C++ notes 2017 – 2018

Ondřej Měkota

24.10.2017

Storage classes (slide 43)

global, static member static local variables, string constants

- one instance (per process)
- static keyword
- static var is created on function call

thread local storage

- marked thred_local

automatic allocation

- no garbage collector, bad practice to let the os to unallocate

- new/delete operators
- smart pointers - automatically disallocates when the last pointer disappears

automatic storage - stack frames

- variables declared inside a function, deleted afterwards
- goto cannot jump to after initialization
- throw exception, break, goto,... deallocates used vars correctly !!

- dont place variables to saves space
- *exception should be used rarely*
- dynamic allocation is slow because the piece of memory have not been used for a while -> is not present in cache
- threads have their own ip,sp,registers, stacks, data memory, program memory

31.10.2017

dynamic allocation

- use smart pointers

```
include <memory>

//unique_ptr is as fast as T* pointer
std::unique_ptr<T> p = std::make_unique<T>();
std::unique_ptr<T> q = std::move(p); //p is nullptr

//slower.. reference counting
std::shared_ptr<T> p = std::make_shared<T>();
std::shared_ptr<T> q = p; //pointer copied
```

- *weak pointer is not included in reference counter. better to avoid cyclic structs than using weak ptrs*
- **observers** - pointers not claiming ownership T *
  - can be modifying or constant. const T *
- if function does not modify class members it has to be marked const *void f() const* > & returns observer pointer > * returns reference to the object
- *dont use new use smart pointers*

```
NOT T * p = new T;
//possible to kill the object before function
NOT auto p = make.unique<T>();
    //terminates

//but not possible to kill the object before function terminates
YES T obj;
    //it is stored on stack
```

CV 01.11.2017

07.11.2017

Tuples and arrays

(my slides 91) - avoid native T a[n] arrays - use `std::array<T,n> a;`

- elements of structs are default public, elems of class are default private, otherwise NO difference.

- structs cannot have generic types
- tuple can have generic tubles

```
std::tuple<T1,T2,T3> a;
std::get<0>(a) = /*....*/;
std::get<1>(a).foo();
```

- vector of different types > it is nessesary to have common interface Tbase

```
std::vector<std::unique_ptr<Tbase>> a;
a.push_back(std::make_unique<T1>());
```

Containers (vector, ..)

- always have their own data space, they cannot share it. also copying the
- data can cause errors.
- std::move solves that, plus it does no allocation, so it is fast
- it is not possible to return reference to ccontainer from func.

  solution:

```
vector<> f() {
    vector<> x;
    return x;
}
z = f();
// this creates x container, tmp container, z container. content of
// x is moved to tmp and that is moved to z variable.
```

- it is reasonable to take address of something, to which some value can be assigned (eg. l-value).

```
int x;
int * p = &x;

not p = &(b+c);
// slide 95
```

- **r-value** ... *something, which disappears after creating* ex. b+c

```
Complex(1,2); //created on stack - fast and compiler does cleanup
// type Complex
new Complex(1,2); //created on heap - slower and i have to clean it.
// type Complex * //pointer
```

- almost never write "new"
  - simple cases write it without it
  - complicated case write smart pointers
  - avoid implementing operator= and the other funcs
  - the rule of five

28.11.2017

slide 133 `X::y` when X is a type or a namespace type::m object.m pointer->m

- namespace can be reopened and things can be added to it. class cannot do that (because private vars could be accesible.)!
- class may be a template argument

```
std::complex<double> x;
1.0 + x; // in std operator+(double, std::complex);
1.0 std::+ x; //does not work, compiler finds operator+ on its own.

std::unique_ptr<T> p;
//possible compliler will find move because p is of type which is in std
move(p);

virtual ~Base() noexcept {}; //noexcept means the func wont throw

!   virtual functions work only if used with pointers
    std::unique_ptr<Base> p = std::make_unique<Derived>();
    p.f(); // calls Derived::f not Base::f.
    Base b = d;
    b.f(); //this will call Base::f.
```

29.11.2017 cv = vyhodnoceni domaciho ukolu

- `int_least32_t` pokud to pouzivam ze zadani, tak prejmenovat na něco jiného
- `argv[1] != "0" //je true porad, porovnani char * a char.*`

05.12.2017

inheritance

use ISA hiearchy - virtual inheritance (:virtual public BaseClass) - all inherited class are elsewhere and there are pointers to them. - we can compute their position by the type of the instatiated class. - the base class is represented only once, even if both parents inherit from it.

