



Classes

class classname: defines a class

public classname: inherits a class leaving scope of members intact

private classname: inherits a class making all members private

protected classname: inherits a class making public members protected

virtual public classname: inherits a virtual base class. Classes share one copy of the base class

private:: member access inside class only

type variable: define private members

type functionname (arguments){commands};

public:: member access outside the class

type variable: define public members

classname(type variable=value,...): constructor

classname()**=default**: specify default constructor

classname(const classname&): copy constructor

classname(const classname&)**=delete**: no default copy constructor

classname& **operator=** (const classname&): assignment operator for a class

classname& **operator=** (const classname&)**=delete**: no default assignment constructor

~classname(): destructor, always to be used when working with dynamic arrays

classname(type variable,...):privatevariable {variable}: constructor assigning values to private

explicit classname(type variable=value,...): constructor where type has to be explicit correct

friend class classname: classname has access to the private members of the class

friend classname **operator**operator (arguments) {commands; **return** value;}: friend operator

```
friend Stock operator*(int d,const Stock& s) {
    return Stock(d*s.small, d*s.big); } // d * s
```

operator stdtype() **const** {**return** value;}: converts a class to a standard type

```
operator int() const {return s.small;}
```

type functionname (arguments){commands};

type functionname (arguments) **const** {commands;}: function does not change private members of the class

type functionname (arguments) {commands;}

override: overrides a virtual function in baseclass

type classname::**operator**operator(arguments) {commands; **return** value;}: operator overloading

virtual type functionname (arguments) {commands;}: in baseclass for pointers or references to the baseclass, will call appropriate function in derived classes with overridden functions

```
Person *p, *s, *e; //pointers to base class
s = new Student; //Student is a derived class Person
s -> to_string(); //call to_string from Student class
```

protected:: member access for inherited classes

type variable: define protected members

type functionname (arguments){commands;}

Directives

#define name: conditional compilation name

#ifndef name commands; **#endif**: conditional block

#include <name>: includes header file

#include "name": includes header file

// comment: includes single line comment

/* comment */: includes multi line comment

using namespace::class: default scope resolution

Exceptions

#include <exception>

exception.what(): returns a readable error string

throw: re-throws an exception in nested functions

throw int: throws an exception with type int

throw classname(arguments): throws class type

try { commands; } catch(exception) {commands;}

catch(exception){commands;}: try commands, if an exception of any type is raised execute **commands**. Common exception examples:

```
std::bad_alloc //memory full
classname variable //can use variable in commands
std::string s // with throw std::string("error")
std::exception& exc //can use exc.what()
... //catch all other exceptions
```

Functions

type functionname (): function that returns a type

type& functionname (): function that returns a reference to a type, if an lvalue is needed

type* functionname (): function that returns an address to a type

void functionname (): function without arguments

void functionname (type variable,...): function with arguments as value (no intention to change the value of the arguments in the function)

void functionname (type variable[][int]): argument is a 2 dimensional array, needs dimensions after the first dimension since memory is a flat model

void functionname (type& variable,...): function with arguments as reference (use if you want to change the argument in the function)

void functionname (const type& variable,...): argument as a const reference, saves stack memory

void functionname (type variable=value): sets a default value for arguments

void (*functionname)(type): a pointer to a function with one argument of type type

```
int sum(int x,int y);
int product(int x, int y);
int calc(int a,int b, int(*f)(int,int));
calc(1,2,product); //function pointer as argument
```

Initialization

delete[] variable: deletes a dynamic array

this: pointer to current object in a class

***this**: current object in a class

type* variable=**new** type[int]: makes a dynamic (at runtime) array of int elements

type* variable=**nullptr**: pointer to address 0

type variable {value}: uniform initialization

static type variable=value: the variable initializes once to value and keeps its changing value when entering the process block it was defined

Iterators

class classiterator {: make a custom iterator class

private: node* pointer;

public:

classiterator(node* init=nullptr):pointer{init}

int& **operator***() {**return** pointer->element();}

classiterator& **operator++**() {**return** nextpointer}

bool operator!=(classiterator iter){**return** pointer != iter.pointer;}

std::string::iterator variable: pointer to chars

```
std::string s {"abcd"};
std::string::iterator pos;
for(pos=s.begin(); pos!=s.end(); ++pos) {};
```

std::vector<type>::iterator variable: vector pointer

Lambda functions

[capture list]: variables to capture for the function

```
[a] //captures variable a by copy
[&b] //captures variable b by reference
[=] //captures all variables by copy
[&] //captures all variables by reference
[] //don't capture any variable
[this] //capture this pointer of current object
```

(parameter list): parameters for the function

mutable: optional keyword in order to change captured variables by copy in the function

->return type: define a return type for the function

```
{commands;}
```

```
{factor}(double x)->double {return x*factor;}
{factor}(int&x)mutable{x*=factor;factor++;}
```

