Modern C++ Programming

5. Basic Concepts IV

- Functions and Preprocessing

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.06

Table of Context

Declaration and Definition

Declaration/Definition

Declaration/Prototype

A **declaration** (or prototype) of an entity is an identifier describing its type

A declaration is what the compiler and the linker needs to accept references to that identifier

Definition/Implementation

An entity **definition** is the implementation of a declaration

Declaration/Definition Example

A declaration without a concrete implementation is an incomplete type (as `void`)

C++ entities (class, functions, etc.) can be declared multiple times (with the same signature)

```
struct A;    // declaration 1
struct A;    // declaration 2 (ok)

struct B {   // declaration and definition
    int b;
//  A   x;   // compile error incomplete type
    A*  y;   // ok, pointer to incomplete type
};

struct A {   // definition
    char c;
}
```

Functions

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task*

Purpose:

- **Avoiding Code Duplication** less code for the same functionality $\rightarrow$ less bugs
- **Readability** better express what the code does
- **Organization** break the code in separate modules

Signature

Type signature defines the *inputs* and *outputs** for a function. A type signature includes the number of arguments, the types of arguments and the order of the arguments contained by a function

Function Parameter [formal]

A **parameter** is the variable which is part of the method's signature

Function Argument [actual]

An **argument** is the actual value (instance) of the variable that gets passed to the function

* (return type) if the function is generated from a function template
`https://stackoverflow.com/a/292390`

```
void f(int a, char* b);  // function declaration
                         // signature:  (int, char*)
                         // parameters: int a, char* b

void f(int a, char*) {}  // function definition
                         // b can be omitted if not used

// char f(int a, char* b); // compile error same signature
                           // but different return types

// void f(const int a, char* b); // invalid conflict
                                 // const int == int

void f(int a, const char* b);    // ok

f(3, "abc");  // function arguments: 3, "abc"
              // "f" calls f(int, const char*)
```

Pass by-Value

Call-by-value

The object is copied and assigned to input arguments of the method `f(T x)`

Advantages:

- Changes made to the parameter inside the function have no effect on the argument

Disadvantages:

- Performance penalty if the copied arguments are large (e.g. a structure with a large array)

When to use:

- Built-in data type and small objects ($\leq$ 8 bytes)

When not to use:

- Fixed size arrays which decay into pointers
- Large objects

Pass by-Pointer

Call-by-pointer

The address of a variable is copied and assigned to input arguments of the method `f(T* x)`

Advantages:

- Allows a function to change the value of the argument
- Copy of the argument is not made (fast)

Disadvantages:

- The argument may be null pointer
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- When passing *raw* arrays (use `const T*` if read-only)

When not to use:

- All other cases

Pass by-Reference

Call-by-reference

The reference of a variable is copied and assigned to input arguments of the method `f(T& x)`

Advantages:

- Allows a function to change the value of the argument (better readability compared with pointers)
- Copy of the argument is not made (fast)
- References must be initialized (no null pointer)
- Avoid implicit conversion (without `const T&`)

When to use:

- All cases except raw pointers

When not to use:

- Pass by-value *could* give performance advantages and improve the readability with built-in data type and small objects

Examples

```cpp
struct MyStruct;

void f1(int a);          // pass by-value
void f2(int& a);         // pass by-reference
void f3(const int& a);   // pass by-const reference
void f4(MyStruct& a);    // pass by-reference

void f5(int* a);         // pass by-pointer
void f6(const int* a);   // pass by-const pointer
void f7(MyStruct* a);    // pass by-pointer

void f8(int*& a);        // pass a pointer by-reference
//--------------------------------------------------------------
char c = 'a';
f1(c);     // ok, pass by-value (implicit conversion)
// f2(c); // compile error different types
f3(c);     // ok, pass by-value (implicit conversion)
```

Function Overloading

Overloading

An **overloaded declaration** is a declaration with the same name as a previously declared identifier which have different number of arguments and types

Overload resolution rules:

- An exact match
- A promotion (e.g. `char` to `int`)
- A standard type conversion (e.g. `float` and `int`)
- A constructor or user-defined type conversion

