Inhalt

[C++ NOTES 3](#_Toc179457409)

[C++ Compilers: 3](#_Toc179457410)

[Compile Frontend 3](#_Toc179457411)

[Compiler backend 3](#_Toc179457412)

[Compile Components and Compile Time: 4](#_Toc179457413)

[Run-Time: 4](#_Toc179457414)

[Main Method: 4](#_Toc179457415)

[Operators: 5](#_Toc179457416)

[Arrays: 6](#_Toc179457417)

[Pointers: 6](#_Toc179457418)

[Smart Pointer: 7](#_Toc179457419)

[Shared Pointer: 7](#_Toc179457420)

[Reference: 7](#_Toc179457421)

[Auto Type Deduction 8](#_Toc179457422)

[if Statement: 8](#_Toc179457423)

[Switch 8](#_Toc179457424)

[For loop 9](#_Toc179457425)

[while loop 9](#_Toc179457426)

[Do statement: 9](#_Toc179457427)

[Qualifier constants : 10](#_Toc179457428)

[Qualifier constexpr: 10](#_Toc179457429)

[std::string: 10](#_Toc179457430)

[Functions: 11](#_Toc179457431)

[Operators new[] and delete[] 12](#_Toc179457432)

[ENUM 12](#_Toc179457433)

[Unscoped ENUM: 12](#_Toc179457434)

[Scoped ENUM: 13](#_Toc179457435)

[Namespace 13](#_Toc179457436)

[Organizing code in header and source files 14](#_Toc179457437)

[Via Namespace: 14](#_Toc179457438)

[Via header and source: 14](#_Toc179457439)

[Exception: 15](#_Toc179457440)

[Logic Errors: 15](#_Toc179457441)

[Runtime errors 16](#_Toc179457442)

[bad\_alloc 16](#_Toc179457443)

[Example of Stack Unwinding: 18](#_Toc179457444)

# C++ NOTES

**Introduction to the C++ Programming Language**

05-06. Sep. 2024

12-13. Sep. 2024

19-20. Sep. 2024

FAU@Erlangen

Instructor: **Slobodan Dmitrovic**

C++ code files have extensions .cpp

C++ header files have extensions .h or .hpp

## C++ Compilers:

Compiler is C++ program which compile .cpp source files into object files. Linker links those object files and product an executable file or a library. (to make a library we must change our project setting in Visual Studio). A C++ library contains object file and header file. Source file usually do not include in to library.

A C++ Compiler has two parts.

1. Compiler Frontend and
2. Compiler Backend

Compile Frontend read and verify the syntax according to c++ rules and create object file. If there is any syntax mistake or warning, compile frontend gives compiler error and show it in the IDE and ask the developer to resolve it. Once all compiler errors are removed, compile frontend generate object file. For warning there are different options in compile frontend, we can suppress warning, or we can also ask compiler to take warning seriously and stop compile if a warning occurs. Usually this produces very high-quality code without any warning or error. These warning and syntax checks are part of compile frontend program. There are many Compiler Frontend in the market.

Some of the compiler frontends are:

1. g++ compiler frontend (as part of the GCC)
2. Visual C++ or MSVC (as part of Visual Studio IDE)
3. Clang or clang++ compiler frontend (as part of LLVM)

Compiler backend are also C++ Program which takes these object files and creates machine code against different hardware structure like X86, ARM and now also for RISC-V (new architecture).

In case of C ++ is shift to another hardware, it does not require any change in cpp source file. That means the same object file which gots created by Compiler Frontend, can be used to generate a new machine code via Compiler Backend.

Source code

Source code

Source code

Compiler Frontend code

obj. File

X86 Compiler backend code

ARM Compiler backend code

RISC-V Compiler backend code

X86 machine code File

arm machine code File

Risc-v machine code File

### Compile Components and Compile Time:

Preprocessor (Combining different source files (as added via #include <> in your main file into a single file)

Compilation (Production of object file)

Linkage (link object file to create an executable file)

Total to of running above 3 components is called Compile Time.

### Run-Time:

The time with the loading or execution of program or a library and end with the termination of our program or a library.

### Main Method:

In earlier times, (I think from c++ 11 or earlier), an integer value always returns to give information about successful running / termination of our program. This integer-based return value is still there, we do not pass any return value specifically. I think the compiler return itself a Zero 0 at the end of main method body.

