C++

Урок 29

Mutex

Mutex

Определение

Мьютекс («взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти.

Идея

В программе наступает момент **барьерной синхронизации** (построение потоков). Для этого построения и нужен Mutex.

Mutex = регулировщик, который в определенный момент поднимает ключ и говорит ”стоять” остальным потокам. Как только поток завершил свое действие, он сообщает регулировщику, что остальные потоки могут продолжать.

Mutex – объект для синхронизации потоков.

Mutex

Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно).

Мьютекс отличается от семафора общего вида тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.

Mutex

Задачи

- В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом.
- Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

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

std::mutex mutex;

void thread_func1(std::vector<int> &x)
{

    x.push_back(0);
}
void thread_func2(std::vector<int> &x)
{
    x.pop_back();
}

int main()
{
    std::vector<int> vec;
    std::thread th1([&]
                    { thread_func1(vec); });
    std::thread th2([&]
                    { thread_func2(vec); });

    for (auto &it : vec)
    {
        std::cout << it << " ";
    }
    th1.join();
    th2.join();
    return 0;
}
```

Какая проблема тут есть?

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

std::mutex mutex;

void thread_func1(std::vector<int> &x)
{

    x.push_back(0);
}
void thread_func2(std::vector<int> &x)
{
    x.pop_back();
}

int main()
{
    std::vector<int> vec;
    std::thread th1([&]
                    { thread_func1(vec); });
    std::thread th2([&]
                    { thread_func2(vec); });

    for (auto &it : vec)
    {
        std::cout << it << " ";
    }
    th1.join();
    th2.join();
    return 0;
}
```

Есть шанс получить **ошибку сегментации**

Mutex

Основные действия:

- Объявление | `std::mutex mutex_name`;
- Захват мьютекса | `mutex_name.lock();`
  Поток запрашивает монопольное использование общих данных, защищаемых мьютексом. Дальше два варианта развития событий: происходит захват мьютекса этим потоком (и в этом случае ни один другой поток не сможет получить доступ к этим данным) или поток блокируется (если мьютекс уже захвачен другим потоком).
- Метод `try_lock` пытается получить права владения мьютексом без блокировки. Его возвращаемое значение можно преобразовать в bool и оно является true, если метод получает права владения; в противном случае — false.
- Освобождение мьютекса | `mutex_name.unlock();`
  Когда ресурс больше не нужен, текущий владелец должен вызвать функцию разблокирования unlock, чтобы и другие потоки могли получить доступ к этому ресурсу. Когда мьютекс освобождается, доступ предоставляется одному из ожидающих потоков.

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

std::mutex mutex;

void thread_func1(std::vector<int> &x)
{
    mutex.lock();
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    x.push_back(0);
    mutex.unlock();
}
void thread_func2(std::vector<int> &x)
{
    mutex.lock();
    x.pop_back();
    mutex.unlock();
}

int main()
{
    std::vector<int> vec;
    std::thread th1([&]
                    { thread_func1(vec); });
    std::thread th2([&]
                    { thread_func2(vec); });

    for (auto &it : vec)
    {
        std::cout << it << " ";
    }
    th1.join();
    th2.join();
    return 0;
}
```

Решение проблемы с mutex

Lock_guard

Lg

- — обертка
- конструктор вызывает метод lock для заданного объекта, а деструктор вызывает unlock
- в конструктор класса std::lock_guard можно передать аргумент std::adopt_lock - индикатор, означающий, что mutex уже заблокирован и блокировать его заново не надо
- std::lock_guard не содержит никаких других методов, его нельзя копировать, переносить или присваивать

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

std::mutex mutex;

void thread_func1(std::vector<int> &x)
{
    std::lock_guard<std::mutex> lock(mutex);

    x.push_back(0);
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
}
void thread_func2(std::vector<int> &x)
{
    mutex.lock();
    x.pop_back();
    mutex.unlock();
}

int main()
{
    std::vector<int> vec;
    std::thread th1([&]
                    { thread_func1(vec); });
    std::thread th2([&]
                    { thread_func2(vec); });

    th1.join();
    th2.join();
    return 0;
}
```

lock_guard

Обработка исключений

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(int);

int main()
{
    int val = -1;
    double rez = sqrt_val(val);

    std::cout << rez << std::endl;

    return 0;
}
double sqrt_val(int test)
{
    return sqrt(test);
}
```

Скомпилируется ли программа?

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(int);

int main()
{
    int val = -1;
    double rez = sqrt_val(val);

    std::cout << rez << std::endl;

    return 0;
}
double sqrt_val(int test)
{
    return sqrt(test);
}
```

