Lecture 6

TOC

异常

异常是一种程序运行时的错误处理机制，用于处理程序运行时的异常情况。C++ 中的异常处理机制是通过 `try-catch` 语句实现的。

```cpp
#include <iostream>
#include <stdexcept>

int main() {
    try {
        throw std::runtime_error("An error occurred");
    } catch (const std::exception &e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

通过 `throw` 关键字可以人为抛出异常，`catch` 关键字用于捕获异常。标准库提供了 `std::exception`，通过创建对应的派生类，可以通过 `what()` 方法获取异常信息。

- 异常的类型可以是任意类型，会从上到下按照类型匹配异常。如果没有匹配的异常类型，异常会被传递到上一层调用栈，直到被捕获。
- 多层 catch 块时，异常会被第一个匹配的 catch 块捕获。
- 如果异常没有被捕获，程序会调用 `std::terminate` 终止。

```
try {
    throw 1;
} catch (int e) {
    std::cerr << "Catch int: " << e << std::endl;
} catch (double e) {
    std::cerr << "Catch double: " << e << std::endl;
}
```

栈解退(stack unwinding)

当异常被抛出时，程序会在调用栈中查找匹配的 catch 块，如果找到匹配的 catch 块，程序会执行 catch 块中的代码，然后继续执行。 若当前调用栈中没有匹配的 catch 块，异常会被传递到上一层调用栈，同时该层调用栈中的局部变量会被销毁，直到找到匹配的 catch 块或触发 `std::terminate`

析构函数一定要为 `noexcept`，否则如果在栈解退过程中析构函数抛出异常，程序会调用 `std::terminate` 终止。

noexcept

`noexcept` 是 C++11 引入的关键字，用于指明函数是否抛出异常。`noexcept` 可以用于函数声明和函数定义。

```
void foo() noexcept;         // foo 不会抛出异常
void bar() noexcept(false); // bar 可能抛出异常
static_assert(noexcept(foo()), "foo() should not throw exceptions");
static_assert(!noexcept(bar()), "bar() may throw exceptions");
```

`noexcept` 用于指明函数是否抛出异常，编译器会根据 `noexcept` 优化代码。如果函数声明为 `noexcept`，但实际抛出异常，程序会调用 `std::terminate` 终止。

模板

模板是 C++ 的一种重要特性，它是泛型编程的基础。模板使得编写泛型代码变得更加容易，减轻了代码的重复性。

eg:

```cpp
template <typename T>
T getMax(T a, T b) {
    return a > b ? a : b;
}

int main() {
    std::cout << getMax(1, 2) << std::endl;
    std::cout << getMax(1.1, 2.2) << std::endl;
    std::cout << getMax('a', 'b') << std::endl;
    return 0;
}
```

模板的定义以关键字 `template` 开始，后面跟着模板参数列表，模板参数列表以 `<>` 括起来，模板参数列表中的参数可以是类型参数，也可以是非类型参数(如整型常量, const char* 等)。

模板函数和模板类

模板函数,模板类的定义方法类似

```cpp
#include <iostream>
template <typename T> T getMax(T a, T b) { return a > b ? a : b; }
// 在早期的 C++ 代码中，使用 class 而不是 typename；直到 C++98 标准，typename 才被引入。

template <typename A, typename B> class Pair {
  A first_;
  B second_;
  int useless_{1023};

public:
  Pair(A a, B b) : first_(a), second_(b) {}
  A first() const { return first_; }
  B second() const { return second_; }
};

int main() {
  std::cout << getMax(1, 2) << std::endl;
  Pair<int, double> p(1, 2.2);
  std::cout << p.first() << " " << p.second() << std::endl;
  return 0;
}
```

模板实例化

模板函数和模板类的实例化是在编译期完成的，编译器会根据模板参数的类型生成对应的函数或类。

- 显式实例化： 显示指明模板参数类型，编译器会生成对应的函数或类。

```
extern template int getMax<int>(int, int);
extern template class Pair<int, double>;

// some.cpp
template int getMax<int>(int, int);
template class Pair<int, double>;
```

- 隐式实例化（默认行为）：编译器会根据调用的参数类型自动实例化模板函数。C++17 之后，编译器支持类模板参数自动推导。

```
std::cout << getMax(1, 2) << std::endl;
Pair p(1, 2.2);
```

由于模板函数和模板类的实例化是在编译期完成的，因此模板函数和模板类的定义通常放在头文件中。多个编译单元中的相同实例化模板不会违反ODR。

模板特化

模板特化是指对模板的一个或多个特定类型提供特定实现。模板特化分为全特化和偏特化。

- 全特化：对模板的所有参数都提供特定实现。
- 偏特化：对模板的部分参数提供特定实现。

```
#include <iostream>
template <typename A, typename B> auto sum(A a, B b) -> decltype(a + b) {
  return a + b;
}

// □□□
template <> auto sum(int a, int b) -> decltype(a + b) { return a + b; }

// □□□
template <typename A> A sum(bool a, A b) { return 1 + b; }

int main() {
  std::cout << sum(1, 2) << std::endl;
  std::cout << sum(true, 2.2) << std::endl;
  return 0;
}
```

模板别名

使用 `using` 关键字可以定义模板别名，实现部分模板参数的固定。

```
template <typename A, typename B> struct Pair {
  A first_;
  B second_;
};

template <typename T> using FirstIntPair = Pair<int, T>;// typedef 无法实现部分模板参数的固定

int main() {
  FirstIntPair<double> p{1, 2.2};
  return 0;
}
```

Concepts&Constraints(C++20)

C++20 引入了 Concepts，用于约束模板参数的类型。

```cpp
#include <iostream>
#include <concepts>

// 定义一个 Concept，要求类型必须是整数类型
template <typename T>
concept Integral = std::is_integral_v<T>;