Manipulators

#include <iomanip>

std::dec: sets decimal value of integer

std::fixed: fixed notation of floating point

std::hex: sets hexadecimal value of integer

std::left: aligns the text left

std::oct: sets octal value of integer

std::right: aligns the text right

std::scientific: e-notation of floating point

std::setfill(char): sets fill character

std::setprecision(int): sets number of decimals

std::setw(int): sets width

std::showpoint: shows the decimal point

Modifiers

type* **const** variable: the pointer is constant

const: makes a value constant for the compiler

const type* variable: a const pointer to a constant

const type* **const** variable: const pointer to const

constexpr: makes an expression constant

Namespace std

std::cin: gets characters from the input buffer

std::cout: sets characters to the console output

Operators

&&: logical AND

&variable: returns the address of the variable

||: logical OR

!: logical NOT

!=: does not equal to

==: is equal to

=: assignment operator

+: sums the operands

++: increment by 1

```
int i=10;
int n=++i; //prefix n gets the value 11
int m=i++; //postfix m gets the value 10
int* j;
j++; //moves the pointer sizeof(int) bytes
```

-: subtracts the operands

--: decrements by 1

->: call member function of a pointer

*****: multiply the operands

/: divides the operands



%: modulo by division of the operands
<: smaller then
<=: smaller then or equal
<<: sends characters to the function
>: bigger then
>=: bigger then or equal
<=>: spaceship operator compares all attributes of objects for find or sort algorithms
>>: gets characters from the function
[]: collection index operator
(): function call operator
dynamic_cast<type>(variable): casts at runtime
reinterpret_cast<type>(variable): forces conversion to a specific type via binary format
sizeof(type): returns the size in bytes of the type
static_cast<type>(variable): typecasting at compile time

Smart pointers

#include <memory>: header for smart pointers
std::shared_ptr<type> variable: reference counted pointer, when all pointers that reference a memory block go out of scope the memory block is released
std::weak_ptr<type> variable: defines a unique pointer, when pointer goes out of scope memory block is released
std::weak_ptr<type> variable: only to be used when you have a circular list of pointers, connect the last pointer with a weak_ptr to the first one

Standard library

#include <algorithm>: uses algorithms
std::find(start, end, value): iterator to position
std::find_if(container, predicatefn): predicate fn
bool pair(const int& n) {return n%2==0;}
pos=std::ranges::find_if(v, pair);
std::iota(start, end, value): fill container with range
std::iota(v.begin(), v.end(), 3); //v={3,4,5,6,...}
#include <ranges>: uses ranges functions
std::ranges::copy(range, start): copies containers
std::ranges::copy(original, destination.begin());
std::ranges::copy(range, inserter): uses inserters
std::ranges::copy(org, std::back_inserter(dest));
std::ranges::copy(org, std::front_inserter(dest));
std::ranges::copy(org.begin(), org.end()+5, std::inserter(dest, dest.begin()));
std::ranges::for_each(start, end, function): applies function from start to end iterators
std::ranges::for_each(container, function)
std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
void print(int x) {std::cout<<x<<' '};
std::ranges::for_each(v, print);
std::ranges::for_each(container, function, projection): projection is a object attribute
std::ranges::for_each(v, print, &Person::name);
std::ranges::sort(container): sorts a collection
std::ranges::sort(container, comparator): compare
bool compnr(const Person& p1, const Person& p2) {return p1.getnr() < p2.getnr();}
std::ranges::sort(v, compnr);
std::ranges::sort(container, comparator, projection): see comparator and projection before
std::views::drop(int): drops first int elements
std::views::iota(start, end): generates int range
std::views::filter(predicatefn): filter using predicate
std::views::take(int): take first int elements
std::views::transform(function): applies function
auto square=[](auto x){return x*x};
auto v_square=std::views::transform(v, square);
auto v2_square=std::views::transform(v_square); use
#include <deque>: a double ended queue

#include <list>: linked list with two pointers
std::list<std::string> animals;
animals.push_back("Cat"); //pop_back
animals.push_front("Dog"); //pop_front
animals.unique(); //preserves unique values
animals.merge(humans); //merges two lists
#include <stack>: a LIFO stack list
std::stack<int> s;
s.push(123);
std::cout << s.top();
s.pop();
#include <queue>: a FIFO stack list
Statements
condition ? truecommand:falsecommand
break: breaks a loop statement block
continue: exits body, but continuous loop itself
do {commands;} **while** (condition)
for (type variable: container) {commands;}: a range based for
for (type& variable: container) {commands;}: a range based for for referencing the variable
for (initialization; condition; raise) {commands;}
if (condition) {commands;} **else** {commands;}
switch (variable) {case value: commands; **break**; default: commands;}
typedef type name: gives type another name
while (condition) {commands;}
Streams
#include <iostream>: header for standard streams
std::cin >> c; //standard input
std::cout << c; //standard output
std::cerr << c; std::clog << c; //error logs
#include <fstream>: header for file streams
//write to a file
std::ofstream out("output.txt");
std::ofstream out{"output.txt", ios::app}; //append
std::ofstream out{"output.txt", ios::trunc}; //clear
std::ofstream out{"out.txt", ios::noreplace}; //newfile
//read from a file
std::ifstream in("input.txt");
std::ifstream input; input.open("input.txt");
//read and write to a file
std::fstream file("readwrite.txt", ios::in|ios::out);
//check for errors
if (!in || !out) {error};
//using std::ranges::copy
std::vector<int> v {1,2,3,4,5,6,7,8,9};
std::ranges::copy(v, std::ostream_iterator<int>(out));
std::ranges::copy(std::istream_iterator<int>(in), std::istream_iterator<int>(), back_inserter(v));
//using operators
out << "string test" + '\n';
std::string s;
while (in) { // while (!in.eof()) {
std::getline(in, s); //getline for strings, not >>!
if (in.fail()) {wrong information in file};
//random access
out.seekp(0); //put-pointer to position 0
in.seekg(0); //get-pointer to position 0
//binary files
std::ofstream bin("binary.txt", ios::binary);
bin.put("a"); //saves one byte
bin.write(reinterpret_cast<char*>(&Cclass), sizeof(Cclass)); //writes block, not for dynamic memory
std::ifstream bin("binary.txt", ios::binary);
bin.get(c); //reads one byte
bin.read(reinterpret_cast<char*>(&Cclass), sizeof(Cclass)); //reads block, not for dynamic memory
std::cout << bin.rdbuf(); //pointer to buffer file
//close file handles
out.close(); in.close();
#include <sstream>: header for string streams
std::ostringstream os;
os << "test" << i << " integer, double: " << d;
std::string s = os.str();