#include <iostream>

int main()

{

Std::cout << “hello World” << ‘\n’;

}

DataTypes:

Char, short, int, float, long, double, longlong

|  |  |  |  |
| --- | --- | --- | --- |
| Datatype | Size (bytes) | Range | Initialization example |
| char | 1 |  | char c { ‘A’ };  char c = ‘A’; |
| int | 4 *( 64 bits system)*  2 *(32 bits system)* |  | int x = 123;  int x {123};  int x {012}; // octal literal  int x {0xA}; // hexadecimal literal |
| double | 4 and 8 *(32 / 64 systems)* |  | double d = 456.789;  double d { 456.789}; |

Type modifiers:

Affect the signedness and size of the data type.

Int x = 123456; (take both negative and positive value within its range)

unsigned y = 4294967295 (only zero and positive value)

unsigned long int z = 4294967295;

char: 1 byte

bool: 1 byte

short: 2 bytes

int: 4 bytes

float: 4 bytes

long: 8 bytes

double: 8 bytes

### Operators:

|  |  |
| --- | --- |
| Assignment Operators | = to assign a value to a variable/object |
| Arithmetic Operators | + // addition  - // subtraction  \* // multiplication  / // division  % // modulo |
| Increment and decrement Operators | ++a // pre-increment operator  a++ // post-increment operator  --a // pre-decrement operator  a-- // post-decrement operator |
| Compound assignment Operators | += // compound addition  -= // compound subtraction  \*= // compound multiplication  /= // compound division  %= // compound modulo |
| Logical operators | !a // logical NOT  a && b // logical AND  a || b // logical OR |
| Comparison Operators | a == b  a != b  a < b  a <= b  a > b  a >= b |
| Member Access operator | **a.b** Member of an Object Operator  **a->b** Member of a pointer operator  **a[b]** Subscript operator  **\*a** Indirection / dereferencing |

Standard I/O

Int a = 555;

std::court << “this message goes to the left and print on the scream via std::cout stream”;

std::cout <<< “a = “ << a ; // it will print value of a.

a << std::cin; // it will take some input from user via keyboard and store into variable a. take extra care to the datatype of variable a. it is of int type.

### Arrays:

int arr[5]; // declare an array of 5 integers.

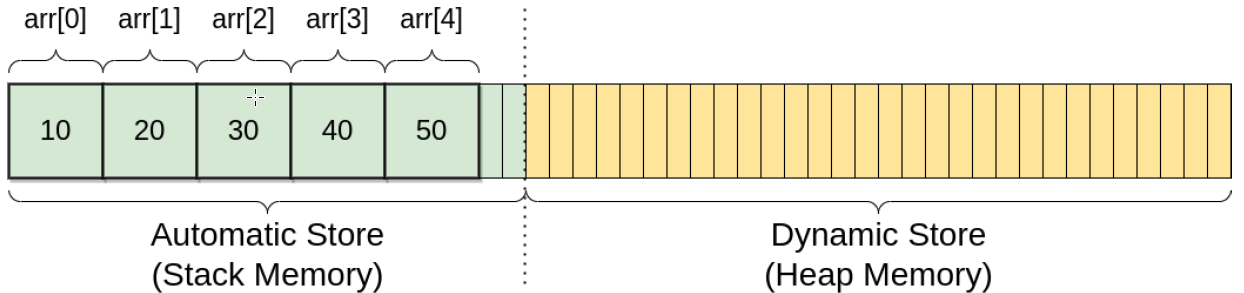
int myarr[5] = { 10, 20, 30, 40, 50 }; //initialize an array of 5 el

to access use subscript operator.

std::cout << arr[0]; // will print value at index 0.

Array size cannot be **changed,** and it is **fixed**. We cannot **insert to** or **delete from** an array.

Array elements are positioned sequentially in memory.



### Pointers:

Pointer hols the address of a particular object. We use address of & operator.

int x = 123;

int\* p = &x; // p will store the address of variable x in memory

char c = ‘A’;

char\* p = &c; // p will store the address of variable c in memory

\*p = b; // it will go in memory and change the value of variable c

std::cout << c; //it will print b

std::cout << \*p; //it will print b. it is called pointer dereferencing.

delete p;

The above example is a raw pointer. This has a disadvantage that raw pointer does not automatically destroy and do not deallocate the memory once they go out of scope. Therefore, we have to delete it manually otherwise it will take space in memory even after it gets out of scope.

### Smart Pointer:

Smart pointers own the object they point to and automatically destroy the object and de-allocate the memory once they go out of scope. Smart pointer. Smart pointers are implemented using templates and to use them, user has to include <memory. Following are few examples:

std::unique\_ptr<int> up (new int { 123 } ) ; // object is created on heap.

std:: cout << \*p; // will print the memory address.

