Modern C++ Programming

21. Advanced Topics II

Federico Busato

2025-02-24

Table of Contents

Table of Contents

Table of Contents

Undefined Behavior

Undefined Behavior Overview

Undefined behavior means that the semantic of certain operations is

- *Unspecified behavior*: outside the language/library specification, two or more choices
- *Illegal*: the compiler presumes that such operations never happen, e.g. integer overflow
- *Implementation-defined behavior*: depends on the compiler and/or platform (not portable)

Motivations behind undefined behavior:

- *Compiler optimizations*, e.g. signed overflow or NULL pointer dereferencing
- *Simplify compile checks*
- *Unfeasible/expensive* to check

- `What Every C Programmer Should Know About Undefined Behavior`, *Chris Lattner*
- `What are all the common undefined behaviors that a C++ programmer should know about?`
- `Enumerating Core Undefined Behavior`

- `const_cast` applied to a `const` variables

```
const int          var  = 3;
const_cast<int&>(var) = 4;
... // use var
```

- **Memory alignment**

```
char* ptr  = new char[512];
auto  ptr2 = reinterpret_cast<uint64_t*>(ptr + 1);
ptr2[3]; //  ptr2 is not aligned to 8 bytes (sizeof(uint64_t))
```

- **Memory initialization**

```
int  var;              // undefined value
auto var2 = new int; // undefined value
```

- **Memory access-related**: *Out-of-bound access:* the code could crash or not depending on the platform/compiler

- **Strict aliasing**

```
float x = 3;
auto  y = reinterpret_cast<unsigned&>(x);
// x, y break the strict aliasing rule
```

- **Lifetime issues**

```
int* f() {
    int tmp[10];
    return tmp;
}
int* ptr = f();
ptr[0];
```

- **One Definition Rule violation**
  - Different definitions of `inline` functions in distinct translation units

- **Missing `return` statement**

```
int f(float x) {
    int y = x * 2;
}
```

- **Dangling reference**

```
int        n = 1;
const int& r = std::max(n-1, n+1); // dangling
// GCC 13 experimental -Wdangling-reference (enabled by -Wall)
```

- **Illegal arithmetic and conversion operations**
  - Division by zero `0 / 0`, `fp_value / 0.0`
  - Floating-point to integer conversion

Platform Specific Behavior

- **Memory access-related**: `NULL` *pointer dereferencing*: the `0x0` address is valid in some platforms

- **Endianness**

```
union U {
    unsigned x;
    char     y;
};
```

- **Type definition**

```
long x = 1ul << 32u; // different behavior depending on the OS
```

- **Intrinsic functions**

Unspecified Behavior

Legal operations but the C++ standard does not document the result → different compilers/platforms can show different behavior

- Signed shift of negative values `-2 << x` (before C++20), large-than-type shift `3u << 32`, etc.
- Floating-point narrowing conversion to floating-point or integer types with unrepresentable values `double → float`, `float → int`
- Arithmetic operation ordering `f(i++, i++)`
- Function evaluation ordering

```
auto x = f() + g(); // C++ doesn't ensure that f() is evaluated before g()
```

- Signed overflow

```
for (int i = 0; i <= N; i++)
```

if `N` is `INT_MAX`, the last iteration is undefined behavior. The compiler can assume that the loop is finite and enable important optimizations, as opposite to `unsigned` (wrap around)

- Trivial infinite loops, until C++26

```
int main() {
  while (true)   // -> std::this_thread::yield(); in C++26
    ;
}
void unreachable() { cout << "Hello world!" << endl; }
```

the code print `Hello world!` with some clang versions

Detecting Undefined Behavior

There are several ways to detect or prevent undefined behavior at compile-time and at run-time:

- Modify the compiler behavior, see `Debugging and Testing: Hardening Techniques`
- Using undefined behavior sanitizer, see `Debugging and Testing: Sanitizer`
- Static analysis tools
- `constexpr` expressions doesn't allow undefined behavior

```
constexpr int   x1 = 2147483647 + 1;   // compile error
constexpr int   x2 = (1 << 32);        // compile error
constexpr int   x3 = (1 << -1);        // compile error
constexpr int   x4 =  3 / 0;           // compile error
constexpr int   x5 = *((int*) nullptr) // compile error
constexpr int   x6 = 6
constexpr float x7 = reinterpret_cast<float&>(x6); // compile error
```

Error Handing

Recoverable Error Handing

Recoverable *Conditions that are not under the control of the program*. They indicate *"exceptional"* run-time conditions. e.g. file not found, bad allocation, wrong user input, etc.

