.e. error convert double to int
type var(value);    // parenthesis initialization

```

Definition 2.7 (definition and declaration).

```

// definition: allocate storage and define only once
// declaration: based type followed by a list of declarators specifying type and variable name

```

```

// headers should have guards to be safe to be included multiple times
#ifndef HEADER_H
#define HEADER_H
extern type var;
#endif
// source
#include "header.h"
type var = ...;
// main
#include "header.h"

```

Definition 2.8 (preprocessor). *Preprocessor is a program that runs before the compiler.*

```

// preprocessor are not part of std namespace
// refer to preprocessor directly without std:: prefix
// preprocessor automatically replace preprocessor variable by its value

```

2.2 Compound Types

Definition 2.9 (compound type). *Compound type is defined in terms of another type.*

Definition 2.10 (reference). *A reference defines an alternative name for an object.*

```

// reference must be initialized with object
type var = ...;
type &ref = var; // bind reference to its initializer
// reference cannot be rebind

```

Definition 2.11 (pointer). *A pointer is an object that holds the address of another object.*

```

type var;
type *p = &var; // pointer to var
// pointer type and object type to which pointer points must match
*p // dereference return lvalue object to which pointer points
p = nullptr; p = 0; // null pointer doesn't pointer to any object

// void* pointer
void *p = &var; // can hold address of any object type
// cannot use void* to operate on object it address
// use void* pointer to deal with memory rather than access object stored in that memory

// pointer to pointer
type **pp = &p; // indicate each pointer level by its own *

// reference to pointer
type *&ref = p; // refer to type*

```

Definition 2.12 (compound type declaration).

```

// variable definition consists of a base type and a list of declarators
// each declarator can relate its variable to the base type
type var = ..., *p = &var, &ref = var;

```

2.3 const

Definition 2.13 (const). *const makes a variable unchangeable with compiler replacing const variable with its value.*

```
// const object must be initialized at compile time or run time
// use an object to initialize another object,
// it doesn't matter whether either or both of objects are consts
// copy an object doesn't change that object
const type var = ...;
```

Definition 2.14 (const objects are local to file).

```
// const variables are local to file by default
// to share a const object among multiple files,
// use extern on both its definition and declaration
// header
extern const type var; // extern signifies that var is not local to this file,
// and that its definition will occur elsewhere
// source
#include "header.h"
extern const type var = ...;
// main
#include "header.h"
var
```

Definition 2.15 (const reference).

```
// bind reference to const to const object
// cannot be used to change the object to which reference bounds
const type var = ...;
const type &ref = var;
// can initialize from any expression that can be converted to the type of reference,
// i.e. can bind const reference to nonconst object, literal, or expression
double d = 3.14;
const int &ref = d; // const int temp = d; const int &ref = temp;
// bound to an unnamed temporary object created by compiler when it needs place
// to store result from expression
```

Definition 2.16 (pointer to const).

```
// cannot change the object to which the pointer points
const type var = ...;
const type *p = &var;
// can point to nonconst object
// pointer to const only affects what we can do with pointer
```

Definition 2.17 (const pointer).

```
// const pointer must be initialized
// the address pointer holds cannot be changed
type var = ...;
type *const p = &var; // p always points to var
// const pointer says nothing about whether we can use pointer to change object,
// which depends on type to which pointer points
```

Definition 2.18 (const level).

```
// top-level const: an object itself is const and cannot be changed
const type var = ...;
int *const p = &var;

// low-level const: point or refer to a const object
```

```
// const reference is always low-level
const type *p = &var;
const type &ref = var;

// top-level const is ignored when copying an object
// low-level const is never ignored
const type *const p1 = &var;
const type *p2 = p1;
```

Definition 2.19 (constexpr). *The value of constant expression cannot change and can be evaluated at compile time (not until run time).*

```
// whether an object is constant expression depends on types and initializers
// literal and const object that is initialized
// from constant expression is constant expression
const type var = ...;
// declare variable in constexpr declaration to ask compiler
// to verify that variable is a constant expression
// variable declared as constexpr are implicitly const and
// must be initialized by constant expression
constexpr type var = func(arguments); // can use constexpr function in the initializer

// can only use constexpr on literal types
// arithmetic, reference, pointer types are literal types
// constexpr pointer can be initialized from nullptr or 0 literal, or
// point/bind to fixed address object
// address of object defined outside any function is constant expression