12.12.2017

STL - slide 183

containers

- cont cannot hold descendants of T, only T. (~ solved using pointers)
- T needs to be (copy|move)able
- sequential containers
  - iterators - begin(), end(), it
- key-value
  - ordered - implemented as trees
    - set, multiset, map, multimap
  - hashed
    - unordered_map, unordered_set, ...
- functors
- iterators
  - insert, remove in vector/basic_string/deque invalidate all associated iterators

13.12.2017 cv

ÚKOL 2 do 3.1.2018

- soubory - oddeleny tabulatorem
- cislo linky, cislo jizdy, cislo zastavky
- rychle najit odjezd ze zastavky po nejakem čase.
- data ve dvou kontejnerech
  - jeden - zastávka a odjezdy z ní
  - druhý - jeden trip nejake linky
  - containery (2) budou ve tride timetable

03.0#2018 cv

```
class riterator {
    bool operator==(const riterator & b) const{
        return ...;
    }
    riterator & operator++(){
        ...;
        return *this;
    };
    riterator & operator++(int){
```

```
        auto x = *this;
        ++*this;
        return x;
    };
    row_reference operator *() const {

    };
    const row_reference operator *() const {

    };
private:
    [const] Matrix<T> * m_;
    size_t i_;
};
```

09.01.2018

compilování a linkování

- inline funkce
  - se při kompilaci expandují přímo na místo jejich volání (old style) zároveň je linkeru řečeno aby ignoroval duplicity
- c++ nepoužívá generiku za runtimu
- exceptions
- catch musí brát referenci, aby mohl přijmout i zděděnou třídu
- vector používá pouze typ, který má noexcept konstructor, aby to nespadlo v polovině kopírování

10.01.2018

vyhodnocovavni vyrazu v infixu

Advanced C++ 21.02.2018

Exception handling

- `std::current_exception()` – returns pointer to currently handled exception
- constructor of move noexcept - means that the moving is done faster and it cannot fail
- if the constructor is not NOEXCEPT - it is not moving but COPYing! - slower

- noexcept can be conditional noexcept(const bool)
- stdexcept
  - class exception should not be used - it is abstract class
- exception-safe programming
  - std::lock_guard - destructor releases resources even in case of exception
    - `std::mutex mm;`
    - `std::lock_guard<std::mutex> lock_mutex(mm);`
  - weak exception safety, string exception safety

07.03.2018

Variadic templates

Perfect forwarding

- reference is imutable - once its created it cannot be changed => reference to a reference does not make sense.
  - lvalue reference is more important, therefore it is selected whenever there is ambiguity in rvalue/lvalue
  - *forwarding reference* also known as *universal reference*

```
template<typename T> void f(T &&p){
    g(std::forward<T>(p)); //"podmineny move" lvalue/rvalue
}
```

  - this code means, there is a list `TList` of types a list of values `plist` they are both the same size and there is an iteration on both of them at the same time.

```
template< typename ... TList>
iterator emplace( const_iterator p, TList && ... plist) {
    void * q = /* the space for the new element */;

    value_type * r = new( q) value_type( std::forward< TList>( plist) ..
/* ... */
}
```

14.03.2018

- `ftor<std::remove_reference_t<T>> make_ftor(T && p)`
- use forwarding reference T3 && instead of lvalue refernce T3 &

std::tuple template

- **traits**
  - templates not designed to be instatiated into objects
  - contains type defs, constats, static funcs
  - used as a compile-time function which assigns types/constants/run-time
- **policy class**
  - non-template (!) class
  - compile time equivalent of objects
  - contains types/constat/run-time funcs
  - passed as a template argument to function templates
- **functor**
  - class containing non-static func named operator()
  - usually passed as run-time arg to func templates
  - acts as a function, cerated by packaging a function body together with some data referenced in the body
- **tag classes**
  - empty classes ... (it is in slides)
- `decltype(v)` - this converts v to the TYPE of v
- `declval<T>()` - for the given type T it returns SOME value of the type.
- C++ sort is faster than C sort since in C there are no functors and templates