Function Overloading + Ambiguous Matches

```
void f(int   a);
void f(float b);         // overload
void f(float b, char c); // overload

void g(int a);

//--------------------------------------------------------------

   f(0);         // ok
// f('a');       // compile error ambiguous match
   f(2.3f);      // ok
// f(2.3);       // compile error ambiguous match
   f(2.3, 'a'); // ok

   g(2.3);       // ok, standard type conversion
```

Function Default Parameters

Default/Optional parameter

A **default parameter** is a function parameter that has a default value provided to it

- If the user does not supply a value for this parameter, the default value will be used
- All default parameters must be the rightmost parameters
- Default parameters can only be declared once
- Default parameters can improve compile time and avoid redundant code because they avoid defining other overloaded functions

```
void f(int a, int b = 20);    // declaration
// void g(int a = 10, int b); // compile error
                              // it is not the rightmost
//void f(int a, int b = 10) { ... } // default value of "b"
                                    // already set in the declaration
void f(int a, int b) { ... } // default value of "b" is already set

f(5);  // b is 20
```

inline

`inline` specifier allows a function to be defined identically (not only declared) in multiple translation units (source file + headers) [see "Translation Units" slides]

- `inline` is one of the most misunderstood features of C++
- `inline` is a hint for the linker. Without it, the linker can emit "`multiple definitions`" error
- It can be applied for optimization purposes only if the function has *internal* linkage (`static` or inside an `anonymous namespace`)
- C++17 `inline` can be also applied to variables

```
inline        void f()  { ... } // external linkage
static        void g1() { ... } // internal linkage
static inline void g2() { ... } // internal linkage
namespace {
    inline void g3() { ... }    // internal linkage
}// anonymous namespace -> same as static
```

`f()`:

- Can be defined in a header and included in multiple source files
- The linker removes all definitions except one
- Declaring `void f();` in a file that does not include the header is still valid because the function has *external* linkage

`g1(), g2(), g3()`:

- Can be defined in a header included in multiple source files
- The compiler replicates the code in each translation unit (the linker does not see these functions)
- Declaring `void g1();` in a file that does not include the header is no more valid because the function has *internal* linkage

inline (internal linkage)

`inline` specifier is a hint for the compiler. The code of the function can be copied where it is called (inlining)

```
inline void f(int a) { ... }
```

- It is just a hint for the compiler that can ignore it (`inline` increases the compiler heuristic threshold)
- `inline` functions increase the binary size because they are expanded in-place for every function call

GCC/Clang extensions allow to *force* inline/non-inline functions:

```
__attribute__((always_inline)) void f(int a) { ... }
__attribute__((noinline))      void f(int a) { ... }
```

Function Attributes

C++ allows marking functions with standard properties to better express their intent:

- C++11 `[[noreturn]]` indicates that the function does not return
- C++14 `[[deprecated]]`, `[[deprecated("reason")]]` indicates the use of a function is discouraged (for some reason). It issues a warning if used
- C++17 `[[nodiscard]]`
  C++20 `[[nodiscard("reason")]]` issues a warning if the return value is discarded
- C++17 `[[maybe_unused]]` suppresses compiler warnings on unused functions, if any (it applies also to other entities)

```cpp
[[noreturn]] void f() {
    std::exit(0);
}

[[deprecated]] void my_rand() {
    rand();
}

[[nodiscard]] int g() {
    return 3;
}

[[maybe_unused]] void h() {}

//-------------------------------
my_rand();    // WARNING "deprecated"
g();          // WARNING "discard return value"
int x = g(); // no warning
```

Function Objects and Lambda Expressions

Standard C achieves generic programming capabilities and composability through the concept of **function pointer**

A function can be passed as a pointer to another function and behaves as an *"indirect call"*

```
#include <stdlib.h>

int descending(const void* a, const void* b) {
    return *((const int*) a) > *((const int*) b);
}

int array[] = { 7, 2, 5, 1 };
qsort(array, 4, sizeof(int), descending);
// array: { 7, 5, 2, 1 }
```

```cpp
int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}
// type: int (*)(int, int)
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

cout << eval(4, 3, add); // print 7
cout << eval(4, 3, sub); // print 1
```

Problems:

safety There is no check of the argument type in the generic case (e.g. `qsort`)

performance Any operation requires an indirect call to the original function. Function inlining is not possible