// 使用 Constraints 约束模板参数
template <Integral T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(3, 4) << std::endl;     // 合法
    std::cout << add(3.5, 4.2) << std::endl; // 非法，double 不满足 Integral 概念
    return 0;
}
```

在没有 Concepts 的时候，我们通常使用 SFINAE 来约束模板参数的类型。

多线程编程

多线程编程是一种并发编程的方式，可以充分利用多核处理器的性能。C++11 引入了多线程支持，提供了 `std::thread` 类来支持多线程编程。封装了各个不同平台的线程库，提供了统一的接口。

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

void threadFunc() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Hello, thread!" << std::endl;
}

int main() {
    std::thread t(threadFunc);
    t.join();
    std::cout << "Hello, main!" << std::endl;
    return 0;
}
```

线程创建

- `std::thread` 构造函数接受一个可调用对象，创建一个新线程。
- `std::thread::join()` 在当前线程等待对应线程结束。
- `std::thread::detach()` 分离线程，线程结束后自动回收资源。

```cpp
#include <iostream>
#include <thread>
class Foo {
  int a;
public:
  Foo(int a) : a(a) {}
  void print(const char *val) { std::cout << a << '\n' << val << std::endl; }
};

int main() {
  Foo foo{1};
  std::thread b(&Foo::print, &foo, "Hello, member function!");
  std::thread c([]() {
    std::cout << "Hello, thread" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::cout << "Thread finished" << std::endl;
  });
  b.join();
```

- 如果线程函数抛出异常，`std::terminate` 会被调用，程序会终止。
- 当 main 函数结束时，所有线程会被终止，未完成的 detach 线程会被强制终止。

一个 `std::thread` 对象只能被移动，不能被复制。被析构时，如果线程没有被 join 或 detach，程序会调用 `std::terminate` 终止。

线程同步

线程同步是多线程编程中的重要问题，多个线程访问共享资源时，需要保证线程安全。对于共享资源的访问有三种情况：

- 只读访问：不需要同步，线程安全。
- 只写访问：需要同步，保证写操作的原子性。
- 读写访问：需要同步，保证读写操作的原子性。

> 原子性：一个操作要么全部执行成功，要么全部不执行。

对于线程同步，标准库提供了多种同步机制，如互斥锁、条件变量、原子操作等。

互斥锁

互斥锁是一种最基本的同步机制，用于保护共享资源。C++11 提供了 `std::mutex` 类来支持互斥锁。

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

std::mutex mtx;

void threadFunc() {
    std::lock_guard<std::mutex> lock(mtx);
    std::cout << "Hello, thread!" << std::endl;
}

int main() {
    std::thread t(threadFunc);
    t.join();
    std::unique_lock<std::mutex> lock(mtx);
    std::cout << "Hello, main!" << std::endl;
    return 0;
}
```

为了避免忘记解锁，避免直接使用 `std::lock_guard`，使用标准库提供的 RAII 类来管理锁。

- `std::lock_guard`：自动加锁，作用域结束时自动解锁。最基本的 RAII 机制。
- `std::unique_lock`：提供更多的灵活性，可以手动加锁和解锁。
- `std::scooped_lock`：可以按统一顺序加锁，避免死锁。

读写锁

读写锁是一种特殊的互斥锁，用于读写分离的场景。C++11 提供了 `std::shared_mutex` 类来支持读写锁。

```
std::shared_mutex mtx;
std::shared_lock<std::shared_mutex> lock(mtx); // □□
std::unique_lock<std::shared_mutex> lock(mtx); // □□
```

在高并发读多写少的场景下，读写锁可以提高并发性能。因为读锁之间不会互斥，只有写锁和读锁之间互斥。

std::recursive_mutex

递归锁是一种特殊的互斥锁，允许同一个线程多次加锁。C++11 提供了 `std::recursive_mutex` 类来支持递归锁。

```
std::recursive_mutex mtx;
for (int i = 0; i < 10; ++i) {
    std::lock_guard<std::recursive_mutex> lock(mtx);
    std::cout << i << std::endl;
}
```

当一个线程加锁时，其他线程无法加锁，直到解锁。递归锁允许同一个线程多次加锁，但要保证解锁次数和加锁次数相同。

条件变量

条件变量是一种同步机制，用于线程之间的通信。C++11 提供了 `std::condition_variable` 类来支持条件变量。

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void threadFunc() {
    std::unique_lock<std::mutex> lock(mtx);
    ready = true;
    cv.notify_one();
}

int main() {
    std::thread t(threadFunc);
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    std::cout << "Hello, main!" << std::endl;
```

条件变量常用于生产者-消费者模型，生产者生产数据，消费者消费数据。生产者生产数据后通知消费者，消费者消费数据后通知生产者。

使用流程：

1. 占有互斥锁
2. 通过该互斥锁等待条件变量，此时会释放互斥锁
3. 其他线程执行完操作后，通知等待条件变量的线程
4. 等待条件变量的线程被唤醒，重新占有互斥锁，并检查是否满足条件。若不满足条件，则继续等待。

原子操作

原子操作是一种不可分割的操作，保证操作的原子性。C++11 提供了 `std::atomic` 来创建支持原子操作的变量。

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

std::atomic<int> counter{0};
counter.fetch_add(1);
counter++;
```

原子操作是 CPU 提供的硬件级别的操作，如 CAS (Compare And Swap) 操作，保证操作的原子性。

```
std::atomic<int> counter{0};
while (counter.compare_exchange_strong(0, 1)) {
    std::cout << "CAS failed, Retry" << std::endl;
}
```

通过原子操作，可以实现无锁编程，在某些场景下提高并发性能。

Q&A