A *recoverable* should be considered *unrecoverable* if it is extremely rare and difficult to handle, e.g. bad allocation due to out-of-memory error

The common ways for handling recoverable errors are:

Exceptions Robust but slower and requires more resources

Return code Fast but difficult to handle in complex programs

Error Handing References

- Modern C++ best practices for exceptions and error handling
- Back to Basics: Exceptions - CppCon2020
- ISO C++ FAQ: Exceptions and Error Handling
- Zero-overhead deterministic exceptions: Throwing values, P0709
- C++ exceptions are becoming more and more problematic, P2544
- std::expected
- C++ Error Handling Strategies - Benchmarks and Performance

Return Code

Historically, C programs handled errors with return codes, even for unrecoverable errors

```
enum Status { IllegalValue, Success };

Status f(int* ptr) { return (ptr == nullptr) ? IllegalValue : Success; }
```

Why such behavior? Debugging → need to understand what / where / why the program failed

A better approach in C++ involves `std::source_location()` C++20 and `std::stacktrace()` C++23

ABI related issues:

- Removing an enumerator value is an API breaking change
- Adding a new enumerator value associated to a return type is also problematic as it causes ABI breaking change

C++ Exceptions - Advantages

C++ Exceptions provide a well-defined mechanism to detect errors passing the information up the call stack

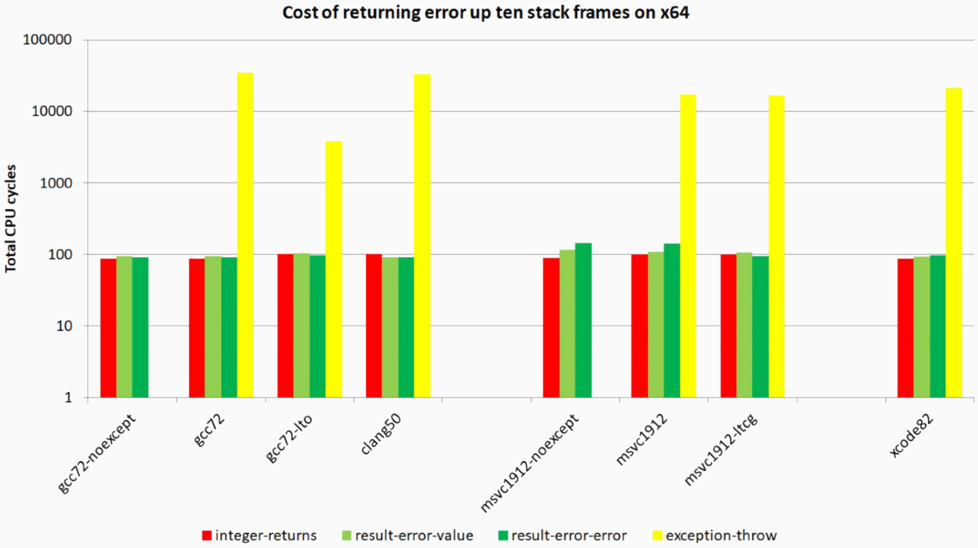
- **Exceptions cannot be ignored**. Unhandled exceptions stop program execution (call `std::terminate()`)
- **Intermediate functions are not forced to handle them**. They don't have to coordinate with other layers and, for this reason, they provide good composability
- Throwing an exception **acts like a return statement** destroying all objects in the current scope
- An exception enables a **clean separation** between the code that detects the error and the code that handles the error
- Exceptions work well with object-oriented semantic (constructor)