Another way (a better way) to define smart pointer is to create them by make\_unique.

std::unique\_ptr<int> up = std::make\_unique<int>(123);

std::unique\_ptr<double> up = std::make\_unique<double>(123,456);

### Shared Pointer:

We can have multiple pointers pointing to a single object in memory. The memory gets deallocated only when all shared pointers pointing to this same location get out of scope.

std::shared\_ptr<int> sp1 = std::make\_shared<int(123);

std::shared\_ptr<int> sp2 = sp1;

std::shared\_ptr<int> sp3 = sp1;

**SHARED Pointer can be copied. But UNIQUE Pointers can only be moved**. Therefore, for a unique pointer up1 following copy assignment is illegal (compile time error)

std::unique\_ptr<int> up1 (new int { 123 } ) ;

std::unique\_ptr<int> up2 = up1; // compile time error

std::unique\_ptr<int> up2 = std::move(up1) // will work

### Reference:

To variable sharing the same object in memory. Changing one will change other.

int x = 123;

int& y = x; // y is a reference to x

x = 456; // both x and y now hold the value of 456

y = 789; // both x and y now hold the value of 789

QUIZ:

int x = 10;

const int& y = x;

x = 5; // guess what will happen ?

### Auto Type Deduction

use auto specifier when the type name is:

1. - hard to deduce
2. - hard to type manually

**auto var\_name = initializer;**

auto c = 'a'; // char type

auto x = 123; // int

auto d = 456.789; // double

auto d = 123.456 / 789.10; // double

int x = 123;

auto& y = x; // y is of int& type

const auto x = 123; // y is of const int type

### if Statement:

bool b = true;

if (b) std::cout << "Condition true.";

bool b = true;

if (b) { std::cout << "I am true."; } // will print I am true.

bool b = false;

if (b) { std::cout << "I am true.";

} else { std::cout << "I am false."; } // will print, I am false.

**Conditional operator ?:**

Bool thecondition = true;

int x = 0;

int x = (thecondition) ? 10 : 20; // x is now equal to 10

Switch:

int x = 2;

switch (x)

{

case 1:

std::cout << "The value of x is 1.";

break;

case 2:

std::cout << "The value of x is 2.";

break;

default:

std::cout << "None of the above.";

break;

}

### For loop

for (init\_statement; condition; iteration\_expression)

{

// statements

}

The for-statement executes the init\_statement only once. Then, the condition is evaluated before every loop. If the condition is true, the statements inside the for-loop body are executed, and finally, the iteration\_expression is evaluated before the end of each loop.

#include <iostream>

int main()

{

for(inti = 0; i < 10; i++)

{

std::cout << "The counter is: " << i << '\n';

}

}

### while loop

while (condition)

{

// statements

}

#include <iostream>

int main()

{

int i = 0;

while (i < 10)

{

std::cout << "The value of i is: " << i << '\n';

i++;

}

}

### Do statement:

The do-statement is similar to the while-statement, but the condition comes after the body.

do

{

// statements

} while (condition);

code inside the do-while loop is guaranteed to execute at least once.

#include <iostream>

int main()

{

int i = 0;

do

{

std::cout << "The value of i is: " << i << '\n';

i++;

} while (i < 10);

}

### Qualifier constants :

const int n = 5; // now n can not be modified,It is defined as const

const int n; // error, no initializer

const int m = 123; // OK

const int n = 5;

n++; // error, cannot modify a read-only object

The const modifies an entire type. For example, const int and int are two different types.

## Qualifier constexpr:

Another const qualifier is the constant expression named constexpr. It is a constant that can be evaluated at compile-time. Initializers themselves must be constant expressions.

constexpr int n = 123; // OK

constexpr double d = 456.78; // OK

constexpr double d2 = d; // OK

int x = 123;

constexpr int n2 = x; // error, x is not a constant expression

## std::string:

Type:

#include <string>

std::string s = "Hello ";

char c = 'W';

s += c; // adding a character

s += "or"; // adding a string literal

std::string s2 = "ld.";

s += s2; // adding another string

std::string s = "Hello World.";

charc1 = s[0]; // 'H'

charc2 = s.at(0); // 'H';

|  |  |
| --- | --- |
| Comparing two strings | If (s1 == s2) {std::cout << “equal”; |
| To get pointer to the first element of a string | std::string s1 = “Hello”;  std::unique\_pointer<std::string> sp1 = s1.c\_str(); |
| Getting a string from user via getline | std::getline(std::cin, s1) |
| .substr | std::string s1 = “Hello”;  std:string substr = s1.substr(6,5) // from position 6 to length 5 |

