Zadanie 1.

#include "conv.h"

int main()

{

   Derived1 d1;

   Derived3 d3;

   Derived2 d2 = d3;

   // Derived1 d1\_make\_err = d2;

   // std::vector<Base> v\_make\_err = {d1, d2, d3};

   std::vector<Base\*> v = {&d1, &d2, &d3};

   std::cout << \*v.front()<<std::endl;

   std::cout << v;

}

Wynik:

virtual std::ostream& Derived1::print(std::ostream&) const

[

virtual std::ostream& Derived1::print(std::ostream&) const,

virtual std::ostream& Derived2::print(std::ostream&) const,

virtual std::ostream& Derived3::print(std::ostream&) const

]

Rozwiazanie przykładowe 1:

#include <vector>

#include <iostream>

class Base

{

public:

   virtual std::ostream& print(std::ostream&) const = 0;

};

class Derived3;

class Derived1 : public Base

{

public:

   std::ostream& print(std::ostream& out) const

   {

   return out << "Derived1\n";

   }

};

class Derived2 : public Base

{

public:

   Derived2(const Derived3&) { }

   std::ostream& print(std::ostream& out) const

   {

   return out << "Derived2\n";

   }

};

class Derived3 : public Base

{

public:

   std::ostream& print(std::ostream& out) const

   {

   return out << "Derived3\n";

   }

};

std::ostream& operator<< (std::ostream& out, const Base& base)

{

   return base.print(out);

}

std::ostream& operator << (std::ostream& out, std::vector<Base\*> v)

{

   out << "[\n";

   for(unsigned i=0; i<v.size(); ++i)

   out << \*v[i];

   out << "]\n";

   return out;

}

Zadanie 2.

#include "fifo.h"

int main()

{

   fifo<> ft;

   for(int i: {0,1,2,3,4})

   ft.push(i);

   fifo<int, std::deque> f = ft;

   std::cout << f.get();

   std::cout << f.size();

   for(fifo<>::storage\_type::const\_iterator i = f.begin(); i != f.end(); ++i)

   std::cout << \*i << ",";

}

Wynik:

fifo<T, StorageType>::fifo() [with T = int, StorageType = std::deque]

0

4

1,2,3,4,

Rozwiązanie przykładowe:

#ifndef FIFO\_H

#define FIFO\_H

#include <deque>

#include <typeinfo>

#include <iostream>

template<class T = int, template <class P = T, class Alloc = std::allocator<T>>

class U = std::deque > //template ma przyjmować, typ (T) i kontener (U<T, Alloc>). wiemy z treści ze kontenerem jest deque, więc odtwarzamy jego argumenty szablonowe czyli P - typ i allocator, który też jest template’em dla typu P ktory ma byc T. Dopiero jak argumenty szablonowe kontenera pokryja sie z deklaracja deque można go przypisać jako parametr domyślny, inaczej rzuci jakimś invalid use of std::deque.

//wydaje mi sie ze powinno byc class Alloc = std::allocator<P> a nie od T

class fifo

{

public:

   void push(T & in) {m\_Container.push\_back(in);} //wrzuca kolejny elemnet na koniec

   fifo() //glośny konaltruktor

   {

       std::cout<<typeid(\*this).name()<<std::endl;

   }

   T get() //zwraca pierwszy element (first in -> first out)

   {

       T tmp = m\_Container.front(); //przypisuje wartosc

       m\_Container.pop\_front(); //usuwa pierwszy element

       return tmp; //zwraca wartosc usunietego elementu

   }

   typedef U<T> storage\_type; //tworzy typ strage\_type

   typename storage\_type::const\_iterator begin() //ktos cos? po co ten typename

   { //w sensie czemu bez niego nie kompiluje?

