Modern C++ Programming

18. Advanced Topics II

Federico Busato

2023-04-02

Table of Context

1 **Undefined Behavior**

2 **Error Handing**

- C++ Exceptions
- Defining Custom Exceptions
- `noexcept` Keyword
- Memory Allocation Issues
- Alternative Error Handling Approaches

3 C++ Idioms

- Rule of Zero/Three/Five
- Singleton
- PIMPL
- CRTP
- Template Virtual Functions

Table of Context

4 **Smart pointers**

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

5 **Concurrency**

- Thread Methods
- Mutex
- Atomic
- Task-based parallelism

Undefined Behavior

Undefined Behavior Overview

Undefined behavior means that the semantic of certain operations is undefined (outside the language/library specification) or illegal, and the compiler presumes that such operations never happen

Motivations behind undefined behavior:

- *Compiler optimizations*, e.g. signed overflow or NULL pointer deferencing
- *Simplify compile checks*

Some undefined behavior cases provide an *implementation-defined behavior* depending on the compiler and platform. In this case, the code is *not portable*

- `What Every C Programmer Should Know About Undefined Behavior`
- `What are all the common undefined behaviours that a C++ programmer should know about?`

Undefined Behavior Common Cases

- `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;
// use var
auto var2 = new int;
// use var2
```

Undefined Behavior Common Cases 2/4

- **Memory access-related**
  - NULL pointer deferencing
  - Out-of-bound access
- **Platform specific behavior**
  - Endianness

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

  - Type definition

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

  - Intrinsic functions

- **Strict aliasing**

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

- **Lifetime issues**

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

- **Unspecified behavior**
  - A legal operation but the C++ standard does not document the results
  - Signed shift `-2 << x` (before C++20), large-than-type shift `3 << 32`, signed overflow, etc.
  - Operation ordering `f(i++, i++)`
- **One Definition Rule violation**
  - Different definitions of `inline` functions in distinct translation units
- **Missing `return` statement**

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

Detecting Undefined Behavior

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

- Using GCC/Clang undefined behavior sanitizer (run-time check)
- Static analysis tools
- Use `constexpr` expressions as undefined behavior is not allowed

```cpp
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 indicates *"exceptional"* run-time conditions. e.g. file not found, bad allocation, wrong user input, etc.

The common ways for handling recoverable errors are:

Exceptions Robust but slower and requires more resources

Error values Fast but difficult to handle in complex programs

---

- 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
- C++ exceptions are becoming more and more problematic

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)

- **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

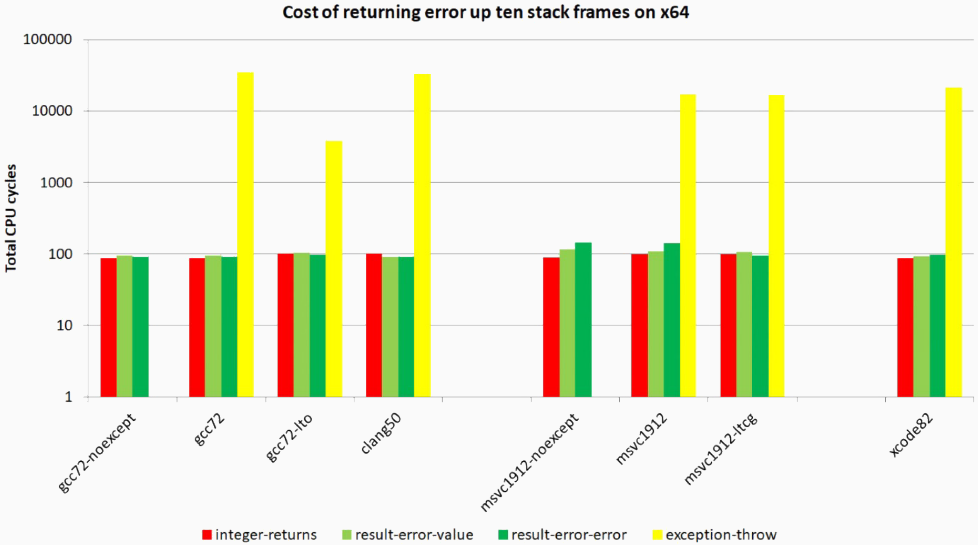
- **Performance:** Extreme performance overhead in the failure case (violate the zero-overhead principle)

- **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>`

```cpp
#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

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

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`

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

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

```cpp
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
};
```

Alternative Error Handling Approaches

- **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

C++ Idioms

Rule of Zero

The **Rule of Zero** is a rule of thumb for C++

Utilize the *value semantics* of existing types to avoid having to implement *custom* copy and move operations

Note: many classes (such as `std` classes) manage resources themselves and should not implement copy/move constructor and assignment operator

```
class X {
public:
    X(...); // constructor
            // NO need to define copy/move semantic
private:
    std::vector<int>     v; // instead raw allocation
    std::unique_ptr<int> p; // instead raw allocation
};                          // see smart pointer
```

Rule of Three

The **Rule of Three** is a rule of thumb for C++(03)

> If your class needs <u>any</u> of
>
> - a copy constructor `X(const X&)`
> - an assignment operator `X& operator=(const X&)`
> - or a destructor `~X()`
>
> defined explicitly, then it is likely to need <u>all</u> three of them

Some resources <u>cannot</u> or <u>should not</u> be copied. In this case, they should be declared as deleted

```
X(const X&) = delete
X& operator=(const X&) = delete
```

Rule of Five

The **Rule of Five** is a rule of thumb for C++11

If your class needs <u>any</u> of

- a copy constructor `X(const X&)`
- a move constructor `X(X&&)`
- an assignment operator `X& operator=(const X&)`
- an assignment operator `X& operator=(X&&)`
- or a destructor `~X()`

defined explicitly, then it is likely to need <u>all</u> five of them

Singleton

Singleton is a software design pattern that restricts the instantiation of a class to one and only one object (a common application is for logging)