Function Object

A **function object**, or **functor**, is a *callable* object that can be treated as a parameter

C++ provides a more efficient and convenience way to pass *"procedure"* to other functions called **function object**

```
#include <algorithm> // for std::sort

struct Descending { // <-- function object
    bool operator()(int a, int b) {
        return a > b;
    }
};

int array[] = { 7, 2, 5, 1 };
std::sort(array, array + 4, Descending{});
// array: { 7, 5, 2, 1 }
```

Advantages:

safety Argument type checking is always possible. It could involves templates

performance The compiler injects `operator()` in the code of the destination function and then compile the routine. Operator inlining is the standard behavior

C++11 simplifies the concept by providing less verbose `function objects` called **lambda expressions**

Lambda Expression

Lambda Expression

A **lambda expression** is an *inline local-scope* function object

```
auto x = [capture clause] (parameters) { body }
```

- The `[capture clause]` marks the declaration of the lambda and how the local scope arguments are captured (by-value, by-reference, etc.)
- The `parameters` of the lambda are normal function parameters (optional)
- The `body` of the lambda is a normal function body

The expression to the right of the `=` is the **lambda expression**, and *the runtime object* `x` created by that expression is the **closure**

Lambda Expression

```
#include <algorithm> // for std::sort

int array[] = { 7, 2, 5, 1 };
auto lambda = [](int a, int b){ return a > b; }; // named lambda

std::sort(array, array + 4, lambda);
// array: { 7, 5, 2, 1 }

// in alternative, in one line of code:          // unnamed lambda
std::sort(array, array + 4, [](int a, int b){ return a > b; });
// array: { 7, 5, 2, 1 }
```

Capture List

Lambda expressions *capture* external variables used in the body of the lambda in two ways:

- Capture *by-copy*
- Capture *by-reference* (can modify external variable values)

Capture list can be passed as follows

- `[]` no capture
- `[=]` captures all variables *by-copy*
- `[&]` captures all variables *by-reference*
- `[var1]` captures only `var1` *by-copy*
- `[&var2]` captures only `var2` *by-reference*
- `[var1, &var2]` captures `var1` *by-copy* and `var2` *by-reference*

Capture List Examples

```
// GOAL: find the first element greater than "limit"
#include <algorithm> // for std::find_if
int limit = ...

// capture by-value
auto lambda1 = [=](int value)      { return value > limit; };
// capture by-reference
auto lambda2 = [&](int value)      { return value > limit; };
// capture "limit" by-value
auto lambda3 = [limit](int value)  { return value > limit; };
// capture "limit" by-reference
auto lambda4 = [&limit](int value) { return value > limit; };
// no capture
// auto lambda5 = [](int value)    { return value > limit; }; // error

int array[] = { 7, 2, 5, 1 };
std::find_if(array, array + 4, lambda1);
```

Capture List - Other Cases

- `[=, &var1]` captures all variables used in the body of the lambda **by-copy**, except `var1` that is captured **by-reference**
- `[&, var1]` captures all variables used in the body of the lambda **by-reference**, except `var1` that is captured **by-value**
- A lambda expression can read a variable without capturing it if the variable is a `constexpr`

```
constexpr int limit = 5;
int var1 = 3, var2 = 4;

auto lambda1 = [](int value) { return value > limit; };

auto lambda2 = [=, &var2]()  { return var1 > var2;   };
```

Lambda Expressions - Other Features

C++14 Lambda expression parameters can be automatically deduced

```
auto x = [](auto value) { return value + 4; };
```

C++14 Lambda expression parameters can be initialized

```
auto x = [](int i = 6) { return i + 4; };
```

Lambda expressions can be composed

```
auto lambda1 = [](int value) { return value + 4; };
auto lambda2 = [](int value) { return value * 2; };

auto lambda3 = [&](int value) { return lambda2(lambda1(value)); };
// returns (value + 4) * 2
```

A function can return a lambda

```
auto f() {
    return [](int value) { return value + 4; };
}

auto lambda = f();
cout << lambda(2); // print "6"
```

constexpr/consteval Lambda Expression

C++17 Lambda expression supports `constexpr`

C++20 Lambda expression supports `consteval`

```
// constexpr lambda
constexpr auto factorial = [](int value) constexpr {
    int ret = 1;
    for (int i = 2; i <= value; i++)
        ret *= i;
    return ret;
};

constexpr int v1 = factorial(4); // '24'

constexpr int f() {
   return factorial(3); // 6
}
```