//poniewaz kompilator nie wie czy sie odwolujemy do zmiennej statycznej czy typu, a typeamem mu mowimy ze odwolujemy sie do typu

       std::cout<<std::endl;

       return m\_Container.begin();

   }

   typename storage\_type::const\_iterator end(){return m\_Container.end();}

   int size()

   {

       std::cout<<std::endl;

       return m\_Container.size();

   }

private:

   U <T> m\_Container;

};

#endif /\* FIFO\_H \*/

Przykładowe rozwiązania z wykorzystaniem \_\_PRETTY\_FUNCTION\_\_

#ifndef CONV\_H

#define CONV\_H

#include <iostream>

#include <vector>

class Base

{

public:

virtual std::ostream& print(std::ostream& o) const = 0;

};

class Derived1: public Base

{

public:

virtual std::ostream& print(std::ostream& o) const

{

return o << \_\_PRETTY\_FUNCTION\_\_;

}

};

class Derived2: public Base

{

public:

virtual std::ostream& print(std::ostream& o) const

{

return o << \_\_PRETTY\_FUNCTION\_\_;

}

};

class Derived3: public Derived2

{

public:

virtual std::ostream& print(std::ostream& o) const

{

return o << \_\_PRETTY\_FUNCTION\_\_;

}

};

std::ostream& operator <<(std::ostream& o, const Base& base)

{

return base.print(o);

}

std::ostream& operator <<(std::ostream& o, const std::vector<Base\*>& base)

{

o << "[\n";

for(unsigned i = 0; i < base.size(); ++i)

{

o << \*(base[i]) << ",\n";

}

o << "]\n";

return o;

}

#endif // CONV\_H

#ifndef FIFO\_H

#define FIFO\_H

#include <iostream>

#include <deque>

template<class T = int, template <class U = T, class Allocator = std::allocator<T> > class StorageType = std::deque >

class fifo

{

public:

using storage\_type = StorageType<T>;

fifo()

{

std::cout<< \_\_PRETTY\_FUNCTION\_\_ << std::endl;

}

   void push(const T& a)

   {

    container.push\_back(a);

   }

   T get()

   {

       T temp = container.front();

       container.pop\_front();

       return temp;

   }

   unsigned size() const

   {

       return container.size();

   }

   typename storage\_type::const\_iterator begin()

   {

       return container.begin();

   }

   typename storage\_type::const\_iterator end()

   {

    return container.end();

   }

private:

   StorageType<T> container;

};

#endif // FIFO\_H

----------------------------

// podsiadlo - wlasciwie to przerobiony przyklad z zadan z wykladu (zadanie numer 09.22)

// roznica byla tylko z lifo na fifo i dodatkowy typedef + iteratory (schemat sam w sobie podobny)

//plik fifo.h

#pragma once // alternatywnie include guardsy

#include <deque>

#include <iostream>

#include <memory> // std::allocator

template<class T = int,

    template <typename ElemType, typename AllocType> class StorageType = std::deque> // domyslne parametry takie a nie inne bo inaczej nie powiedzie sie uzycie konstruktora kopiujacego w main (inne parametry == niezgodnosc typow)

//patrz podrozdzial template template parameters w <https://en.cppreference.com/w/cpp/language/template_parameters>

class fifo {

   StorageType<T, std::allocator<T> > \_cont; //dopiero od c++ 11 można >> zamiast > > xD

public:

   //using storage\_type = decltype(\_cont); // c++11

   typedef StorageType<T, std::allocator<T> > storage\_type;

   fifo() {

   std::cout << \_\_PRETTY\_FUNCTION\_\_ << std::endl; //tylko gcc, mozna uzyc typeid z naglowka typeinfo ale wyglada jak szit

   }

   unsigned size() const {

   return \_cont.size();

   }

   T get() {

   T tmp = \_cont.front();

   \_cont.pop\_front();

   return tmp;

   }

   typename storage\_type::const\_iterator begin() const { // kiedy uzywac nalezy slowka typename patrz w <https://stackoverflow.com/questions/7923369/when-is-the-typename-keyword-necessary>

   return \_cont.begin();