```
class Singleton {
public:
    static Singleton& get_instance() {  // note "static"
        static Singleton instance { ..init.. } ;
        return instance;    // destroyed at the end of the program
    }                       // initiliazed at first use

    Singleton(const& Singleton)      = delete;
    void operator=(const& Singleton) = delete;

    void f() {}
private:
    T _data;

    Singleton( ..args.. ) { ... }  // used in the initialization
}
```

Pointer to IMPLementation (PIMPL) idiom allows decoupling the interface from the implementation in a clear way

`header.hpp`

```cpp
class A {
public:
    A();
    ~A();
    void f();
private:
    class Impl;  // forward declaration
    Impl* ptr;   // opaque pointer
};
```

NOTE: The class does not expose internal data members or methods

PIMPL - Implementation

`source.cpp` (`Impl` actual implementation)

```cpp
class A::Impl { // could be a class with a complex logic
public:
    void internal_f() {
        ..do something..
    }
private:
    int   _data1;
    float _data2;
};

A::A()     : ptr{new Impl()} {}
A::~A()     { delete ptr; }
void A::f() { ptr->internal_f(); }
```

PIMPL - Advantages, Disadvantages

Advantages:

- ABI stability
- Hide private data members and methods
- Reduce compile type and dependencies

Disadvantages:

- Manual resource management
  - `Impl* ptr` can be replaced by `unique_ptr<impl> ptr` in C++11
- Performance: pointer indirection + dynamic memory
  - dynamic memory could be avoided by using a reserved space in the interface e.g. `uint8_t data[1024]`

PIMPL - Implementation Alternatives

What parts of the class should go into the `Impl` object?

- *Put all private and protected members into* `Impl`:
  Error prone. Inheritance is hard for opaque objects

- *Put all private members (but not functions) into* `Impl`:
  Good. Do we need to expose all functions?

- *Put everything into* `Impl`*, and write the public class itself as only the public interface, each implemented as a simple forwarding function*:
  Good

https://herbsutter.com/gotw/_100/

The **Curiously Recurring Template Pattern (CRTP)** is an idiom in which a class `X` derives from a class template instantiation using `X` itself as template argument

A common application is *static polymorphism*

```
template <class T>
struct Base {
    void my_method() {
        static_cast<T*>(this)->my_method_impl();
    }
};

class Derived : public Base<Derived> {
//  void my_method() is inherited
    void my_method_impl() { ... } // private method
};
```

```cpp
#include <iostream>
template <typename T>
struct Writer {
    void write(const char* str) {
        static_cast<const T*>(this)->write_impl(str);
    }
};
class CerrWriter : public Writer<CerrWriter> {
    void write_impl(const char* str) {  std::cerr << str;  }
};
class CoutWriter : public Writer<CoutWriter> {
    void write_impl(const char* str) {  std::cout << str;  }
};
CoutWriter x;
CerrWriter y;
x.write("abc");
y.write("abc");
```

```
template <typename T>
void f(Writer<T>& writer) {
    writer.write("abc);
}

CoutWriter x;
CerrWriter y;
f(x);
f(y);
```

Virtual functions cannot have template arguments, but they can be emulated by using the following pattern

```
class Base {
public:
    template<typename T>
    void method(T t) {
        v_method(t);    // call the actual implementation
    }
protected:
    virtual void v_method(int t)    = 0; // v_method is valid only
    virtual void v_method(double t) = 0; // for "int" and "double"
};
```

Actual implementations for derived class `A` and `B`

```
class AImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for A
        std::cout << "A " << t << std::endl;
    }
};

class BImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for B
        std::cout << "B " << t << std::endl;
    }
};
```

```cpp
template<class Impl>
class DerivedWrapper : public Impl {
private:
    void v_method(int t) override {
        Impl::t_method(t);
    }
    void v_method(double t) override {
        Impl::t_method(t);
    } // call the base method
};

using A = DerivedWrapper<AImpl>;
using B = DerivedWrapper<BImpl>;
```

```cpp
int main(int argc, char* argv[]) {
    A a;
    B b;
    Base* base = nullptr;

    base = &a;
    base->method(1);   // print "A 1"
    base->method(2.0); // print "A 2.0"

    base = &b;
    base->method(1);   // print "B 1"
    base->method(2.0); // print "B 2.0"
}
```

`method()` calls `v_method()` (pure virtual method of `Base`)
`v_method()` calls `t_method()` (actual implementation)

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 <u>not shared</u> by multiple objects

```
#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

```cpp
#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

```
#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

```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'
```

`std::weak_ptr` - Weak Pointer 2/3

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

```cpp
#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 from the thread handle

```cpp
#include <chrono>   // the following program should (not deterministic)
#include <iostream> // produces the output:
#include <thread>   //    child thread exit
                    //    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); // t2
        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);    // t1
    std::cout << "main thread exit\n";
}
// if t1 < t2 the should program prints:
```

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 <iostream>
#include <thread>
void f(int& a, const int& b) {
    a = 7;
    const_cast<int&>(b) = 8;
}
int main() {
    int a = 1, b = 2;
    std::thread th1(f, a, b);                    // wrong!!!
    std::cout << a << ", " << b << std::endl;     // print 1, 2!!

    std::thread th2(f, std::ref(a), std::cref(b)); // correct
    std::cout << a << ", " << b << std::endl;     // print 7, 8!!
    th1.join(); th2.join();
}
```

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

```
#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;
}
```

Mutex

C++11 provide 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

```cpp
#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();
    std::cout << value;
}
```

Atomic

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

```cpp
#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)

```cpp
#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());
}
```