// constexpr applies to pointer, not the type to which pointer points
// constexpr imposes top-level const on the object it defines
const type *p = nullptr; // pointer to const type
constexpr type *p = nullptr; // const pointer to type
constexpr const type *p = ...; // const pointer to const type
```

Definition 2.20 (typedef). *A type alias is a name that is synonym for another type.*

```
typedef type T;
// alias declaration
using T = type;

typedef type *T; // type* pointer to type
// const appears in the base type modify the given type
// const T is const pointer to type
const T p = 0; // const pointer to type
const T *p; // pointer to const pointer to type
```

Definition 2.21 (auto). *Use auto type specifier to let compiler to deduce type from initializer.*

```
// variable that uses auto specifier must have an initializer
auto a = ...;

// when using reference as initializer,
// the initializer is the corresponding object,
// and the compiler uses that object's type for auto's type deduction
// auto drops top-level consts and low-level consts are kept
const type var = ...;
const type &ref = var;
auto a = ref; // initializer object is const type var

// make deduced type have top-level const
const auto a = var; // a is const type
```

```
// reference to auto-deduced type,
// where top-level consts in initializer are not ignored
auto &a = var; // a is const type bound to var
```

Definition 2.22 (decltype). *decltype type specifier returns type of its operand without evaluating the expression.*

```
decltype(expr) d = ...;
// decltype returns the type of variable, including top-level const and references
const type var = ..., &ref = var, *p = &var;
decltype(var) d = ...; // d is const type
decltype(ref) d = ...; // d is const type& and must be initialized
// decltype return a reference type for expressions that yield lvalue objects
// type deduced by dereference a pointer is reference and must be initialized
decltype(*p) d = ...; // d is const type& reference

// decltype of a parenthesized variable is always a reference
decltype((var)) d = ...; // d is const type& and must be initialized
```

3 String

Definition 3.1 (string). *A string is a variable-length sequence of characters.*

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

// class defines how objects can be initialized
std::string s = "A"; // not include null char at the end of string literal
std::string s(N, 'c'); // N copies of char

// 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). *Array is a fixed-size container of unnamed objects of a single type.*

```
// arrays are compound type
type a[N];
```

```
// cannot use auto to deduce type
// N is constexpr unsigned size_t
```

5.1 Array Declarations

Definition 5.2 (pointer, ref, and array).

```
// cannot initialize or assign array with another array

// 1. array of pointers
type *a[N]; // array of N pointers to type

// 2. pointers or reference to array
type (*p)[N] = &a; // pointer to array of N type elements
type (&r)[N] = a; // reference to array of N type elements

// 3. pointers or reference to array element
type *p = &a[0];
type *p = a; // a is type* pointer to first element

// auto return type is type* pointer to first element
auto p = a; // p = &a[0]
// decltype return type is array of N type elements
decltype(a) a2;
```

5.2 Access Elements of Array

Definition 5.3 (array element).

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

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

// subscript
// should define var to have unsigned size_t type
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'}

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

5.3 Arrays of Arrays

Definition 5.4 (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)
    ...

```

Definition 6.2 (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)

```

6.1 Type conversion

Definition 6.3 (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). *Function is a block of code with name.*

```

type func(params) // single declarator: type v1, type v2, ...
{
    return value;
}

// call operator
// 1. initialized func params from arguments
// 2. execute calling function -> execute called function ->
// ends when encountering return statement ->
// return value -> back to calling function ->
// initialize call expression with returned value
func(arguments);

```

Definition 8.2 (lifetime).

```

// global objects
// allocated at program start-up and destroyed when the program ends

// local variable
// automatic object
// params destroyed after function call

// local static object
// destroyed when program terminates
static type var = ...;

```

8.1 Func Declaration

Definition 8.3 (func prototype).

```
// func must be declared before we can use

// header
type func(params); // semicolon replaces func body in decl

// source
// should include header so that compiler will verify func def and decl are consistent
#include "func.h"
type func(params) { ... }

// separate compilation
g++ -c main.cpp // generate object file main.o
g++ -c func.cpp // func.o
g++ main.o func.o // link object files
```

8.2 Parameter Initialization

Definition 8.4 (param initialization).

```
// C++ use reference param to avoid copies
type& param = argument // param bound to argument

// nonreference
type param = argument // argument value is copied
// C use pointer param to access object
type* param = &argument

// const param
// top-level const are ignored
type param = const argument
// low-level const cannot be ignored
const type& param = const argument
const type* param = &const argument
```

Definition 8.5 (array param).