   }

   typename storage\_type::const\_iterator end() const {

   return \_cont.end();

   }

   void push(const T val) {

   \_cont.push\_back(val);

   }

};

// plik main.cpp

#include “fifo.h”

namespace std { class queue{}; }

int main()

{

fifo<> ft;

for(int i: {0,1,2,3,4,5,6,7,8,9}) // c++11, zakresowy for pracujacy na kopiach

    ft.push(i);

fifo<int, std::deque> f = ft;

std::cout << f.get() << "\n";

std::cout << f.size() << "\n";

for(fifo<>::storage\_type::const\_iterator i = f.begin(); i != f.end(); ++i) //tu juz nie trzreba typename, patrz link wyzej

    std::cout << \*i << ",";

return 0; //mindur w ogole uzywa return 0; :O?

}

// moje do pierwszego, plik conv.h

#pragma once

#include <vector>

#include <iostream>

class Base {

public:

   virtual ~Base() {} // dobre zwalnianie pamiecy przy polimorfizmie

   virtual std::ostream & print(std::ostream & ostrm) const = 0; //abstrakcyjny interfejs, niemozliwosc instancjonowania

};

class Derived1 : public Base

{

public:

   virtual std::ostream & print(std::ostream & ostrm) const {

   return ostrm << "virtual std::ostream& Derived1::print(std::ostream&) const";

   } //alternatywnie podstawic \_\_PRETTY\_FUNCTION\_\_

};

class Derived2 : public Base

{

public:

   virtual std::ostream & print(std::ostream & ostrm) const {

   return ostrm << "virtual std::ostream& Derived2::print(std::ostream&) const";

   }

};

class Derived3 : public Derived2

{

public:

   virtual std::ostream & print(std::ostream & ostrm) const {

   return ostrm << "virtual std::ostream& Derived3::print(std::ostream&) const";

   }

};

std::ostream & operator<<(std::ostream & ostrm, const Base & b) {

   return b.print(ostrm);

}

std::ostream & operator<<(std::ostream & ostrm, const std::vector<Base \*> v) {

   ostrm << "[" << std::endl;

   for (auto base\_ptr : v) // typ: Base \* (kopia)

   base\_ptr->print(ostrm) << "," << std::endl;

   return ostrm << "]";

}

//main.cpp

#include "conv.h"

int main() {

   Derived1 d1;

   Derived3 d3;

   Derived2 d2 = d3;

   //Derived1 d\_make\_eror = d2;

// std::vector<Base> v\_make\_err = {d1, d2, d3};

std::vector<Base\*> v = {&d1, &d2, &d3};

   std::cout << \*v.front() << std::endl;

   std::cout << v;

   std::cout << std::endl;

   std::vector<Base \*> b = {&d1, &d2, &d3};

}

// z takim drzewkiem dziedziczenia nie trzeba pisac zadnych konstruktorow

**CPP II TERMIN, 6 LIPCA 2018r.**

**ZAD1 (5 pkt)**

**--------------**

#include <type\_traits> // Dla std::is\_pod i std::integral\_constant

#include “copy.h”

struct A {};

struct A\_ { virtual ~A\_() {} };

int main()  {

//prosze wykorzystac te informacje

Std::integral\_constant<bool,true> tA = std::is\_pod<A>::type();

std::integral\_constant<bool,false> tA\_ = std::is\_pod<A\_>::type();

//Iterator dla każdego kontenera ma zdefiniowany

//typ value\_type określający typ obiektu na który wskazuje

std::deque<A> vA1;

std::vector<A> vA2;

my\_copy(vA1.begin(), vA1.end(), vA2.begin() ) ;

std::vector <A\_> vA\_1;

std::deque<A\_> vA\_2;

my\_copy(vA\_1.begin(), vA\_1.end(), vA\_2.begin() ) ;