C++ Exceptions - Disadvantages 1/2

- **Code readability**: Using exception can involve more code than the functionality itself
- **Code comprehension**: Exception control flow is invisible and it is not explicit in the function signature
- **Performance:** Extreme performance overhead in the failure case (violate the zero-overhead principle)
- **Dynamic behavior**: `throw` requires dynamic allocation and `catch` requires `RTTI`. It is not suited for real-time, safety-critical, or embedded systems
- **Code bloat**: Exceptions could increase executable size by 5-15% (or more*)

*Binary size and exceptions

C++ Exceptions - Disadvantages 2/2



C++ Exception Basics

C++ provides three keywords for exception handling:

- `throw` Throws an exception
- `try` Code block containing potential throwing expressions
- `catch` Code block for handling the exception

```
void f() { throw 3; }

int main() {
    try {
        f();
    } catch (int x) {
        cout << x; // print "3"
    }
}
```

`std` Exceptions

`throw` can throw everything such as integers, pointers, objects, etc. The standard way consists in using the `std` library exceptions `<stdexcept>`

```
#include <stdexcept>

void f(bool b) {
    if (b)
        throw std::runtime_error("runtime error");
    throw std::logic_error("logic error");
}
int main() {
    try {
        f(false);
    } catch (const std::runtime_error& e) {
        cout << e.what();
    } catch (const std::exception& e) {
        cout << e.what();  // print: "logic error"
    }
}
```

Exception Capture

NOTE: C++, differently from other programming languages, does not require explicit dynamic allocation with the keyword `new` for throwing an exception. The compiler implicitly generates the appropriate code to construct and clean up the exception object. Dynamically allocated objects require a `delete` call

The right way to capture an exception is by `const`-reference. Capturing by-value is also possible but, it involves useless copy for non-trivial exception objects

`catch(...)` can be used to capture any thrown exception

```
int main() {
    try {
        throw "runtime error"; // throw const char*
    } catch (...) {
        cout << "exception";    // print "exception"
    }
}
```

Exception Propagation

Exceptions are automatically propagated along the call stack. The user can also control how they are propagated

```cpp
int main() {
    try {
        ...
    } catch (const std::runtime_error& e) {
        throw e; // propagate a copy of the exception
    } catch (const std::exception& e) {
        throw;   // propagate the exception
    }
}
```

Defining Custom Exceptions

```cpp
#include <exception> // to not confuse with <stdexcept>

struct MyException : public std::exception {
    const char* what() const noexcept override { // could be also "constexpr"
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch (const std::exception& e) {
        cout << e.what(); // print "C++ Exception"
    }
}
```

`noexcept` Keyword

C++03 allows listing the exceptions that a function might directly or indirectly throw, e.g. `void f() throw(int, const char*) {`

C++11 deprecates `throw` and introduces the `noexcept` keyword

```
void f1();                 // may throw
void f2() noexcept;        // does not throw
void f3() noexcept(true);  // does not throw
void f4() noexcept(false); // may throw
template<bool X>
void f5() noexcept(X);     // may throw if X is false
```

If a `noexcept` function throw an exception, the runtime calls `std::terminate()`

`noexcept` should be used when throwing an exception is impossible or unacceptable. It is also useful when the function contains code outside user control, e.g. `std` functions/objects

Function-try-block

Exception handlers can be defined around the body of a function.

The behavior is the same as using the `try/catch` blocks within the function scope

→ less verbose

```
void f() try {
    ... // do something
} catch (const std::runtime_error& e) {
    cout << e.what();
} catch (...) { // other exception
    ...
}
```

Memory Allocation Issues

The `new` operator automatically throws an exception (`std::bad_alloc`) if it cannot allocate the memory

`delete` never throws an exception (unrecoverable error)

```
int main() {
    int* ptr = nullptr;
    try {
        ptr = new int[1000];
    }
    catch (const std::bad_alloc& e) {
        cout << "bad allocation: " << e.what();
    }
    delete[] ptr;
}
```