`template` Lambda Expression

C++20 Lambda expression supports `template` and `requires` clause

```
auto lambda = []<typename T>(T value)
              requires std::is_is_arithmetic<T> {
    return value * 2;
};

auto v = lambda(3.4); // v: 6.8 (double)

struct A{} a;
// auto v = lambda(a); // compiler error
```

`mutable` Lambda Expression

Lambda capture is *by-const-value*

`mutable` specifier allows the lambda to modify the parameters captured *by-value*

```
int  var = 1;

auto lambda1 = [&]() { var = 4; };          // ok
lambda1();
cout << var; // print '4'

// auto lambda2 = [=]() { var = 3; };       // compile error
// lambda operator() is const

auto lambda3 = [=]() mutable { var = 3; }; // ok
lambda3();
cout << var;  // print '4', lambda3 captures by-value
```

Capture List and Classes ⋆

- `[this]` captures the current object `(*this)` *by-reference*
- `[x = x]` captures the current object member `x` *by-copy* C++14
- `[&x = x]` captures the current object member `x` *by-reference* C++14
- `[=]` default capture of `this` pointer by value has been deprecated C++20

```
class A {
    int data = 1;

    void f() {
        int var = 2; // <--
        // return 3 (nearest scope)
        auto lambda1 = [=]() { int var = 3; return var; };
        // return 2 (nearest scope)
        auto lambda2 = [=]()     { return var; };  // copy by-value
        auto lambda3 = [this]()  { return data; }; // copy by-reference
        auto lambda3 = [*this]() { return data; }; // copy by-value, only C++17
//      auto lambda4 = [data]()  { return data; }; // compile error not visible
        auto lambda5 = [data = data]() { return data; }; // return 1
    }
};
```

Preprocessing

Preprocessing and Macro

Preprocessor directives are lines preceded by a *hash* symbol (#) which tell the compiler how to interprets the source code *before* compiling

Macro are preprocessor directives which replace any occurrence of an *identifier* in the rest of the code by replacement

Macro are evil:

Do not use macro expansion!!

...or use as little as possible

- Macro cannot be debugged
- Macro expansions can have strange side effects
- Macro have no namespace or scope

Preprocessors

All statements starting with `#`

- `#include "my_file.h"`
  Inject the code in the current file

- `#define MACRO <expression>`
  Define a new macro

- `#undef MACRO`
  Undefine a macro
  (a macro should be undefined as early as possible for safety reasons)

Multi-line Preprocessing: `\` at the end of the line

Indent: `#   define`

Conditional Compiling

- ```
  #if <condition>
      code
  #elif <condition>
      code
  #else
      code
  #endif
  ```

- `#if defined(MACRO)` equal to `#ifdef MACRO`
  Check if a macro is defined

- `#if !defined(MACRO)` equal to `#ifndef MACRO`
  Check if a macro is not defined
  ```

Macro (Common Error 1)

Do not define macro in header files and before includes!!

```cpp
#include <iostream>

#define value    // very dangerous!!
#include "big_lib.hpp"

int main() {
    std::cout << f(4); // should print 7, but it prints always 3
}
```

big_lib.hpp:

```cpp
int f(int value) {    // 'value' disapear
    return value + 3;
}
```

It is very hard to see this problem when the macro is in a header

Macro (Common Error 2)

Use parenthesis in macro definition!!

```cpp
#include <iostream>

#define SUB1(a, b)  a - b        // WRONG
#define SUB2(a, b) (a - b)       // WRONG
#define SUB3(a, b) ((a) - (b))  // correct

int main() {
    std::cout << (5 * SUB1(2, 1));   // print 9 not 5!!
    std::cout << SUB2(3 + 3, 2 + 2); // print 6 not 2!!
    std::cout << SUB3(3 + 3, 2 + 2); // print 2
}
```

Macros make hard to find compile errors!!

