四川大學
SICHUAN UNIVERSITY

Lecture11 Advanced Topics

Qijun Zhao
College of Computer Science
Sichuan University

Spring 2023

Table of Context

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.

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

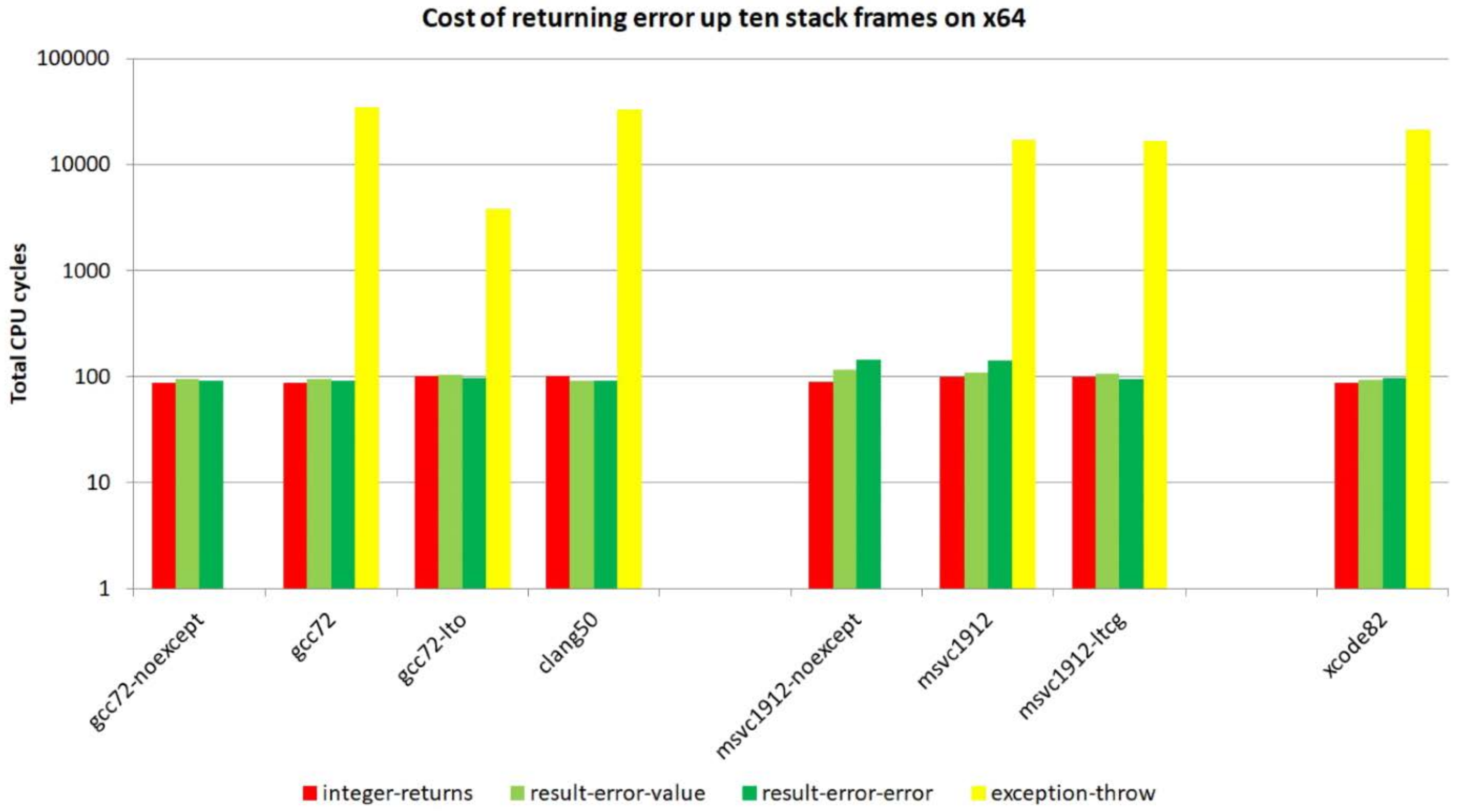
C++ Exceptions provide a well-defined mechanism to detect errors passing the information up the call stack