## Functions:

Overloaded Functions declaration:

void myFunction();

int myFunction();

int myFunction(int);

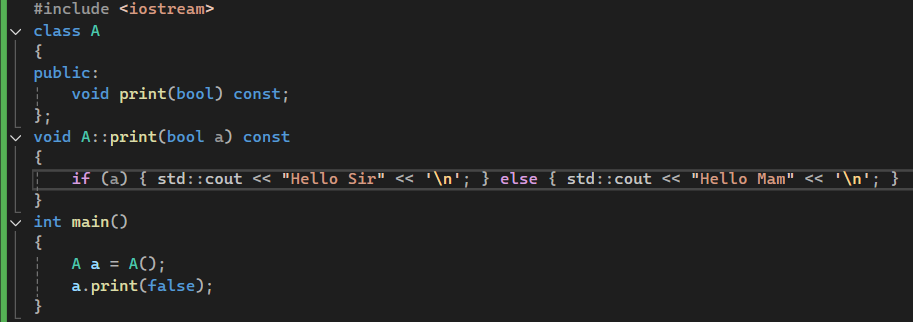
int myFunction(int x);

double myFunction(double, double);

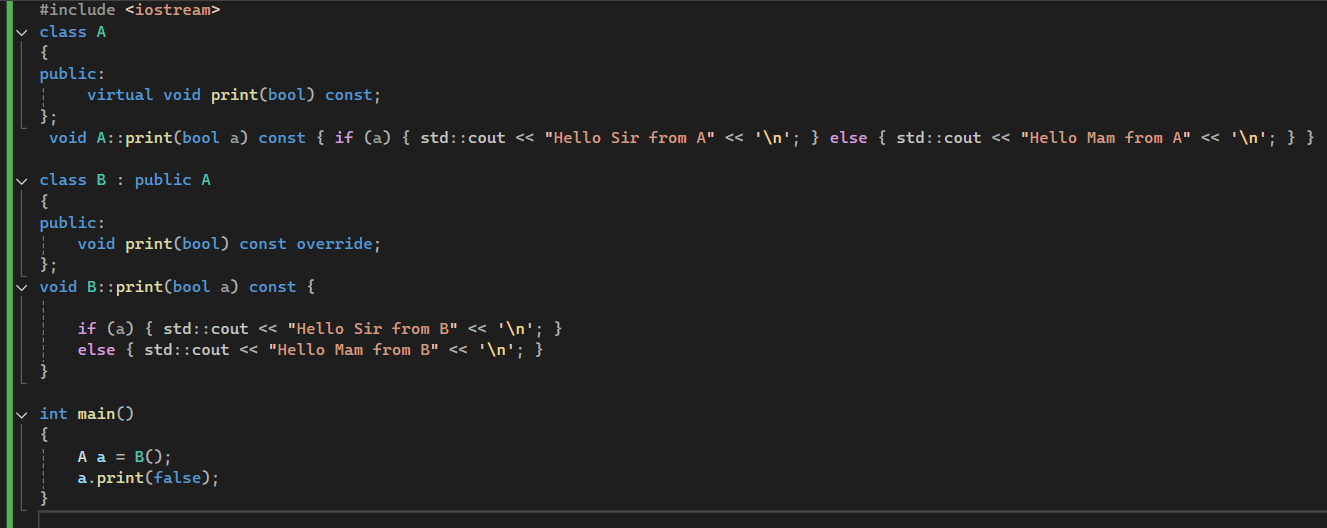
double myFunction(double x, double y);

|  |  |
| --- | --- |
| Function passing an arg. by value | void myFunction(int byvalue) |
| by reference. | void myFunction(int& byreference) |
| by const reference. | void myFunction(const std::string& byconstreference) |
| Passing a pointer | ? |

If a function is declared as class member then it can be defined as inside class or outside class. But defining the function outside the class would require to calls it via class name. const show that this function does not do any data modification.



If this function belongs to an abstract class and it has to be implemented in child classes, then we have to add virtual in the beginning of function declaration in base class add override in child class function declaration.



