datachunk.h

```
/*
 * datachunk.h
 *
 *  Created on: Oct 8, 2019
 *      Author: osboxes
 */

#ifndef DATACHUNK_H_
#define DATACHUNK_H_

extern int counter;
extern int copyCounter;

class data_chunk {
public:
    int i;
    data_chunk() : i(counter) {
        std::cout << "Constructor: " << i << std::endl;
        ++counter;
        ++copyCounter;
    };
    data_chunk(const data_chunk &d0) : i(d0.i) {
        std::cout << "Copy Constructor: " << i << std::endl;
        ++copyCounter;
    };

    virtual ~data_chunk() {
        std::cout << "Destructor: " << i << std::endl;
        --copyCounter;
    };
};

#endif /* DATACHUNK_H_ */
```

thread safe_queque

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

#define USE_MAKESHARED
#ifdef USE_MAKESHARED
#define make_ptr(type, towhat) std::make_shared<type>(towhat)
#define T_PTR std::shared_ptr<T>
#else
#define make_ptr(type, towhat) new type(towhat)
#define T_PTR T*
#endif

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
    std::condition_variable empty_cond;

    void popping() {
        data_queue.pop();
        if (data_queue.empty())
            empty_cond.notify_all();
    }

public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard lk(other.mut);
        data_queue=other.data_queue;
    }

    void push(T new_value)
    {
        std::lock_guard lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
```

→ did not have "delete" in the program
→ leaking memory

```
    }

    void drain()
    {
        std::unique_lock lk(mut);
        empty_cond.wait(lk,[this]{return data_queue.empty();});
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});

        value=data_queue.front();
        popping();
    }

    auto wait_and_pop()
    {
        std::unique_lock lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});

//      std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
//      auto              res(std::make_shared<T>(data_queue.front())); // changed by Craig
        auto              res(     make_ptr   (T, data_queue.front()));
        popping();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard lk(mut);
        if(data_queue.empty)
            return false;
        value=data_queue.front();
        popping();
        return true; // Added by Craig
    }

    T_PTR try_pop()
    {
        std::lock_guard lk(mut);
        if(data_queue.empty())
```

```
            return nullptr; // T_NULL;
//      std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
//      auto              res(std::make_shared<T>(data_queue.front())); // changed by Craig
        auto              res(     make_ptr   (T, data_queue.front()));
        popping();
        return res;
    }

    bool empty() const
    {
        std::lock_guard lk(mut);
        return data_queue.empty();
    }
};

//int main()
//{}
```

CraigMain.cpp

```
// Adapted by Craig Scratchley from Listing 4.1
#include <thread>
#include <iostream>
#include <memory>

#include "threadsafe_queue.hpp"
#include "datachunk.h"

using namespace std;

int counter = 1;
int copyCounter = 0;

bool more_data_to_prepare()
{
    return counter <= 3;
    //return true; // false;
}

struct data_chunk2
{};

data_chunk prepare_data()
{
    return data_chunk();
}

void process(data_chunk&)
{}

bool is_last_chunk(data_chunk&)
{
    return false; // true;
}

threadsafe_queue<data_chunk> ts_queue; // renamed from data_queue

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        ts_queue.push(data);  // copy of data in push func
```

```
        ts_queue.drain();
    }
}

void data_processing_thread()
{
//    while(true)
    {
        // examples of the use of shared pointers
        auto myRes = ts_queue.wait_and_pop();
      //auto myPtr(make_shared<data_chunk>(prepare_data()));
      //auto myPtr(new        data_chunk (prepare_data())); //we didn't call delete→ leaking
        auto myPtr(make_ptr   (data_chunk, prepare_data()));
        data_chunk data=*myRes;
        process(data);
        process(*myPtr);
        process(*ts_queue.wait_and_pop());
        //if(is_last_chunk(data))
        //    break;

        //myPtr = myRes; // the prepared data will be destroyed
        myPtr = nullptr;// delete myPtr
                        //NULL
        if(myPtr) { // but myPtr is now nullptr
            std::cout << "Won't go here!" <<
            (*myPtr).i <<    // NULL is at address 0 , you cannot deference
            myPtr->i << std::endl;
        }
        myRes = ts_queue.try_pop();
        myRes = ts_queue.try_pop();
    }
}

int main()
{
    std::thread t2(data_processing_thread); // re-ordered thread creation
    std::cout << "Between thread creation" << std::endl; // delay next thread creation
    std::thread t1(data_preparation_thread);

    t1.join();
    t2.join();
    std::this_thread::sleep_for (std::chrono::milliseconds(100));

    cout << "copyCounter:  " << copyCounter << endl;
}
```

- `-fsanitizer=address` : detect memory leak
- `-fsplit-stack`: avoid stack overflow
  - no longer contiguous stack
  - pieces of stack moving in the heap