```cpp
 1: #include <iostream>
 2:
 3: #define F(a) {        \
 4:         ...           \
 5:         ...           \
 6:         return v;
 7:
 8: int main() {
 9:     F(3);      // compile error at line 9!!
10: }
```

- **In which line is the error??!***

*modern compilers are able to roll out the macro

Macro (Common Error 4)

Use curly brackets in multi-lines macros!!

```cpp
#include <iostream>
#include <nuclear_explosion.hpp>

#define NUCLEAR_EXPLOSION                  \  // {
    std::cout << "start nuclear explosion"; \
    nuclear_explosion();
                                              // }
int main() {
    bool never_happen = false;
    if (never_happen)
        NUCLEAR_EXPLOSION
} // BOOM!! ☠
```

The second line is executed!!

Macro (Common Error 5)

Macros do not have scope!!

```cpp
#include <iostream>

void f() {
    #define value 4
    std::cout << value;
}

int main() {
    f();                // 4
    std::cout << value; // 4
    #define value 3
    f();                // 4
    std::cout << value; // 3
}
```

Macro (Common Error 6)

Macros can have side effect!!

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main() {
    int array1[] = { 1, 5, 2 };
    int array2[] = { 6, 2, 4 };
    int i = 0;
    int j = 0;
    int v1 = MIN(array1[i++], array2[j++]); // v1 = 5!!
    int v2 = MIN(array1[i++], array2[j++]); // segfault ☠
}
```

Reference: `arne-mertz.de/2019/03/macro-evil`

When Preprocessors are Necessary

- **Conditional compiling**: different architectures, compiler features, etc.
- **Mixing different languages**: code generation (example: `asm` assembly)
- **Complex name replacing**: see template programming

Otherwise, prefer `const` and `constexpr` for constant values and functions

```
#define SIZE 3      // replaced with
const int SIZE = 3; // only C++11 at global scope

#define SUB(a, b) ((a) - (b)) // replaced with
constexpr int sub(int a, int b) {
    return a - b;
}
```

Commonly used macros:

- `__LINE__` Integer value representing the current line in the source code file being compiled
- `__FILE__` A string literal containing the presumed name of the source file being compiled
- `__DATE__` A string literal in the form "MMM DD YYYY" containing the date in which the compilation process began
- `__TIME__` A string literal in the form "hh:mm:ss" containing the time at which the compilation process began

main.cpp:

```cpp
#include <iostream>
int main() {
    std::cout << __FILE__ << ":" << __LINE__;  // print main.cpp:2
}
```

C++20 provides source location utilities for replacing macro-based approach

`#include <source_location >`

`current()`	get source location info (static)
`line()`	source code line
`column()`	line column
`file_name()`	current file name
`function_name()`	current function name

```
#include <source_location>

void f(std::source_location s = std::source_location::current()) {
    std::cout << "line " << s.line();
}
f(); // print: "line 6"
```

Select code depending on the C/C++ version

- `#if defined(__cplusplus)` C++ code
- `#if __cplusplus == 199711L` ISO C++ 1998/2003
- `#if __cplusplus == 201103L` ISO C++ 2011*
- `#if __cplusplus == 201402L` ISO C++ 2014*
- `#if __cplusplus == 201703L` ISO C++ 2017*

Select code depending on the compiler

- `#if defined(__GNUG__)` The compiler is gcc/g++ †
- `#if defined(__clang__)` The compiler is clang/clang++
- `#if defined(_MSC_VER)` The compiler is `Microsoft Visual C++`

---

* MSVC defines `__cplusplus == 199711L` even for C++11/14. Link: `MSVC now correctly reports __cplusplus Avatar`

† `__GNUC__` is defined by many compilers. Link: `GCC __GNUC__Meaning`

Select code depending on the operation system or environment

- `#if defined(_WIN64)` OS is `Windows 64-bit`
- `#if defined(__linux__)` OS is `Linux`
- `#if defined(__APPLE__)` OS is `Mac OS`
- `#if defined(__MINGW32__)` OS is `MinGW 32-bit`
- ...and many others

Very Comprehensive Macro list:

- `sourceforge.net/p/predef/wiki/Home/`
- `Compiler predefined macros`
- `Abseil platform macros`

Feature Testing Macro

C++17 introduces `__has_include` keyword which return 1 if header or source file with the specified name exists

```
#if __has_include(<iostream>)
#   include <iostream>
#endif
```