**Scope and Storage Duration:**

Scope can be defined via {} brackets. For

**Automatic Storage Duration:**

For variable which are created on stack and end when their scope ends

**Dynamic Storage Duration:**

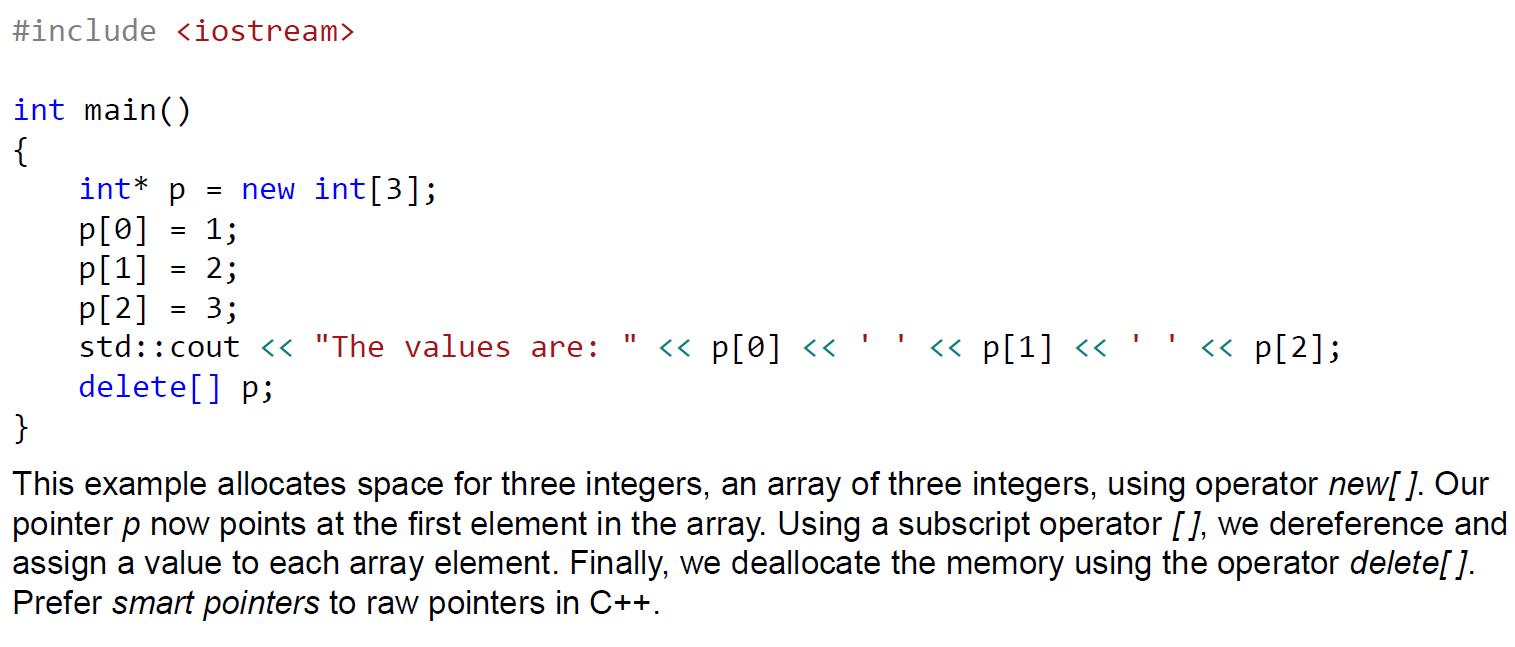
For variable which are created on heap and have to manually de-allocated.

**Static Storage Duration:**

For static variables, which after creation resides in the memory until the program ends.

## Operators new[] and delete[]

If we want to allocate memory for an array, we use the operator new[ ]. To deallocate a memory allocated for an array, we use the operator delete[ ]. Pointers and arrays are similar and can often be used interchangeably. Pointers can be dereferenced using a subscript operator [ ].



ENUM:

Enumerations are user-defined types whose values are constants with symbolic names.

### Unscoped ENUM:

Unscoped enums have underlying integral values and leak their enumerators into the surrounding scope.

enum MyEnum

{

myfirstvalue = 10,

mysecondvalue,

mythirdvalue

};

enum MyOtherEnum

{

myintvalue = 10

};

int main()

{

MyEnum myenum = myfirstvalue;

myenum= mysecondvalue;

}

### Scoped ENUM:

Scoped enums do not leak their enumerators, and we can specify the scoped enum's underlying type. We should prefer scoped enums in C++.

enum class MyEnum

{

myfirstvalue,

mysecondvalue,

mythirdvalue

};

enum MyCharEnum : char // specify the type

{

myintvalue = 'A'

};

int main()