```
// array param
// cannot copy array
type array[N] = {};
type func(const type* param); // array converted to &array[0]
type func(const type param[]); // equivalent to const type* param
type func(const type param[M]); // compiler ignores first dimension
// cannot use type var[] = array in expr!

// when func doesn't need write access to array elements,
// array param should be pointer to const

// array size rely on additional inform:
// C-style string stops at null char
type func(const char* cp) {if(cp) ...}
// beg and end pointer to first and last element in array
const int* beg = std::begin(array)
const int* end = std::end(array)
// define a second param indicates array size
type func(const int[], size_t);
```

```

// array ref param
int (&param)[N] = array // type size limits usefulness of func

// multidimensional array param
// the second and subsequent dimensions must be specified
type func(int (*matrix)[Y][Z][...], int X);
type func(int matrix[][Y][Z][...], int X); // compiler ignores first dimension

// 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, ...)

```

8.3 Return Statement

Definition 8.6 (return). *Terminate func and return control to func call.*

```

// return statement
// analogous to break statement
return; // return with no value used in void func(params);
return value; // type temp = value
// return value is used to initialize a temporary as result of func call

// return reference
// funcs that return reference are lvalues
// other return types are rvalues

// call operator
// same precedence as dot and arrow operator
// call operator is left associative
func(argument).member_func() // return class type object

// return list
std::vector<type> temp = { ... }

// cannot return array or func
// return pointer or reference to array or func
type (*func(params)) [N]; // return pointer to array of N type elements

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

```

```
// main return
// 0 indicates success
// nonzero has machine-dependent meaning
// machine-independent preprocessor variable
// EXIT_SUCCESS
```

8.4 Recursion

Definition 8.7 (recursion). *A function that calls itself is a recursive function.*

8.5 Pointers to Funcs

Definition 8.8 (function pointer). *Function pointer points to func type determined by return 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(params);
type (*func(params)) (params); // directly decl
auto func(params) -> type (*) (params);
decltype(func2) *func(params);
```

8.6 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:

// use small func inside overloaded funcs to avoid writing same code in more than one place
```

8.7 Default Arguments

Definition 8.10 (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);
```

8.8 inline and constexpr Func

Definition 8.11 (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 8.12 (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
```

8.9 Command-Line Options to main

Definition 8.13 (argument to main). *Use argument to main to guid operation of program.*

```
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
// ...
```

8.10 Debugging

Definition 8.14 (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

```

9 class

Definition 9.1 (class). *Data abstraction and encapsulation: separation of interface and implementation*

- *interface: operations that users can execute*
- *implementation: class data members and funcs*

```

// header

// class MyClass; // forward declaration if class appears in friend func params or return

// declare friend func in class header
// friend func can be defined inside class
// friend func decl must still be provided outside class before other member func use it
type nonmember_func(params);

class MyClass
{
public:
    friend type nonmember_func(params);
    // allow nonmember func to access class nonpublic members
    // friends are not affected by access control
    // group friend decls at beginning or end of class
    // friend class MyClass2; // access all member funcs in MyClass2
    // friend type MyClass2::member_func(params);

    // member funcs access object through implicit parameter this
    // any use of member of class inside member func is an implicit reference through this
    type func(params) const; // const member func indicates that this is a pointer to const
    // const member func cannot change data members of object on which it is called

private:
    // class is not defined until its class body is complete
    // class cannot have data members of its own type
    // class can have data members that are pointers or references to its own type
    type var;
}; // must end with ;

// source
type MyClass::func(params) const { ... }
type nonmember_func(params) { ... }

// main
MyClass myObj;

```

9.1 Additional Class Features

Definition 9.2 (`this`).

```
// funcs return reference are lvalue objects
type func(params) { return *this; }

const type func(params) const { return *this; }
// const member func return *this as reference should return a reference to const

// member func can be overloaded based on whether it is const
```

Definition 9.3 (`class scope`).

```
// incomplete type

// access type member outside class
MyClass::T var = ...;

// return type must specify class when member func defined outside class
MyClass::T MyClass::func(params);

// name lookup inside class
// an inner scope can redefine a name from an outer scope except redefining type name
type var = ...;
class MyClass { type var = ...; // can access outer object using scope operator ::var }

// name lookup for member function defined outside class
// func scope -> class scope -> enclosing scope
```

Definition 9.4 (`type member`).

```
class MyClass
{
    typedef type T; // type member must appear before they are used
    // appear at beginning of class

    T var;
}
```