- **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)

C++ Exceptions - Disadvantages

- **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%

C++ Exceptions - Disadvantages



C++ Exception Basics

C++ provides three keywords for exception handling:

throw Throws an exception

try Code block containing potential throwing expressions

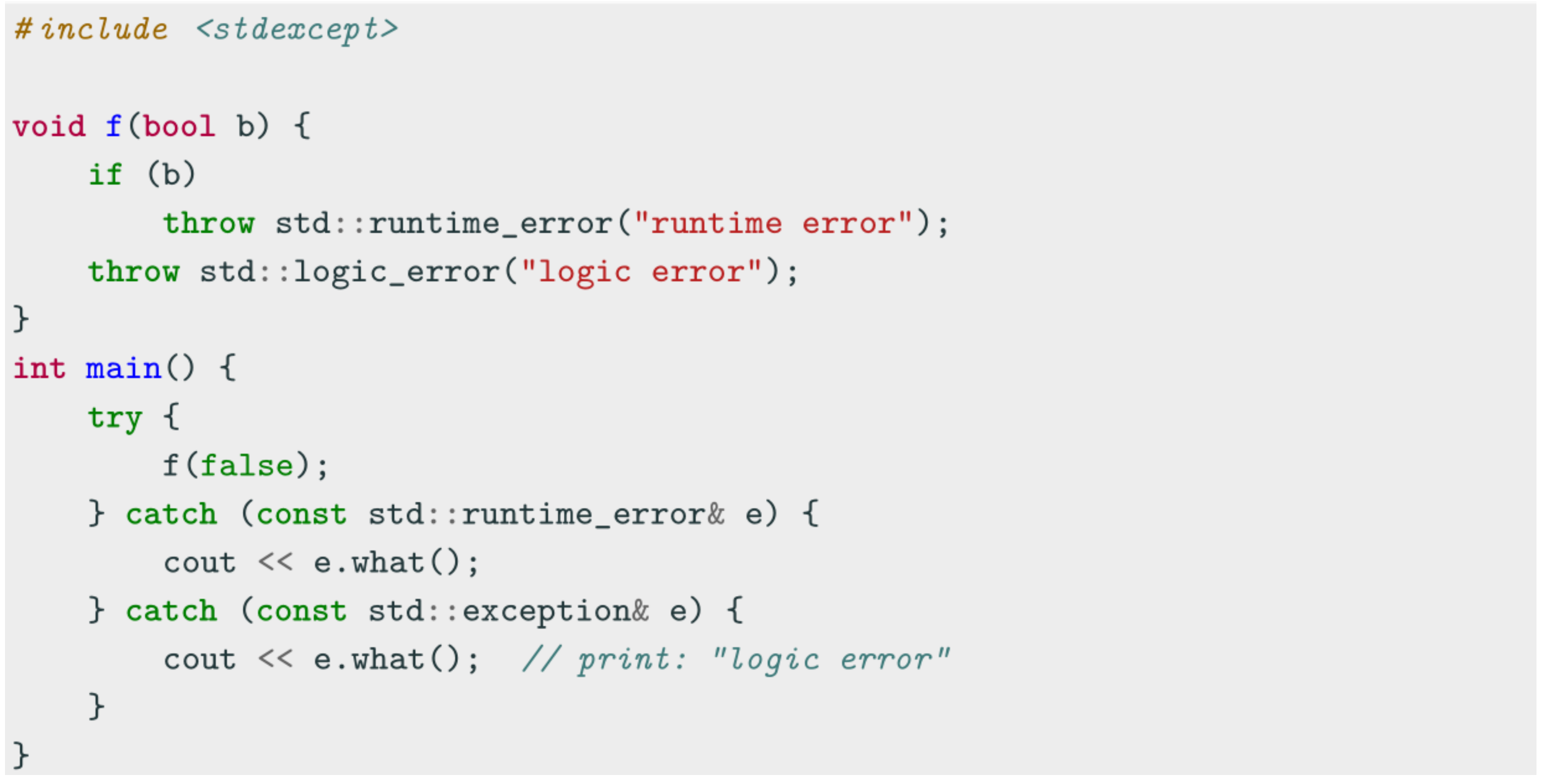