{

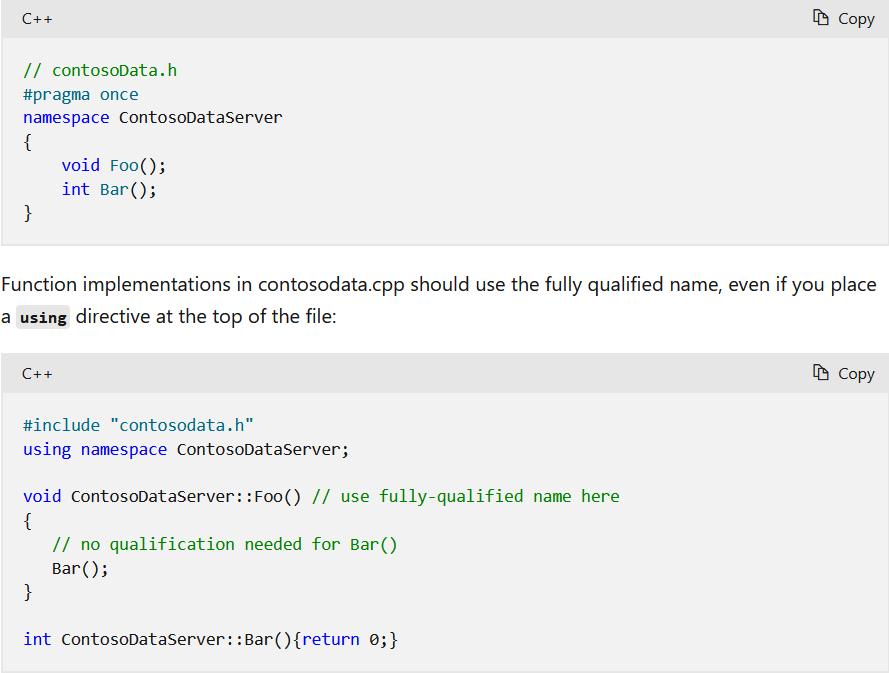
MyEnum myenum = MyEnum::myfirstvalue;

}

erer

Namespace:

Defined usually in Header file and used in cpp file with using directive.



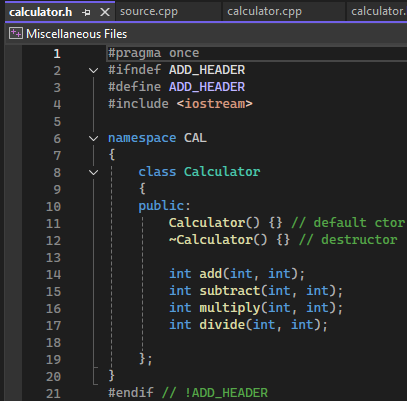
## Organizing code in header and source files

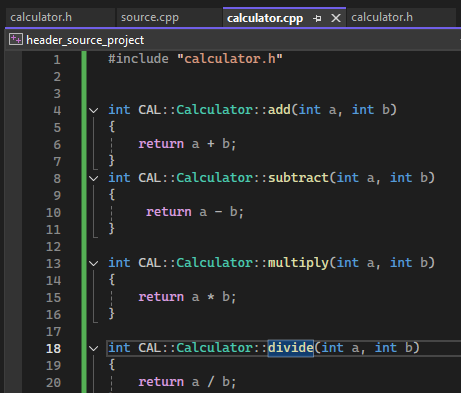
Via Namespace:

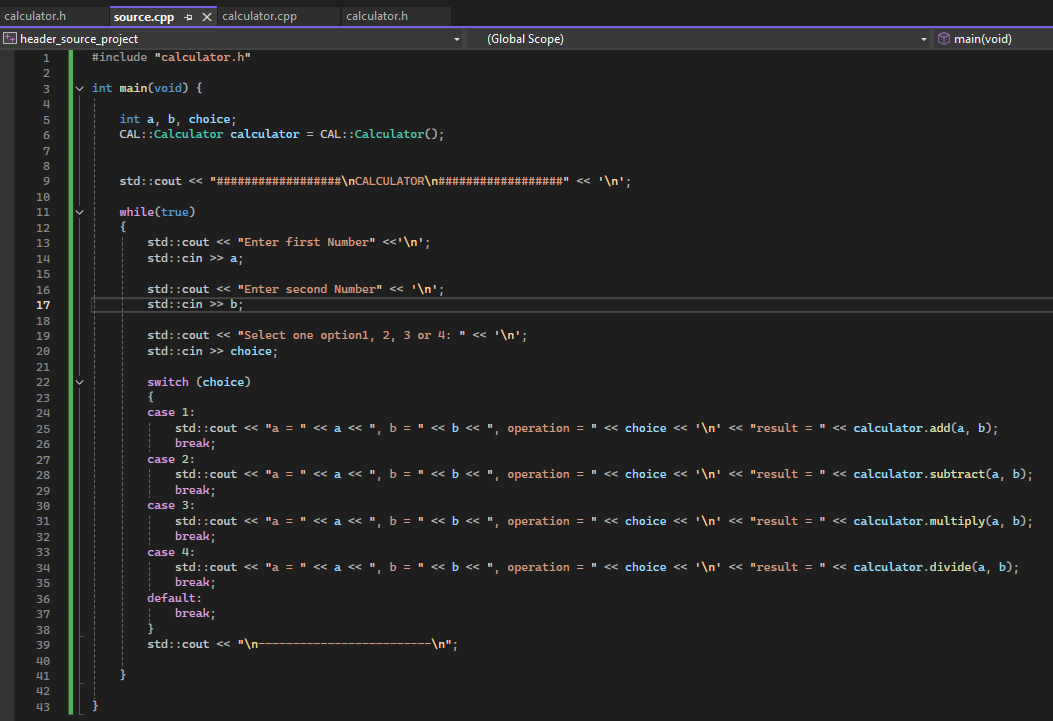
add namespace into header file and use it in source file.