21.03.2018

- implementation of tuple - using recursion (slide ~ 60)
  - explicit specialization `template<> class tuple<> {};`
  - `import <iostream>`
- iteration over tuple
  - `std::get<I>(t);` - `I` has to be constant because it returns different type and types have to be known at compile time.
  -

28.03.2018

Cache

- **cache line**
  - data is being transfered between memory and cache in *cache lines* = 64B

- **cache hit** – requested data is in cache ~ 97%, **cache miss** – otherwise
- **cache line (load|flush)**

Parallelism

- race condition
  - writing in the same space does every cpu serially.

C++ 11 atomic ops

- `<atomics>`
- lock free
- **CAS - compare and switch**

Homework 1

split

- `>>` ma nalevo akceptovat jakehokoliv potomka istreamu
- pokud je argument l-value T, tak je výstupní
- cteci argumenty jsou **char**
- dve l-value nesmi byt vedle sebe
- metoda split nebude vracet primo onen rozdeleny string (treba ve forme `tuple`), ale neco, na cem bude definovan op `>>`
- cist, dokud nenajdu delimiter, pokud na nej nenarazim tak pouzit `\n`, pokud ten tam neni tak cist az k EOFu
- `>>` oznamuje chuby nejakym flagem na stringu. Propagovat ho dale, ale vyjimkou
- `split` pouze sbali argumenty do nejake tridy.

```
template <typename TL>
... split(TL && ... pl) //univerzalni, linking reference
// ... = tuple<TL ...> neco podedeneho. tohle se vetsinou nepouziva
{
// v TL budou dva druhy
}
```

- v TL budou dva druhy typu
  - `T &` - lvalue T
  - `char` - rvalue char
- `make_tuple` stripuje reference
- postupovat rekurzi, ale nerozbalovat `pack<TL...> & p`.
- lze `template <size_t i, __ TL> void doint( ___ is, pack<TL ...> & p)`
  - pujde spaten udelat rozliseni pripadu `char` | `TL`
- lze (a je to **lepsi**!)

- lze to parcialne specializovat

```
template <__ TL1>
struct traits( ___ is, pack<TL ...> & p){
    template<__ TL2>
    static void doit( ___ is, pack<TL2...> & p){
        //...
    }
}

//druhy parametr ma byt "nejaka" reference
template<typename ... TL>
istream & operator>>(istream & is, pack<TL ...> & p){

    return is;
}
```

questions

- Jak řešit,že string je lvalue reference, ikdyž je předán přímo jako argument?

04.04.2018

memory model

- speculative execution
  - load instructions are always executed (unless they are after) goto
  - delayed write
- REL, ACQ - release, acquire
- `void atomic_thread_fence(...)`
- lock-free programming, lock-free data structures
  - performed using only atomic instructions
  - eg. read, compare, write is performed atomically
  - use **compare and switch**
  - **ABA** problem - thrid part loads the structure, changes is (B) and then it puts the A back.
  - solved using counter
- locks take many instructions ~ 100
- **threads**
  - namespace `<thread>`

10.04.2018

- `std::condition_variable`
- keyword `thread_local`
  - function is private for one thread. all functions of the thread use the same copy of the thread_local func.
- `reduce` does a commutative operation on its args. eg. sum two vectors

DU

- ~ 16 vlaken

17.04.2018 C++ 14/17/20

- range-base loop for - C++ 17 structured bindings

##Structured return values

C++ 17 structured return ~~~{.cpp} auto f(....) { .... return {a,b,c}; } auto [x,y,z] = f(); auto [a,b,c] = { 1, 2, 3};

for(auto&& [key,value]:mymap)

~~~~

##Type inference, deduction
~~~~