catch Code block for handling the exception

```
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

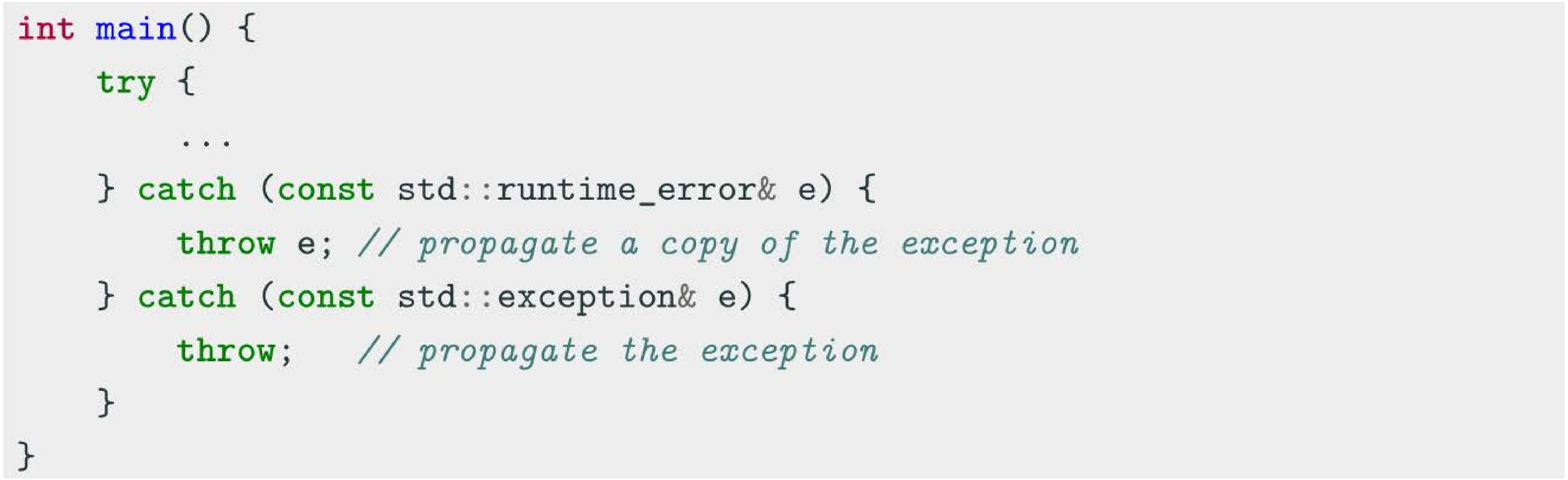
The right way to capture an exception is by `const` -reference. Capturing by-value is also possible but, it involves useless copy for non-trivial exception objects

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