C++ also provides an overload of the `new` operator with non-throwing memory allocation

```
#include <new> // std::nothrow

int main() {
    int* ptr = new (std::nothrow) int[1000];
    if (ptr == nullptr)
        cout << "bad allocation";
}
```

Throwing exceptions in *constructors* is fine while it is not allowed in *destructors*

```
struct A {
    A()  { new int[10]; }
    ~A() { throw -2;    }
};
int main() {
    try {
        A a; // could throw "bad_alloc"
             // "a" is out-of-scope -> throw 2
    } catch (...) {
             // two exceptions at the same time
    }
}
```

Destructors should be marked `noexcept`

```
struct A {
    int* ptr1, *ptr2;

    A() {
        ptr1 = new int[10];
        ptr2 = new int[10]; // if bad_alloc here, ptr1 is lost
    }
};
```

```
struct A {
    std::unique_ptr<int[]> ptr1, ptr2;

    A() {
        ptr1 = std::make_unique<int[]>(10);
        ptr2 = std::make_unique<int[]>(10); // if bad_alloc here,
    }                                       // ptr1 is deallocated
};
```

Return Code and Exception Summary

	Exception	Return Code
Pros	▪ Cannot be ignored ▪ Work well with object-oriented semantic ▪ Information: Exceptions can be arbitrarily rich ▪ Clean code: Conceptually, clean separation between the code that detects errors and the code that handles the error, but. . . * ▪ Non-Intrusive wrt. API: Proper communication channel	▪ Visibility: prototype of the called function ▪ No performance overhead ▪ No code bloat ▪ Easy to debug
Cons	▪ Visibility: Not visible without further analysis of the code or documentation ▪ Clean code: *... handling exception can generate more code than the functionality itself ▪ Dynamic behavior: memory and RTTI ▪ Extreme performance overhead in the failure case ▪ Code bloat ▪ Non-trivial to debug	▪ Easy to ignore, `[[nodiscard]]` can help ▪ Cannot be used with object-oriented semantic ▪ Information: Historically, a simple integer. Nowadays, richer error code ▪ Clean code: At least, an if statement after each function call ▪ Non-Intrusive wrt. API: Monopolization of the return channel

C++23 introduces `std::expected` to get the best properties of return codes and exceptions

The class template `expected<T, E>` contains either:

- A value of type `T`, the expected value type; or
- A value of type `E`, an error type used when an unexpected outcome occured

```
enum class Error { Invalid };

std::expected<int, Error> f(int v) {
    if (v > 0)
        return 3;
    return std::unexpected(Error::Invalid);
}
```

The user chooses how to handle the error depending on the context

```
auto ret = f(n);

// Return code handling
if (!ret)
    // error handling
int v = *ret + 3; // execute without checking

// Exception handling
ret.value(); // throw an exception if there is a problem

// Monadic operations
auto lambda = [](int x) { return (x > 3) ? 4 : std::unexpected(Error::Invalid); };
ret.and_then(lambda)                            // pass the value to another function
   .tranform([](int x) { return x + 4; };)    // transform the previous value
   .transform_error([](auto error_code){ /*error handling*/ };
```

Alternative Error Handling Approaches 1/2

- **Global state**, e.g. `errno`
  - Easily forget to check for failures
  - Error propagation using `if` statements and early `return` is manual
  - No compiler optimizations due to global state

- **Simple error code**, e.g. `int`, `enum`, etc.
  - Easily forget to check for failures (workaround `[[nodiscard]]`)
  - Error propagation using `if` statements and early `return` is manual
  - Potential error propagation through different contexts and losing initial error information
  - Constructor errors cannot be handled

Alternative Error Handling Approaches 2/2

- `std::error_code`, standardized error code
  - Easily forget to check for failures (workaround `[[nodiscard]]`)
  - Error propagation using `if` statements and early `return` is manual
  - Code bloating for adding new enumerators (see `Your own error code`)
  - Constructor errors cannot be handled
- **Supporting libraries**, e.g. `Boost Outcome`, `STX`, etc.
  - Require external dependencies
  - Constructor errors cannot be handled in a direct way
  - Extra logic for managing return values

Smart pointers

Smart Pointers

Smart pointer is a pointer-like type with some additional functionality, e.g. *automatic memory deallocation* (when the pointer is no longer in use, the memory it points to is deallocated), reference counting, etc.