Definition 9.5 (`inline member func`).

```
// member funcs defined inside class are automatically inline
// member funcs defined outside class need to be specified inline
class MyClass { type func(params); }

inline type func(params) { ... }
// inline member func should be defined in class header
```

Definition 9.6 (`mutable data member`). *A mutable data member is never const.*

```
// const member func can change a mutable member
mutable type var;
```

Definition 9.7 (`static class members`). *A member is associated with the class rather than with individual object of class type by adding the keyword `static` to its declaration.*

```
// static data members are not part of individual objects of the class type,
// they are not initialized by class constructors when creating objects of the class
// static member functions are not bound to any object and do not have this pointer
static type var;
static type func(params) { ... }
```

```

// we must define and initialize each static data member outside the Class
// static member are defined outside any function like global objects
type MyClass::var = ...;

// static keyword is used only on the declaration inside class body
// static member defined outside the class do not need to repeat the static keyword

// in-class initialization of static data members
// can provide in-class initializer for static members that have integral type
// must do so for static members that are constexpr of literal type
// initializer must be constant expression
static constexpr type var = ...;
// should be defined outside class definition even if
// const static data member is initialized inside class body
constexpr type MyClass::var; // initializer provided in class definition

// can access a static member directly through scope operator
MyClass::func()

// static data member can have incomplete type,
// i.e. same type as the class type
static MyClass var;

```

9.2 Constructor

Definition 9.8 (constructor). *Class initialization, copy, move, assignment, destruction.*

```

class MyClass
{
public:
    // constructor funcs
    // same name as class
    // no return type

    // default constructor
    // constructor initializer
    // copy constructor

private:
    // in-class initializer
    // = for copy initialization
    // braces for direct initialization
}

```

Definition 9.9 (constructor initializer).

```

// initialize nonstatic data members
// automatically called when an obj is created
MyClass() {} // default constructor
// provide a default constructor if other constructors are defined
MyClass(params) member(expr), ... { ... } // constructor initializer list

// use default constructor to define obj
MyClass myClass; // error MyClass myClass(); to define func

// constructor initializer list specifies only the value used to initialize members
// members are initialized in the order in which they appear in class definition
// write constructor initializer in same order as members are declared
// avoid using members to initialize other members

```

```
// delegating constructor
// use other constructor to perform its initialization
MyClass(params) : var1(expr), ... {} // nondelegating constructor
// delegated-to constructor initializer list and func body are both executed
MyClass() : MyClass(arguments) {} // default constructor
MyClass(params) : MyClass() { ... } // delegate to default constructor
```

Definition 9.10 (converting constructor). *A constructor that can be called with a single argument defines implicit conversion from constructor's parameter type to class type.*

```
// implicit class-type conversion
MyClass(param) {}
type func(MyClass);
// implicit conversion from parameter type to class type
MyClass myClass = argument; // copy initialization
std::string s = "str"; // literal to string
func(argument); // during func call

// explicit constructor
// prevent implicit conversion by declaring constructor as explicit inside class
// be meaningful only on constructors that can be called with a single argument
// only direct initialization is allowed
explicit MyClass(param) {}

// explicit conversion
MyClass(argument);
std::string("str");
```

Definition 9.11 (constexpr constructor).

```
// literal class
// all data members are literal type
// constexpr constructor funcs
// must initialize every data member with constant expression or constexpr constructor
// body is empty
constexpr MyClass(params) : var(expr), ... {}
```

Definition 9.12 (copy control). *Copy control members include copy constructor, copy-assignment operator, move constructor, move-assignment operator, and destructor.*

Definition 9.13 (copy constructor).

```
// first param is a reference to class type and any additional parameters have default values
// memberwise copy the members of argument into the object being created
// copy each nonstatic member from given object into the one being created
MyClass(const MyClass& orig) : var(orig.var), ... {}
```

```
// synthesized copy constructor
```

```
// direct initialization uses constructor initializer
// copy initialization uses copy constructor
```

Definition 9.14 (copy-assignment operator).

```
// overloaded assignment operator function
// usually return a reference to left-hand operand object
// when an operator is a member function,
// the left-hand operand is bound to implicit this parameter
MyClass& operator=(const MyClass &rhs) { var = rhs.var; ... return *this; }
```

```
// synthesized copy-assignment operator
```

Definition 9.15 (destructor). *Destructor free the resources used by an object and destory the nonstatic data members.*

```
// no parameters and return value
// there is only one destructor that cannot be overloaded

// default synthesized destructor
~MyClass() {}
```

Definition 9.16 (move constructor).