```cpp
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



```
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

```
#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
    ...
}
```

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;
}
```

Memory Allocation Issues

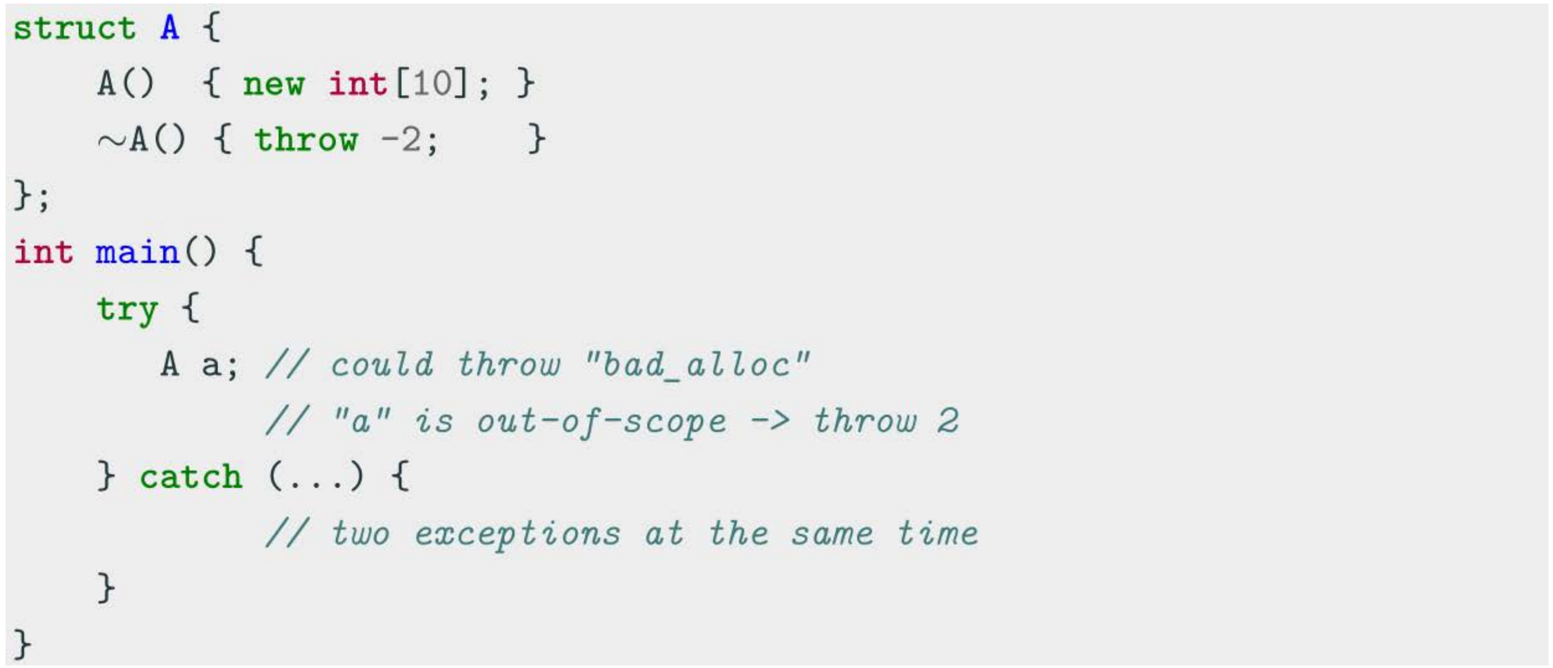
C++ also provides an overload of the **new** operator with non-throwing memory allocation

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

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

Memory Allocation Issues

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

Memory Allocation Issues

```
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
};
```

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

- `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

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 from the thread handle

Thread Methods

```
#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

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

Mutex (The Problem)

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

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

```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)

```
#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
}
```

Task-based parallelism

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

```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());
}
```

Control Flow

Computation is faster than decision

Pipelines are an essential element in modern processors. Some processors have up to 20 pipeline stages (14/16 typically)

The downside to long pipelines includes the danger of **pipeline stalls** that waste CPU time, and the time it takes to reload the pipeline on **conditional branch** operations (`if` , `while` , `for`)

Control Flow

- Prefer `switch` statements instead of multiple `if`
  - If the compiler does not use a jump-table, the cases are evaluated in order of appearance → the most frequent cases should be placed before
  - Some compilers (e.g. clang) are able to translate a sequence of `if` into a `switch`
- Prefer **square brackets** syntax `[]` over pointer arithmetic operations for array access to facilitate compiler loop optimizations (polyhedral loop transformations)
- Prefer **signed integer** for **loop indexing**. The compiler optimizes more aggressively such loops since integer overflow is not defined
- Prefer range-based loop for iterating over a container

Control Flow

- In general, if statements affect performance when the branch is taken

- Some compilers (e.g. clang) use assertion for optimization purposes: most likely code path, not possible values, etc.