C++11 provides three smart pointer types:

- **`std::unique_ptr`**
- **`std::shared_ptr`**
- **`std::weak_ptr`**

Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic

Smart Pointers Benefits

- If a smart pointer goes *out-of-scope*, the appropriate method to release resources is called automatically. The memory is not left dangling
- Smart pointers will automatically be set to `nullptr` if not initialized or when memory has been released
- `std::shared_ptr` provides automatic reference count
- If a special `delete` function needs to be called, it will be specified in the pointer type and declaration, and will automatically be called on delete

`std::unique_ptr` is used to manage any dynamically allocated object that is not shared by multiple objects

```cpp
#include <iostream>
#include <memory>
struct A {
    A()  { std::cout << "Constructor\n"; } // called when A()
    ~A() { std::cout << "Destructor\n";  } // called when u_ptr1,
};                                         // u_ptr2 are out-of-scope
int main() {
    auto                raw_ptr = new A();
    std::unique_ptr<A> u_ptr1(new A());
    std::unique_ptr<A> u_ptr2(raw_ptr);
//  std::unique_ptr<A> u_ptr3(raw_ptr); // no compile error, but wrong!! (not unique)

//  u_ptr1 = raw_ptr;                   // compile error (not unique)
//  u_ptr1 = u_ptr2;                    // compile error (not unique)
    u_ptr1 = std::move(u_ptr2); // delete u_ptr1;
}                               // u_ptr1 = u_ptr2;
                                // u_ptr2 = nullptr
```

`std::unique_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `operator[]` provides indexed access to the stored array (if it supports random access iterator)
- `release()` returns a pointer to the managed object and releases the ownership
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_unique<T>()` creates a unique pointer to a class `T` that manages a new object

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::unique_ptr<A> u_ptr1(new A());
    u_ptr1->value;      // dereferencing
    (*u_ptr1).value;    // dereferencing

    auto u_ptr2 = std::make_unique<A>();  // create a new unique pointer

    u_ptr1.reset(new A());              // reset
    auto raw_ptr = u_ptr1.release();    // release
    delete raw_ptr;

    std::unique_ptr<A[]> u_ptr3(new A[10]);
    auto& obj = u_ptr3[3];              // access
}
```

Implement a custom deleter

```
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    auto DeleteLambda = [](A* x) {
        std::cout << "delete" << std::endl;
        delete x;
    };

    std::unique_ptr<A, decltype(DeleteLambda)>
        x(new A(), DeleteLambda);
} // print "delete"
```

`std::shared_ptr` is the pointer type to be used for memory that can be owned by multiple resources at one time

`std::shared_ptr` maintains a reference count of pointer objects. Data managed by `std::shared_ptr` is only freed when there are no remaining objects pointing to the data

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    std::shared_ptr<A> sh_ptr2(sh_ptr1);
    std::shared_ptr<A> sh_ptr3(new A());
    sh_ptr3 = nullptr; // allowed, the underlying pointer is deallocated
                       //   sh_ptr3 : zero references
    sh_ptr2 = sh_ptr1; // allowed. sh_ptr1, sh_ptr2: two references
    sh_ptr2 = std::move(sh_ptr1); // allowed // sh_ptr1: zero references
}                                            // sh_ptr2: one references
```

`std::shared_ptr` methods

- `get()` returns the underlying pointer
- `operator*` `operator->` dereferences pointer to the managed object
- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`

Utility method: `std::make_shared()` creates a shared pointer that manages a new object. It is more efficient than using the `std::shared_ptr` constructors because it performs a single memory allocation instead of two