ДА

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(int);

int main()
{
    int val = -1;
    double rez = sqrt_val(val);

    std::cout << rez << std::endl;

    return 0;
}
double sqrt_val(int test)
{
    return sqrt(test);
}
```

Что будет выведено?

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(int);

int main()
{
    int val = -1;
    double rez = sqrt_val(val);

    std::cout << rez << std::endl;

    return 0;
}
double sqrt_val(int test)
{
    return sqrt(test);
}
```

Что будет выведено? nan

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(int);

int main()
{
    int val = -1;
    double rez = sqrt_val(val);

    std::cout << rez << std::endl;

    return 0;
}
double sqrt_val(int test)
{
    return sqrt(test);
}
```

Что будет выведено?

Как можно это предусмотреть?

```
double sqrt_val(int test)
{
    if (test < 0)
        test = 0;
    return sqrt(test);
}

double sqrt_val(int test)
{
    if (test < 0)
        return "Число меньше 0";
    return sqrt(test);
}

double sqrt_val(int test)
{
    if (test < 0)
        std::cout << "Число меньше 0" << std::endl;
    return sqrt(test);
}
```

Чем такие варианты реализации плохи?

Обработка исключений

Исключение — это условие ошибки, возможно вне элемента управления программы, которое не позволяет продолжать выполнение программы по обычному пути выполнения.

Операции, включая создание объектов, входные и выходные данные файлов и вызовы функций из других модулей, являются потенциальными источниками исключений, даже если программа работает правильно. В надежном коде можно **предвидеть** и **обработать** исключения.

Обработка исключений

Механизм: чтобы поймать исключение, его нужно где-то (внутри ф-ии) бросать (**throw**), затем проверять эту ф-ию внутри **try** и ,если возникнет исключение, то отреагировать на него в **catch**.

`Try` позволяет определить блок кода, который будет **проверяться** на **наличие ошибок** во время его выполнения;

`Throw` нужен для создания и **отображения исключений** и используется для перечисления ошибок, которые генерирует функция, но не может самостоятельно обрабатывать исключения;

`Catch` - блок кода, который выполняется при **возникновении** определенного **исключения** в блоке try

Обработка исключений

```
try
{
    // проверяем (генерируем) исключительную ситуацию
}
catch (const std::exception &e) // передача информации из try
{
    // ловим и проводим обработку
    std::cout << e.what() << std::endl;
}
```

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(const int &);

int main()
{
    int val = -1;
    try
    {
        double rez = sqrt_val(val);
        std::cout << rez << std::endl;
    }
    catch (const char *msg)
    {
        std::cout << "Ошибка: " << msg << std::endl;
    }
    std::cout << "Завершение работы" << std::endl;
    return 0;
}
double sqrt_val(const int &test)
{
    if (test < 0)
        throw "Корень из отриц числа!";
    return sqrt(test);
}
```

Обработали исключение типа ”строка” при отрицательном числе

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(const int &);

int main()
{
    int val = -1;
    try
    {
        double rez = sqrt_val(val);
        std::cout << rez << std::endl;
    }
    catch (const char *msg)
    {
        std::cout << "Ошибка: " << msg << std::endl;
    }

    catch (const int &value)
    {
        std::cout << "При введенном числе: " << value << " произошла обибка" << std::endl
;
    }
    std::cout << "Завершение работы" << std::endl;
    return 0;
}
double sqrt_val(const int &test)
{
    if (test < 0)
        throw(test);
    return sqrt(test);
}
```

Обработали исключение типа ”целое число” при отрицательном числе

```cpp
#include <iostream>
#include <cmath>

double sqrt_val(const int &);

int main()
{
    int val = -1;
    try
    {
        double rez = sqrt_val(val);
        std::cout << rez << std::endl;
    }

    catch (const char *msg)
    {
        std::cout << "Ошибка: " << msg << std::endl;
    }

    catch (const int &value)
    {
        std::cout << "При введенном числе: " << value << " произошла обибка" << std::endl;
    }

    catch (const std::exception &e)
    {
        std::cout << "Ошибка: " << e.what() << std::endl;
    }