- Not all control flow instructions (or branches) are translated into jump instructions. If the code in the branch is small, the compiler could optimize it in a conditional instruction, e.g. ccmovl

Small code section can be optimized in different ways (see next slides)

Minimize Branch Overhead

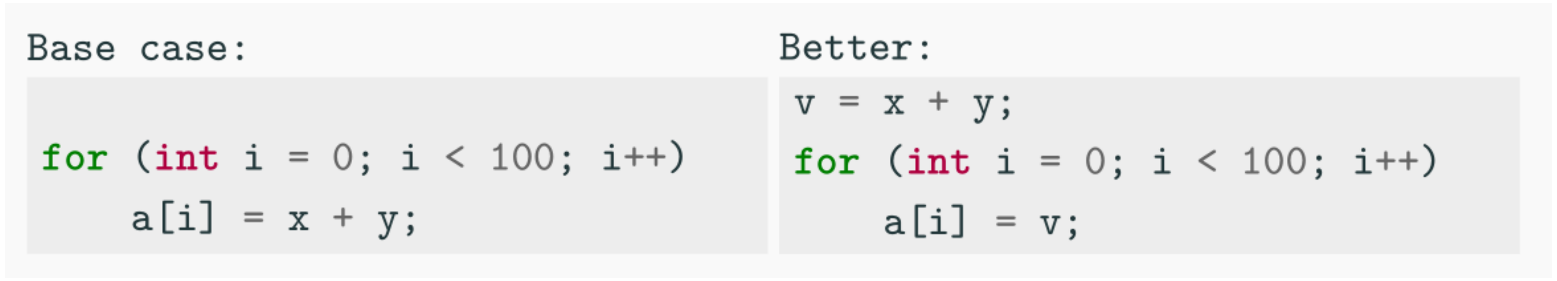
- **Branch prediction**: technique to guess which way a branch takes. It requires hardware support and it is generically based on dynamic history of code executing
- **Branch predication**: a conditional branch is substituted by a sequence of instructions from both paths of the branch. Only the instructions associated to a predicate (boolean value), that represents the direction of the branch, are actually executed

```
int x = (condition) ? A[i] : B[i];
P = (condition) // P: predicate
@P  x = A[i];
@!P x = B[i];
```

- **Speculative execution**: execute both sides of the conditional branch to better utilize the computer resources and commit the results associated to the branch taken

Loop Hoisting

Loop Hoisting, also called loop-invariant code motion, consists of moving statements or expressions outside the body of a loop without affecting the semantics of the program



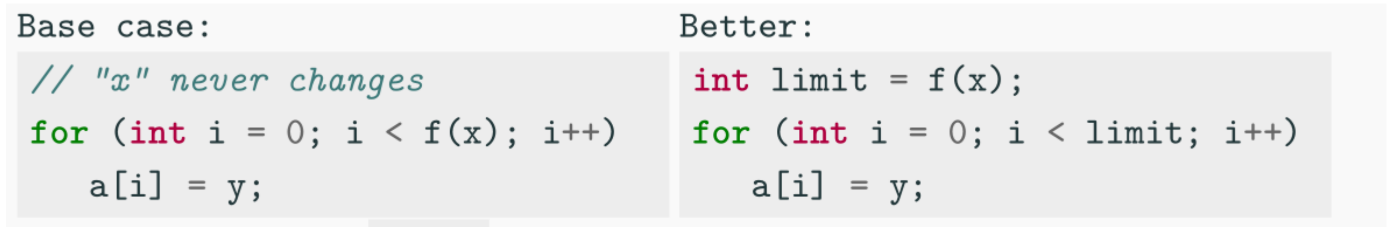
Base case:

```
for (int i = 0; i < 100; i++)
    a[i] = x + y;
```

Better:

```
v = x + y;
for (int i = 0; i < 100; i++)
    a[i] = v;
```

Loop hoisting is also important in the evaluation of loop conditions



Base case:

```
// "x" never changes
for (int i = 0; i < f(x); i++)
    a[i] = y;
```