### Via header and source:

source code into header and source file.







## Exception:

Exceptions are types that communicate an error condition. When an error conditions you can either handle this error by returning an error code or by throwing an exception using **throw** Keyword.

In C++ exceptions are implemented in standard library. Throwing an exception can be within try/catch or without try/catch block.

Exception inheritance structure:

exception:

runtime\_error:

system\_error

underflow\_error

overflow\_error

..

..

logic\_error:

domain\_error

invalid\_argument

length\_error

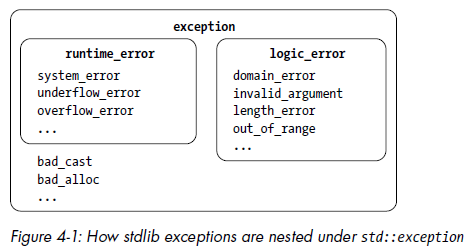
out\_of\_range

..

..

bad\_cast

bad\_alloc



### Logic Errors:

*Logic errors* derive from the logic\_error class. Generally, you could avoid these exceptions through more careful programming. Following exceptions are derived from logic\_error exception.

* domain\_error reports errors related to valid input range, especially for math functions. The square root, for example, only supports nonnegative numbers (in the real case). If a negative argument is passed, a square root function could throw a domain\_error.
* The invalid\_argument exception reports generally unexpected arguments.
* The length\_error exception reports that some action would violate a maximum size constraint.
* The out\_of\_range exception reports that some value isn’t in an expected range. The canonical example is bounds-checked indexing into a data structure.

### Runtime errors

*Runtime errors* derive from the runtime\_error class. These exceptions help you report error conditions that are outside the program’s scope.

* The system\_error reports that the operating system encountered some error. Inside of the <system\_error> header, there’s a large number of *error codes* and *error conditions*. When a system\_error is constructed, Information about the error is packed in so you can determine the nature of the error. The .code() method returns an enum class of type std::errc that has a large number of
* The overflow\_error and underflow\_error report arithmetic overflow and underflow,

### bad\_alloc

Derived directly from exception, bad\_alloc exception, which reports that new failed to allocate the required memory for dynamic storage.

## Rethrowing an exception:

In a catch block, you can use the throw keyword to resume searching for an appropriate exception handler.

try {

// Some code that might throw a system\_error

--snip—

} catch(const std::system\_error& ex) {

if(ex.code()!= std::errc::permission\_denied){

// Not a permission denied error

throw;

}

// Recover from a permission denied

--snip--

}

The keyword noexcept is another exception-related term you should know. You can, and should, mark any function that cannot possibly throw an exception noexcept,

bool is\_odd(int x) noexcept {

return 1 == (x % 2);

}

Functions marked noexcept make a rigid contract. When you’re using a function marked noexcept, you can rest assured that the function cannot throw an exception. In exchange, you must be extremely careful when you mark your own function noexcept, since the compiler won’t check for you. If your code throws an exception inside a function marked noexcept, it’s bad juju. The C++ runtime will call the function std::terminate, a function that by default will exit the program via abort. Your program cannot recover:

The **noexcept** specifier tells the compiler that exception handling is not required, so if an

exception does occur in the function the exception will not be bubbled out of the function.

and the **terminate** function will be called immediately. In this situation, there is no

guarantee that the destructors of the automatic objects are called Exception Object: exception object can be anything you like: an object, a pointer, or a built-in type. But it is desired or standard to throw exception class object.

double reciprocal(double d)

{

if (d == 0)

{

// throw 0;

// throw "divide by zero";

// throw new exception("divide by zero");

throw exception("divide by zero");

}

return 1.0 / d;

}

When an exception is thrown, the exception handling infrastructure takes over. Execution

will stop in the current code block and the exception will be propagated up the call stack.