Streams

#include <iostream>: header for standard streams
std::cin >> c; //standard input
std::cout << c; //standard output
std::cerr << c; std::clog << c; //error logs
#include <fstream>: header for file streams
//write to a file
std::ofstream out("output.txt");
std::ofstream out{"output.txt", ios::app}; //append
std::ofstream out{"output.txt", ios::trunc}; //clear
std::ofstream out{"out.txt", ios::noreplace}; //newfile
//read from a file
std::ifstream in("input.txt");
std::ifstream input; input.open("input.txt");
//read and write to a file
std::fstream file("readwrite.txt", ios::in|ios::out);
//check for errors
if (!in || !out) {error};
//using std::ranges::copy
std::vector<int> v {1,2,3,4,5,6,7,8,9};
std::ranges::copy(v, std::ostream_iterator<int>(out));
std::ranges::copy(std::istream_iterator<int>(in), std::istream_iterator<int>(), back_inserter(v));
//using operators
out << "string test" + '\n';
std::string s;
while (in) { // while (!in.eof()) {
std::getline(in, s); //getline for strings, not >>!
if (in.fail()) {wrong information in file};
//random access
out.seekp(0); //put-pointer to position 0
in.seekg(0); //get-pointer to position 0
//binary files
std::ofstream bin("binary.txt", ios::binary);
bin.put("a"); //saves one byte
bin.write(reinterpret_cast<char*>(&Cclass), sizeof(Cclass)); //writes block, not for dynamic memory
std::ifstream bin("binary.txt", ios::binary);
bin.get(c); //reads one byte
bin.read(reinterpret_cast<char*>(&Cclass), sizeof(Cclass)); //reads block, not for dynamic memory
std::cout << bin.rdbuf(); //pointer to buffer file
//close file handles
out.close(); in.close();
#include <sstream>: header for string streams
std::ostringstream os;
os << "test" << i << " integer, double: " << d;
std::string s = os.str();

Strings

#include <string>: header for std::string class
string variable {}: defines a string variable
.find(string): returns position of string
.length() returns the number of chars in a string
.replace(int, int, string): replaces int chars with

string from position int
.substr(int, int): returns int chars from position int
string variable(int, char): initializes int times char

Templates

template<typename T, ...>: template prefix with template argument T, the yet unknown type
template<typename T=type>: define a default type
T functionname(T arguments): function template
classname<T>::classname(T arguments): a class template of type T with a constructor using T
classname<T>::function(): define class T function

Types

type variable[int]: defines an array of int positions
type variable[int][int]: a int * int array (2 dim)
type* variable: defines a pointer to a type
auto: compiler auto defines type when possible
bool: Boolean value
char: one ASCII character
decltype(variable) **variable**: type **variable**=variable
double: numeric value from 1.7e-308 to 1.7e+308
enum class variable {value, value, ...}: enumerations class, use with variable::value
float: numeric value from 3.4e-38 to 3.4e+38 (eg 3.14159F)
int: numeric value from -2147483648 to 2147483647 (eg 0x1A, 0b1000)
long: numeric value from -9223372036854775808 to 9223372036854775807 (eg 32L)
long double: numeric value from 3.4e-4932 to 3.4e+4932
long long: numeric value from -9223372036854775808 to 9223372036854775807 (eg 32LL)
short: numeric value from -32768 to 32767
unsigned type: only positive numeric values (eg 2U)

Vector

#include <vector>: header for std::vector class
std::vector<type> {}: variable: a vector collection
.capacity() returns the capacity of the vector
.insert(int, value): inserts value at position int
.pop_back() removes last element
.push_back(value): add element to the back
.reserve(int): reserves capacity
.resize(int): resizes the vector (grow only)
.size() returns the size of the vector