Difference in make_shared and normal shared_ptr in C++

```cpp
#include <iostream>
#include <memory>
struct A {
    int value;
};
int main() {
    std::shared_ptr<A> sh_ptr1(new A());
    auto sh_ptr2 = std::make_shared<A>();  // std::make_shared
    std::cout << sh_ptr1.use_count();  // print 1

    sh_ptr1 = sh_ptr2;                 // copy
//  std::shared_ptr<A> sh_ptr2(sh_ptr1); // copy (constructor)
    std::cout << sh_ptr1.use_count();  // print 2
    std::cout << sh_ptr2.use_count();  // print 2

    auto raw_ptr = sh_ptr1.get();      // get
    sh_ptr1.reset(new A());            // reset
    (*sh_ptr1).value = 3;              // dereferencing
    sh_ptr1->value = 2;                // dereferencing
}
```

A `std::weak_ptr` is simply a `std::shared_ptr` that is allowed to dangle (pointer not deallocated)

```
#include <memory>

std::shared_ptr<int> sh_ptr(new int);
std::weak_ptr<int>   w_ptr = sh_ptr;

sh_ptr = nullptr;
cout << w_ptr.expired(); // print 'true'
```

It must be converted to `std::shared_ptr` in order to access the referenced object

`std::weak_ptr` methods

- `use_count()` returns the number of objects referring to the same managed object
- `reset(ptr)` replaces the managed object with `ptr`
- `expired()` checks whether the referenced object was already deleted (`true`, `false`)
- `lock()` creates a `std::shared_ptr` that manages the referenced object

```cpp
#include <memory>

auto sh_ptr1 = std::make_shared<int>();
cout << sh_ptr1.use_count();  // print 1
std::weak_ptr<int> w_ptr = sh_ptr1;
cout << w_ptr.use_count();    // print 1

auto sh_ptr2 = w_ptr.lock();
cout << w_ptr.use_count();    // print 2 (sh_ptr1 + sh_ptr2)

sh_ptr1 = nullptr;
cout << w_ptr.expired();      // print false
sh_ptr2 = nullptr;
cout << w_ptr.expired();      // print true
```

Concurrency

Overview

C++11 introduces the **Concurrency** library to simplify managing OS threads

```cpp
#include <iostream>
#include <thread>

void f() {
    std::cout << "first thread" << std::endl;
}

int main(){
    std::thread th(f);
    th.join();          // stop the main thread until "th" complete
}
```

How to compile:

```
$g++ -std=c++11 main.cpp -pthread
```

Example

```
#include <iostream>
#include <thread>
#include <vector>
void f(int id) {
    std::cout << "thread " << id << std::endl;
}
int main() {
    std::vector<std::thread> thread_vect; // thread vector
    for (int i = 0; i < 10; i++)
        thread_vect.push_back( std::thread(&f, i) );

    for (auto& th : thread_vect)
        th.join();

    thread_vect.clear();
    for (int i = 0; i < 10; i++) {  // thread + lambda expression
        thread_vect.push_back(
            std::thread( [](){ std::cout << "thread\n"; } );
    }
}
```

Thread Methods

Library methods:

- `std::this_thread::get_id()` returns the thread id
- `std::thread::sleep_for( sleep_duration )`
  Blocks the execution of the current thread for at least the specified `sleep_duration`
- `std::thread::hardware_concurrency()` returns the number of concurrent threads supported by the implementation

Thread object methods:

- `get_id()` returns the thread id
- `join()` waits for a thread to finish its execution
- `detach()` permits the thread to execute independently of the thread handle

```cpp
#include <chrono>   // the following program could
#include <iostream> // produces the output (not deterministic):
#include <thread>   //     "child thread exit" (t_child < t_main)
                    //     "main thread exit"
int main() {
    using namespace std::chrono_literals;
    std::cout << std::this_thread::get_id();
    std::cout << std::thread::hardware_concurrency(); // e.g. print 6

    auto lambda = []() {
        std::this_thread::sleep_for(1s); // t_child
        std::cout << "child thread exit\n";
    };
    std::thread child(lambda);
    child.detach(); // without detach(), child must join() the
                    // main thread (run-time error otherwise)
    std::this_thread::sleep_for(2s);    // t_main
    std::cout << "main thread exit\n";
}
```