    std::cout << "Завершение работы" << std::endl;
    return 0;
}
double sqrt_val(const int &test)
{
    if (test < 0)
        throw std::exception();
    return sqrt(test);
}
```

Обработали исключение объекта класса std::exeption

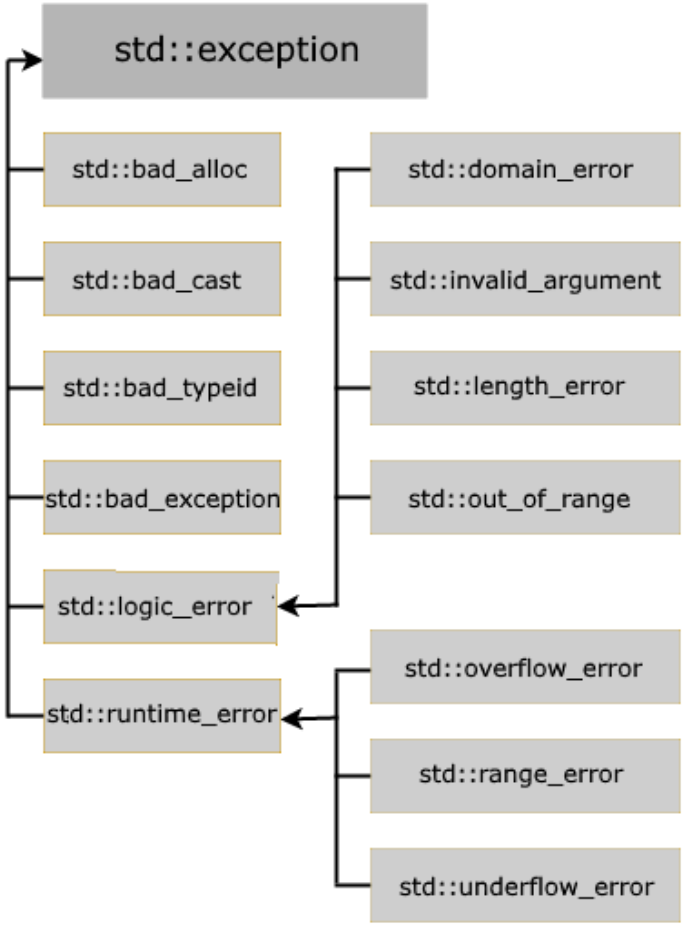
Обработка исключений

Причины использования

- Исключение заставляет вызывающий код **распознавать** условие ошибки и **обрабатывать** его. Необработанное исключение останавливает выполнение программы.
- Исключение переходит к точке стека вызовов, которая может обрабатывать ошибку. Промежуточные функции могут позволить распространению исключения. Они не должны координироваться с другими слоями.
- Механизм очистки стека исключений уничтожает все объекты в области после создания исключения в соответствии с четко определенными правилами.
- Исключение позволяет четко **разделить** код, который обнаруживает ошибку и код, обрабатывающий ошибку.

Обработка исключений

Типы исключений



Обработка исключений

Типы исключений

- **runtime_error:** общий тип исключений, которые возникают во время выполнения;
- **range_error:** исключение, которое возникает, когда полученный результат превосходит допустимый диапазон;
- **overflow_error**: исключение, которое возникает, если полученный результат превышает допустимый диапазон;
- **underflow_error:** исключение, которое возникает, если полученный в вычислениях результат имеет недопустимые отрицательное значение (выход за нижнюю допустимую границу значений);
- **logic_error:** исключение, которое возникает при наличии логических ошибок к коде программы;
- **domain_error:** исключение, которое возникает, если для некоторого значения, передаваемого в функцию, не определено результата;
- **invalid_argument**: исключение, которое возникает при передаче в функцию некорректного аргумента;
- **length_error**: исключение, которое возникает при попытке создать объект большего размера, чем допустим для данного типа;
- **out_of_range:** исключение, которое возникает при попытке доступа к элементам вне допустимого диапазона;

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

double sqrt_val(const int &);

int main()
{
    int val = -1;
    try
    {
        double rez = sqrt_val(val);
        std::cout << rez << std::endl;
    }

    catch (std::overflow_error err)
    {
        std::cout << "Overflow_error: " << err.what() << std::endl;
    }
    catch (std::runtime_error err)
    {
        std::cout << "Runtime_error: " << err.what() << std::endl;
    }
    catch (std::exception err)
    {
        std::cout << "Exception!!!" << std::endl;
    }

    std::cout << "Завершение работы" << std::endl;
    return 0;
}
double sqrt_val(const int &test)
{
    if (test < 0)
        throw std::runtime_error("value <0! ");
    return sqrt(test);
}
```

Обработали под тип исключения