}

/\* output/wyjscie

Copying POD objects

Copying non-POD objects

\*/

**ZAD2 (5 pkt)**

**----------**

namespace std {class auto\_ptr{}; class unique\_ptr{}; class shared\_ptr{} ; }

include “ptr.h”

struct A { int i = 1; };

int main() {

const ptr<A>  a(new A);

const ptr<A> b;

ptr<A> c(new A);

//ptr<A>  \_b\_error = new A;      // Odkomentowanie powoduje błąd kompilacji

//a = a;  // Odkomentowanie powoduje błąd kompilacji

//const ptr<A> \_c\_error = a;     // Odkomentowanie powoduje błąd kompilacji

std::cout << ((\*a).i, a->i)        << “ “ << ++c->i << “ “;

std::cout << (a == ptr<A>() ) << “ “ << (a != b) << ‘“\n”;

}

/\* output/wyjście:

1 2 0 1

\*/

**Ad.1**

// Sorry, że się wtrącam, ale poniższe rozwiązanie imo jest kompletnie błędne, działa jedynie z A i A\_ a nie taki był cel tego zadania. my\_copy ma dostać dowolny iterator i zawołać odpowiednią implementację w zależności czy iterator jest na typ POD lub nie. Proponuję:

// copy.h

#include <iostream>

#include <vector>

#include <deque>

template

<

bool boolean

>

struct ForPOD {};

template

<>

struct ForPOD<true>{

 template

 <

 typename IterI,

 typename IterO

 >

 static void copy(IterI start, IterI end, IterO startO){

   std::cout << "POD!" << std::endl;

 }

};

template

<>

struct ForPOD<false>{

 template

 <

 typename IterI,

 typename IterO

 >

 static void copy(IterI start, IterI end, IterO startO){

   std::cout << "Non POD!" << std::endl;

 }

};

template

<typename IterI, typename IterO>

void my\_copy(IterI start, IterI end, IterO startO){

 ForPOD<std::is\_pod<typename IterI::value\_type>::value>::copy(start, end, startO);

}

-----------------------------------------------------------------------------------------------------------------------

Jedna z opcji

template <typename T>

struct Is\_Pod{};

template<> //specjalizacja dla typu A

struct Is\_Pod<A>{

static void print(){std::cout << "Copying POD objects" << std::endl;}

};

template<> //specjalizacja dla typu A\_

struct Is\_Pod<A\_>{

static void print(){std::cout << "Copying non-POD objects" << std::endl;}

};

template <typename InputIterator, typename DestIterator>

void my\_copy(InputIterator kopać, InputIterator prowadzącego, DestIterator w\_czoło){

Is\_Pod<typename InputIterator::value\_type>::print();

}

/////////////////////////////

Zadanie 2. Plik “ptr.h”:

namespace std {class auto\_ptr{}; class unique\_ptr{}; class shared\_ptr{} ; }

include “ptr.h”

struct A { int i = 1; };

int main() {

const ptr<A>  a(new A);

const ptr<A> b;

ptr<A> c(new A);

//ptr<A>  \_b\_error = new A;      // Odkomentowanie powoduje błąd kompilacji

//a = a;  // Odkomentowanie powoduje błąd kompilacji

//const ptr<A> \_c\_error = a;     // Odkomentowanie powoduje błąd kompilacji

std::cout << ((\*a).i, a->i)        << “ “ << ++c->i << “ “;

std::cout << (a == ptr<A>() ) << “ “ << (a != b) << ‘“\n”;

}

/\* output/wyjście:

1 2 0 1

\*/

ZAD2.

template <typename T>

class ptr{

public:

ptr(){}

ptr(T \* a): obj(a){ }

T operator\* () const{ return \*obj;}

T\* operator->() const{ return obj;}

bool operator==(const ptr<T> & a) const{

return false;

}

bool operator!=(const ptr<T> & a) const{

return true;

}

private:

T \* obj;

void operator= (const  ptr<T> other)const{}

ptr(const ptr<T>&){}

};