Better:

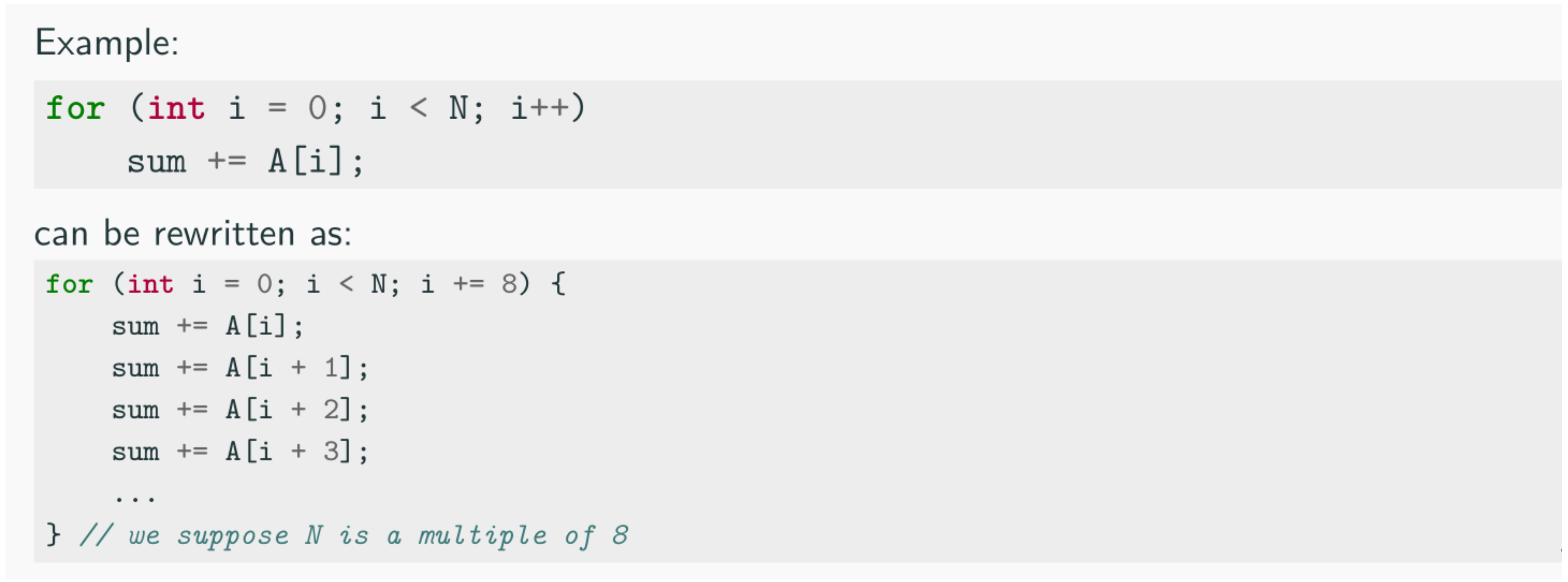
```
int limit = f(x);
for (int i = 0; i < limit; i++)
    a[i] = y;
```

In the worst case, f(x) is evaluated at every iteration (especially when it belongs to another translation unit)

Loop Unrolling

Loop unrolling (or **unwinding**) is a loop transformation technique which optimizes the code by removing (or reducing) loop iterations

The optimization produces better code at the expense of binary size



Example:

```
for (int i = 0; i < N; i++)
    sum += A[i];
```

can be rewritten as:

```
for (int i = 0; i < N; i += 8) {
    sum += A[i];
    sum += A[i + 1];
    sum += A[i + 2];
    sum += A[i + 3];
    ...
} // we suppose N is a multiple of 8
```

Loop Unrolling

Loop unrolling notes:

+ Improve instruction-level parallelism (ILP)