Parameters Passing

Parameters passing *by-value* or *by-pointer* to a thread function works in the same way of a standard function. *Pass-by-reference* requires a special wrapper (`std::ref`, `std::cref`) to avoid wrong behaviors

```
#include <thread>
void f(int& a, const int& b) {
    a = 7 * b;
}
int main() {
    int a = 1, b = 2;
    std::thread th1(f, a, b);                    // wrong!!!
    th1.join();
    cout << a << endl;                           // print 2!!

    std::thread th2(f, std::ref(a), std::cref(b)); // correct
    th2.join();
    cout << a << endl;                           // print 49!!
}
```

The following code produces (in general) a value < 1000:

```cpp
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>
void f(int& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    int value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;
}
```

C++11 provides the **`mutex`** class as synchronization primitive to protect shared data from being simultaneously accessed by multiple threads

`mutex` methods:

- **`lock()`** locks the *mutex*, blocks if the *mutex* is not available
- **`try_lock()`** tries to lock the *mutex*, returns if the *mutex* is not available
- **`unlock()`** unlocks the *mutex*

More advanced mutex can be found here: `en.cppreference.com/w/cpp/thread`

C++ includes three mutex wrappers to provide safe copyable/movable objects:

- `lock_guard` (C++11) implements a strictly scope-based mutex ownership wrapper
- `unique_lock` (C++11) implements movable mutex ownership wrapper
- `shared_lock` (C++14) implements movable shared mutex ownership wrapper

Mutex - Example 1


```
#include <mutex>
#include <thread> // + iostream, vector, chrono
void f(int& value, std::mutex& m) {
    for (int i = 0; i < 10; i++) {
        m.lock();
        value++;    // other threads must wait
        m.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::mutex                  m;
    int                         value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value), std::ref(m)) );
    for (auto& it : th_vect)
        it.join();
    cout << value;
}
```

Mutex - Example 2


```
#include <mutex>
#include <thread> // + iostream, vector, chrono
void f(int& value, std::mutex& m) {
    for (int i = 0; i < 10; i++) {
        {
            const std::lock_guard<std::mutex> lock(m);
            value++;    // other threads must wait
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::mutex                    m;
    int                           value = 0;
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value), std::ref(m)) );
    for (auto& it : th_vect)
        it.join();
    cout << value;
```

Atomic

`std::atomic` (C++11) class template defines an atomic type that are implemented with lock-free operations (much faster than locks)

```
#include <atomic> // chrono, iostream, thread, vector
void f(std::atomic<int>& value) {
    for (int i = 0; i < 10; i++) {
        value++;
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
int main() {
    std::atomic<int> value(0);
    std::vector<std::thread> th_vect;
    for (int i = 0; i < 100; i++)
        th_vect.push_back( std::thread(f, std::ref(value)) );
    for (auto& it : th_vect)
        it.join();
    std::cout << value;     // print 1000
}
```

The **`future`** library provides facilities to obtain values that are returned and to catch exceptions that are thrown by *asynchronous* tasks

Asynchronous call: `std::future async(function, args...)`
runs a function asynchronously (potentially in a new thread)
and returns a `std::future` object that will hold the result

`std::future` methods:

- `T get()` returns the result
- `wait()` waits for the result to become available

`async()` can be called with two launch policies for a task executed:

- `std::launch::async` a new thread is launched to execute the task asynchronously
- `std::launch::deferred` the task is executed on the calling thread the first time its result is requested (lazy evaluation)

Task-based parallelism


```
#include <future> // numeric, algorithm, vector, iostream
template <typename RandomIt>
int parallel_sum(RandomIt beg, RandomIt end) {
    auto len = end - beg;
    if (len < 1000)     // base case
        return std::accumulate(beg, end, 0);

    RandomIt mid = beg + len / 2;
    auto handle = std::async(std::launch::async, // right side
                             parallel_sum<RandomIt>, mid, end);
    int sum = parallel_sum(beg, mid);             // left side
    return sum + handle.get();                    // left + right
}
int main() {
    std::vector<int> v(10000, 1); // init all to 1
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end());
}
```