C++20 introduces a set of macro to evaluate if a given feature is supported by the compiler

```
#if __cpp_constexpr
constexpr int square(int x) { return x * x; }
#endif
```

Full list: `Feature Testing Macros`

Macro (Common Error 7)

Macros depend on compilers and environment!!

```cpp
struct A {
    int x;  // enable C++11 code
#if __cplusplus >= 201103
    A() = default;
#else
    A() {}
#endif
};
```

```cpp
// should return ≈ 10.0f
float safe_function() {
    A a{}; // zero-initialization
    for (int i = 0; i < 10; i++)
        a.x += 1.0f;
    return a.x;
}
// what is the behavior ???
```

The code works fine on Linux, but not under Windows MSVC. MSVC sets `__cplusplus` to `199711` even if C++11/14/17 flag is set!! in this case the code can return `NaN`

---

see Lecture "Object-Oriented Programming II - Zero Initialization" and `MSVC now correctly reports __cplusplus`

Stringizing Operator (#)

The **stringizing macro operator** (`#`) causes the corresponding actual argument to be enclosed in double quotation marks `"`

```
#define STRING_MACRO(string) #string

cout << STRING_MACRO(hello);  // equivalent to "hello"
```

```
#define INFO_MACRO(my_func)                           \
{                                                      \
    my_func                                            \
    cout << "call " << #my_func << " at "              \
         << __FILE__ << ":" __LINE__;                  \
}
void g(int) {}

INFO_MACRO( g(3) ) // print: "call g(3) at my_file.cpp:7"
```

Code injection

```
#include <cstdio>

#define CHECK_ERROR(condition) \
{
   if (condition) {
      std::printf("expr: " #condition " failed at line %d\n",
                  __LINE__);
   }
}

int main() {
    int t = 6, s = 3;
    CHECK_ERROR(t > s) // print "expr: t > s failed at line 13"
    CHECK_ERROR(t % s == 0) // segmentation fault!!! ☠
}       // printf interprets "% s" as a format specifier
```

#error and #pragma

- `#error "text"` The directive emits a user-specified error message at compile time when the compiler parse the related instruction

The `#pragma` directive controls implementation-specific behavior of the compiler. In general, it is not portable

- `#pragma message "text"` Display informational messages at compile time (every time this instruction is parsed)
- `#pragma GCC diagnostic warning "-Wformat"`
  Disable a GCC warning
- `_Pragma(<command>)` (C++11)
  It is a keyword and can be embedded in a `#define`

```
#define MY_MESSAGE \
    _Pragma("message(\"hello\")")
```

Token-Pasting Operator (##) ⋆

The **token-concatenation (or pasting) macro operator** (`##`) allows combining two tokens (without leaving no blank spaces)

```
#define FUNC_GEN_A(tokenA, tokenB)  \
    void tokenA##tokenB() {}

#define FUNC_GEN_B(tokenA, tokenB)  \
    void tokenA##_##tokenB() {}

FUNC_GEN_A(my, function)
FUNC_GEN_B(my, function)

int main() {
    myfunction();  // ok, from FUNC_GEN_A
    my_function(); // ok, from FUNC_GEN_B
}
```

Variadic Macro ★

In C++11, a **variadic macro** is a special macro accepting a varying number of arguments (separated by comma)

Each occurrence of the special identifier `__VA_ARGS__` in the macro replacement list is replaced by the passed arguments

Example:

```
void f(int a)                 { printf("%d", a);             }
void f(int a, int b)          { printf("%d %d", a, b);       }
void f(int a, int b, int c)   { printf("%d %d %d", a, b, c); }

#define PRINT(...) \
    f(__VA_ARGS__);

int main() {
    PRINT(1, 2)
    PRINT(1, 2, 3)
}
```

Macro Trick ⋆

Convert a number literal to a string literal

```
#define TO_LITERAL_AUX(x) #x
#define TO_LITERAL(x)     TO_LITERAL_AUX(x)
```

Motivation: avoid integer to string conversion (performance)

```
int main() {
   int  x1   = 3 * 10;
   int  y1   = __LINE__ + 4;
   char x2[] = TO_LITERAL(3);
   char y2[] = TO_LITERAL(__LINE__);
}
```