As the exception propagates through a code block, all the automatic objects will be

destroyed, but objects created on the heap in the code black will not be destroyed. This is a

process called **stack unwinding,** whereby each stack frame is cleaned up as much as

possible before the exception moves to the stack frame above it in the call stack. If the

exception is not caught, it will propagate up to the main function, at which point the

terminate function will be called to handle the exception (and hence it will terminate the

process).

try

{

string s("this is an object");

vector<int> v = { 1, 0, -1};

reciprocal(v[0]);

reciprocal(v[1]);

reciprocal(v[2]);

}

catch(exception& e)

{

cout << e.what() << endl;

}

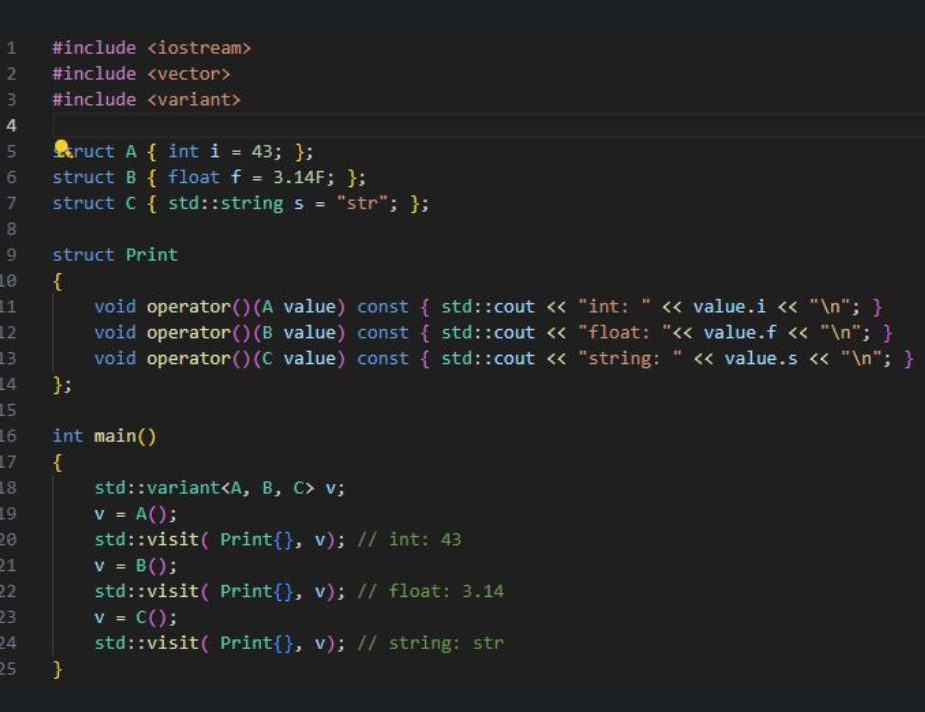
exceptions are caught by reference, so that the actual exception object, and not a copy, is caught.

### Example of Stack Unwinding:

The Standard Library provides a function called uncaught\_exception(), which returns

true if an exception has been thrown but not yet handled.

Questions:



What is strongly typed and weakly typed language:

For sure it’s synthetic example. It supposed to show the flexibility std::variant provides and how it may be used for visitor pattern implementation. Its not code reorganization, it’s just a more clever approach to deal with a problem

there are better ways to do stuff. Inheritance limits u of adding new functionality to classes and introduces overhead related run time polymorphism. Visitor pattern is not a silver bullet, but it can drastically improve performance and maintainability if properly used.

This looks great; printing is of course a trivial load for each instance visited, but the beauty lies in how a more complex action per-visit can be more clearly shown.  
  
I'm more excited about the benefits to long-term maintenance: the compiler is able to detect, say, a "struct D" that's listed as an operator but not on line 18, or vice versa (confirmed this in the "godbolt" playground: define struct D, define in line 18, but no operator in Print{} )  
  
If this sort of compile-time detection of user-error is possible through, for example, a templated function that leverages the std::visit, but Print{} lacks the operator to service this, we should see the compiler flag the std::visit function in the templated code, clearly identifying the mossing chunk.  
  
Other languages would be switch/case the type,and you can miss a type, and not discover at build time.

The above code solves a problem called "pattern matching".