Definition 9.17 (move-assignment operator).

9.3 Operator

Definition 9.18 (overloaded operator). *Overloaded operators are functions that have the name operator followed by the symbol for the operator being defined.*

```
// an operator function has a return type and a parameter list
```

9.4 OOP

Definition 9.19 (OOP). *Object-oriented programming is based on three basic concepts:*

- *data abstraction separates interface from implementation*
- *inheritance model the relationships among similar class types*
- *dynamic binding use objects of these class types*

Definition 9.20 (inheritance). *Classes related by inheritance form a hierarchy.*

```
// base class at the root of the hierarchy
class BaseClass
{
public:
    virtual type func(parmas);
}

// derived class
class MyClass : public BaseClass // class derivation list
{
public:
    type func(parmas);
}
```

Definition 9.21 (dynamic binding). *Dynamic binding happens when a virtual function is called through a refernece or pointer to a base class.*

```
// run-time binding
```

9.5 Template

Definition 9.22 (template). *A template is a blueprint or formula for creating classes or functions.*

Definition 9.23 (function template). *A function template is a formula from which we can generate type-specific versions of that function.*

Definition 9.24 (class template). *A class template is a blueprint for generating classes.*

```
template <typename T> class MyClass {}
```

10 Sequential Containers

Definition 10.1 (container). *A container holds a collection of objects of a specified type.*

Definition 10.2 (sequential container). *A sequential container controls the order in which the elements are stored and accessed.*

```
// vector

// string

// array: fixed size
// does not support operations to add or remove elements or to resize the container

// list

// forward_list

// deque
```

11 Generic Algorithms

12 Associative Containers

13 Dynamic Memory

Definition 13.1 (memory).

- *static memory is used for local static objects, for class static data members, and for variables defined outside any function.*
static objects are allocated before they are used and destroyed when the program ends.
- *stack memory is used for nonstatic objects defined inside functions.*
stack objects exist only when the block is executing.
- *free store or heap dynamically allocate at run time.*
heap objects must be explicitly destroyed when they are no longer needed.

```
// dynamic memory operators
// new: allocates and optionally initialize an object in dynamic memory
// and returns a pointer to that object

// delete: destroys that object and frees memory

// problems:
// forget to free memory -> memory leak
// free memory when there are still pointers referring to that memory ->
// pointer refers to invalid memory
```

Definition 13.2 (smart pointers).

```
// std::shared_ptr
// allows multiple pointers to refer to same object

// std::unique_ptr
// owns the object to which it points

// std::weak_ptr
// a weak reference to an object managed by a std::shared_ptr
```

Definition 13.3. `shared_ptr`

```
// smart pointers are class templates
// must supply the type to which the pointer can point

std::shared_ptr<type> p; // default initialized smart pointer holds a null pointer

// std::make_shared
// allocates and initializes an object in dynamic memory
// and returns a std::shared_ptr that points to that object
std::shared_ptr<type> p = std::make_shared<type>(argument);
// std::make_shared uses its arguments to construct an object of given type
auto p = std::make_shared<type>(argument);

// reference count
// the counter is incremented when we use it to initialize other std::shared_ptr,
// use it as assignment, or pass it to or return it from function by value
// the counter is decremented when we assign a new value to std::shared_ptr
// and when std::shared_ptr itself is destroyed when going out of scope
// when the counter goes to zero,
// std::shared_ptr destructor automatically destroyed the object
```

A Tools

Definition A.1 (namespace). *using declaration let us use a name from namespace without qualifying the name with `namespace::` prefix.*

```
using namespace::name;
```

Definition A.2 (comment).

```
// single line
/* pair cannot nest */

std::cout << /* "/" /* "/" */; // std::cout << " /* ";
// 1. from right to left
// 2. find */, remove everything between */ and left /*
// 3. repeat step 1-2
```

Example A.1 (resources).

- <https://www.learncpp.com/>

References

- [1] S.B. Lippman, J. Lajoie, and B.E. Moo. *C++ Primer*. Addison-Wesley Professional, 2012. ISBN: 978-0-321-71411-4.
- [2] Shapira Yair. *Solving PDEs in C++*. 2nd Edition. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2012. URL: <https://epubs.siam.org/doi/abs/10.1137/1.9781611972177>.