#include <iostream>

#include <random>

**class** Task{

**protected**:

std::string name{};

**int** runtime{};

**public**:

**explicit** Task(std::string &name) {**this**->name=name;}

**virtual bool** execute()=0;

};

**class** SingleTask:**public** Task{

**public**:

**explicit** SingleTask(std::string name):Task(name) {}

**bool** execute() override

{

runtime=rand()%60;

**if**(runtime % 2 == 0){

std::cout<<name<<std::endl;

std::cout<<"\t Tipo: "<<"Simple"<<std::endl;

std::cout<<"\t Tiempo de ejecucion: "<<runtime<<std::endl;

std::cout<<"\t Resultado de ejecucuion: "<<"exitosa\n"<<std::endl;

**return true**;

}**else**{

std::cout<<name<<std::endl;

std::cout<<"\t Tipo: "<<"Simple"<<std::endl;

std::cout<<"\t Tiempo de ejecucion: "<<runtime<<std::endl;

std::cout<<"\t Resultado de ejecucuion: "<<"fallida\n"<<std::endl;

**return false**;

}

}

};

**class** FaultToleranceTask:**public** Task{

**private**:

**int** maxAttempt;

**public**:

FaultToleranceTask(std::string name, **int** maxAttempt) : Task(name) {**this**->maxAttempt=maxAttempt;}

**bool** execute() override

{

std::cout<<name<<std::endl;

std::cout<<"\t Tipo: "<<"Tolerante a fallos"<<std::endl;

std::cout<<"\t Maximo de intentos: "<<maxAttempt<<std::endl;

**for** (**int** i = 0; i < maxAttempt; ++i) {

runtime=rand()%61+60;

**if**(runtime % 2 == 0){

std::cout<<"\t Intento numero: "<<i+1<<std::endl;

std::cout<<"\t Tiempo de ejecucion: "<<runtime<<std::endl;

std::cout<<"\t Resultado de ejecucuion: "<<"fallida"<<std::endl;

}

**if**(runtime % 2 != 0){

std::cout<<"\t Intento numero: "<<i+1<<std::endl;

std::cout<<"\t Tiempo de ejecucion: "<<runtime<<std::endl;

std::cout<<"\t Resultado de ejecucion: "<<"exitosa"<<std::endl;

**break**;

}

}

}

};

**struct** Node{

Node\* next;

Task\* task;

Node(Task\* task, Node\* next)

{

**this**->task=task;

**this**->next=next;

}

};

**class** Queue{

**private**:

Node\* first;

Node\* last;

**public**:

Queue() = **default**;;

**void** enqueue(Task\* task)

{

**if**(last == **nullptr**)

{

Node\* p = **new** Node(task , **nullptr**);

first = p;

last = p;

}

**else**{

Node\* p = **new** Node(task , **nullptr**);

last -> next = p;

last = p;

}

}

Task\* dequeue()

{

**if**(first!= **nullptr**)

{

first=first->next;

**return** first->task;

}

}

Task\* Queue::dequeue(){

Task\* output;

Node\* borrar;

**if**(first != **nullptr**){

borrar = first;

output = first -> task;

first = first -> next;

**delete** borrar;

}

**else**{

**return nullptr**;

}

**return** output;

}

**bool** isEmpty()

{

**return** first == **nullptr**;

}

~Queue() = **default**;

};

**int** main() {

srand(**static\_cast**<**unsigned int**>(time(**nullptr**))); // srand(time(0));

**auto** hello = **new** Queue(); //Queue\* hello = new Queue();

Task\* ft1 = **new** FaultToleranceTask("ols" , 10);

Task\* ft2 = **new** FaultToleranceTask("oli" , 8);

Task\* ft3 = **new** FaultToleranceTask("uwu" , 6);

Task\* ft4 = **new** FaultToleranceTask("dis" , 15);

Task\* ft5 = **new** FaultToleranceTask("per" , 8);

Task\* st1 = **new** SingleTask("aja");

Task\* st2 = **new** SingleTask("dsl");

Task\* st3 = **new** SingleTask("acm");

Task\* st4 = **new** SingleTask("cpc");

Task\* st5 = **new** SingleTask("her");

hello->enqueue(ft1);

hello->enqueue(st1);

hello->enqueue(ft2);

hello->enqueue(st2);

hello->enqueue(ft3);

hello->enqueue(st3);

hello->enqueue(ft4);

hello->enqueue(st4);

hello->enqueue(ft5);

hello->enqueue(st5);

**for**(**int** i = 0 ; i < 10 ; i++){

hello->dequeue()->execute();

}

**delete** hello;

**return** 0;

}