+ Allow vector (SIMD) instructions

+ Reduce control instructions and branches

- Increase compile-time/binary size

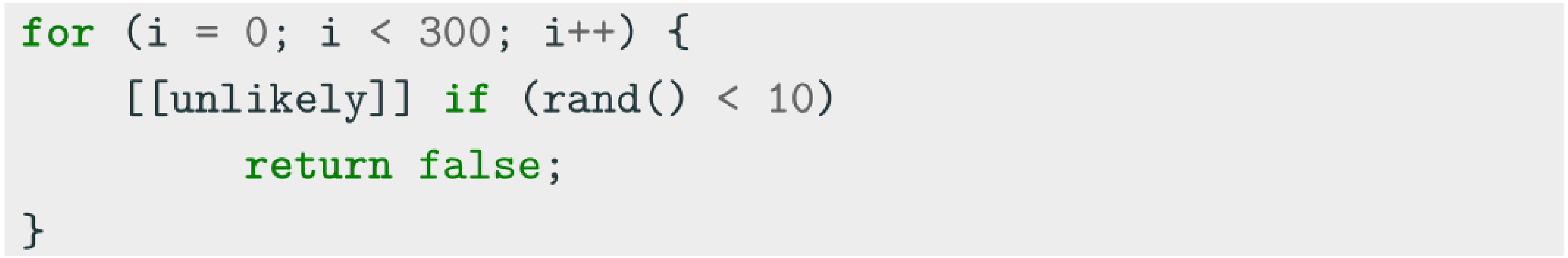
- Require more instruction decoding

- Use more memory and instruction cache

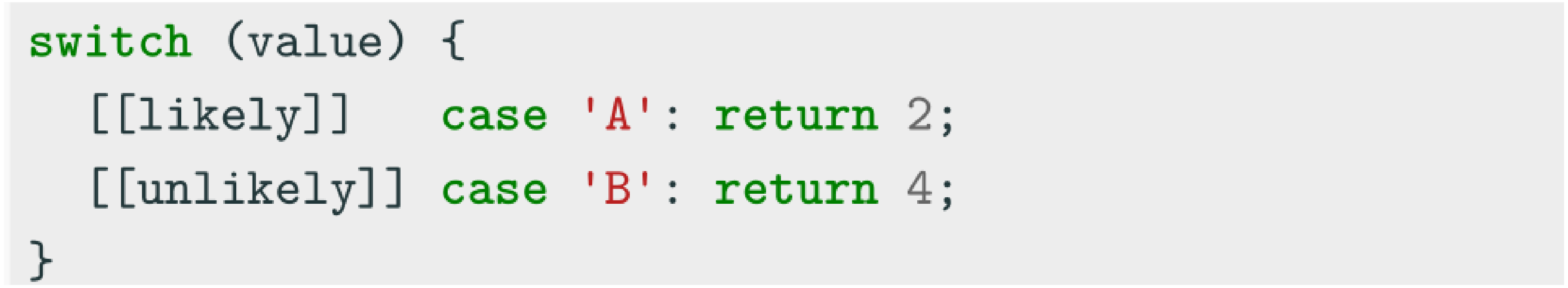
Unroll directive The Intel, IBM, and clang compilers (but not GCC) provide the preprocessing directive #pragma unroll (to insert above the loop) to force loop unrolling. The compiler already applies the optimization in most cases

Branch Hints

C++20 [[likely]] and [[unlikely]] provide a hint to the compiler to optimize a conditional statement, such as while , for , if



```
for (i = 0; i < 300; i++) {
    [[unlikely]] if (rand() < 10)
        return false;
}
```



```
switch (value) {
  [[likely]]   case 'A': return 2;
  [[unlikely]] case 'B': return 4;
}
```

Recursion

Avoid run-time recursion (very expensive). Prefer *iterative* algorithms instead (see next slides)

Recursion cost: The program must store all variables (snapshot) at each recursion iteration on the stack, and remove them when the control return to the caller instance

The **tail recursion** optimization avoids to maintain caller stack and pass the control to the next iteration. The optimization is possible only if all computation can be executed before the recursive call

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn