Importante: Para item abaixo deve ser copiado trechos do código que cumprem o requisito e explicado, se não for aparente, o porquê o requisito é cumprido. Sejam bem explícitos. Deve ser indicado também o .h ou .cpp no qual o trecho do código está. Eu avaliarei o código do Github a partir desse documento para confirmá-lo e também para detectar possíveis erros. **Quem não seguir o que está indicado aqui, não terá o projeto avaliado e perderá a atividade.**

Requisitos funcionais

Polimorfismo

1. Duas classes abstratas, sendo que uma classe abstrata herda da outra classe abstrata na hierarquia de classes;

   Classes: *Utensílio*(Preferi usar essa ao invés de usar item, pois seria muito genérico, e eu teria que fazer bloco herdar de item, o que iria me gerar algum problemas, logo preferi simplificar), *Ferramenta* que herda de *Utensílio* (ambas são puramente virtuais).
2. Cada classe abstrata deve ter pelo menos um parâmetro, o construtor vazio e o construtor default. Deve ter também um método não virtual, que não pode ser set ou get;

   Parâmetros de Utensílio:

```
string nome;
const int maxRes = 0;
float atualRes;
const int tipoN = 0;
const string tipoS = "";
```

   Construtores de Utensílio:

```
Utensilio::Utensilio(){
    this->atualRes = this->maxRes;
}
Utensilio::Utensilio(int maxRes, float atualRes) {
    const_cast<int&> (this->maxRes) = maxRes;
    this->atualRes = maxRes;
}
```

   Os parâmetros tipoN e tipoS, nao podem ser setados nessa classe, logo são setados nas classes não abstratas.

   Parametros de Ferramenta:

```
const int tipoFerramentaN = 0;
const string tipoFerramentaS = "";
void setTipoFerramentaS(int);
Data * dataDeCriacao;
```

Construtores de Ferramenta:

```
Ferramenta::Ferramenta() {
   const_cast<int&> (this->tipoFerramentaN) = TIPO_PICARETA;
   this->setTipoFerramentaS(TIPO_PICARETA);
   this->dataDeCriacao = new Data();
}

Ferramenta::Ferramenta(int tipoFerramenta, Data * dataDeCriacao) {
   const_cast<int&> (this->tipoFerramentaN) = tipoFerramenta;
   this->setTipoFerramentaS(tipoFerramenta);
   this->dataDeCriacao = new Data(*dataDeCriacao); // criado usando um
construtor cópia que foi gerado pelo compilador.
}
```

3. Pelo menos três classes concretas na hierarquia de classes;

   Classes Picareta, Pa e Machado.

4. Usar coerção de tipo C++ e não C, ou seja usar o **static_cast** para fazer o máximo de reutilização de código.

   Utilizado na função de sobrecarga do operador << em todas as classes base:
   Picareta.cpp:

```
ostream &operator<<(ostream &output, const Picareta &picareta) {
   output << static_cast<const Ferramenta&> (picareta);
   output << " de " << picareta.getTipoS();
   return output;
}
```

   Pa.cpp:

```
ostream &operator<<(ostream &output, const Pa &pa) {
   output << static_cast<const Ferramenta&> (pa);
   output << " de " << pa.getTipoS();
   return output;
}
```

   Machado.cpp:

```
ostream &operator<<(ostream &output, const Machado &machado) {
   output << static_cast<const Ferramenta&> (machado);
   output << " de " << machado.getTipoS();
   return output;
}
```

5. Usar um vector de classes concretas, o **dynamic_cast** e o **typeid** como indicado aqui: https://basecamp.com/2595605/projects/7018448/messages/33080741

   Localizado no main.cpp

```
vector<Ferramenta*> inventario;

  inventario.push_back(picareta1);
  inventario.push_back(picareta2);
  inventario.push_back(picareta3);
```

```
inventario.push_back(pa1);
inventario.push_back(pa2);
inventario.push_back(pa3);
inventario.push_back(machado1);
inventario.push_back(machado2);
inventario.push_back(machado3);
```

(Linhas 147 ~ 154)

```
for (int i = 0; i < inventario.size(); i++) {
        Picareta* derivedPic = dynamic_cast<Picareta*> (inventario[i]);
        if (derivedPic != 0) cout << i + 1 << " - " << *derivedPic << endl;
        Pa* derivedPa = dynamic_cast<Pa*> (inventario[i]);
        if (derivedPa != 0) cout << i + 1 << " - " << *derivedPa << endl;
        Machado* derivedMach = dynamic_cast<Machado*> (inventario[i]);
        if (derivedMach != 0) cout << i + 1 << " - " << *derivedMach << endl;
    }
```

(Linhas 74 ~ 83)

```
    cout << "Voce tem uma ";

    Picareta* derivedPic = dynamic_cast<Picareta*> (inventario[itemAtivo]);
    if (derivedPic != 0) cout << *derivedPic;
    Pa* derivedPa = dynamic_cast<Pa*> (inventario[itemAtivo]);
    if (derivedPa != 0) cout << *derivedPa;
    Machado* derivedMach = dynamic_cast<Machado*> (inventario[itemAtivo]);
    if (derivedMach != 0) cout << *derivedMach;

    cout << " na sua mao" << endl;
```

6. Criar uma função no arquivo do **main**, que aceita um ponteiro da classe genérica e mostrar o seu uso para as classes concretas;

Importante: **Todos os Requisitos abaixo ainda devem ser feitos e serão avaliados**

Requisitos funcionais

Todos os atributos e funções membros devem estar relacionados a classe

1. Pelo menos 4 atributos

Em Picareta.h

```
    int tipoN;
    string tipoS;
```

Bloco matMine;

Em Ferramenta.h

int tipoFerramenta;

2. Pelo menos 4 funções membros sem incluir get e set

Em Ferramenta.h

```
virtual void jogarNoChao();
virtual void checarEstado() const;
virtual inline void destruir();
virtual bool quebrarBloco(Bloco &);
```

Requisitos de implementação

3. Todos os atributos devem ser inicializados. Fez validação de dados

Atributos da classe Bloco:

```
string nome;
vector<float> resMat;
vector<bool> colherMat;
```

Inicialização dos atributos no construtor:

```
Bloco::Bloco(string nome, float resMat[5], bool colherMat[5]) {
    this->setNome(nome);
    this->setResMat(resMat);
    this->setColherMat(colherMat);
}
```

Atributos da classe Ferramenta:

```
int tipoFerramenta;
int maxRes;
int atualRes;
```

Inicialização dos atributos no construtor:

```
Ferramenta::Ferramenta(int tipoFerramenta, int maxRes, int atualRes, Data &
dataDeCriacao) {
    this->tipoFerramenta = tipoFerramenta;
    this->setMaxRes(maxRes);
    this->setAtualRes(atualRes);
    this->dataDeCriacao = new Data(dataDeCriacao);
}
```

Atributos da classe Picareta:

```
int tipoN;
string tipoS;
Bloco matMine;
static int numPicaretasQuebradas;
```

4. Três construtores, incluindo um construtor de cópia e construtor com parâmetros defaults. Verifica alocação dentro do construtor de cópia.

Três construtores para Picareta:

```
Picareta(string = "madeira", int = 0, int = 60, int = 60, Bloco = Bloco());
Picareta(const Picareta &);
```

```
Picareta(int);
```

5. Deve ter um atributo string

Atributo string em Picareta.h:

```
string tipoS;
```

6. Um atributo static. Correta modelagem dos statics?

Inicialização do atributo static(Na classe Picareta):

```
int Picareta::numPicaretasQuebradas = 0;
```

Utilização do atributo static:

```
virtual inline void destruir(){
        numPicaretasQuebradas++;
        cout << "Sua Picareta de " << this->tipoS << " quebrou." << endl;
        cout << "Construa uma picareta Nova" << endl;
}
```

7. Um atributo const static

Atributo const static em Ferramenta.h:

```
const static int TIPO_PICARETA = 0;
```

8. Dois métodos constantes (não pode ser get)

Na classe Picareta:

```
void Picareta::checarEstado() const {
        cout << "Sua picareta de " << this->tipoS << " esta com " << (this->atualRes /
this->maxRes) * 100 << "% de resistencia ";
        cout << "(" << this->atualRes << " de " << this->maxRes << ")" << endl;
}
```

Na classe Ferramenta:

```
void Ferramenta::infoItem() const{
        cout << "Este Item e uma ferramenta:" << endl;
}
```

E sobrecarregado na classe Picareta:

```
void Picareta::infoItem() const{
        Ferramenta::infoItem();
        cout << "Picareta de " << this->tipoS << endl;
}
```

9. Um array

10. Uma função inline (não pode ser get ou set)

Método inline em Picareta.h:

```
virtual inline void destruir() {
        numPicaretasQuebradas++;
        cout << "Sua Picareta de " << this->tipoS << " quebrou." << endl;
        cout << "Construa uma picareta Nova" << endl;
}
```

11. Método com passagem por referência usando ponteiro

Método da classe Bloco:

```
bool Bloco::quebrarBloco(const Ferramenta * f) {
	int tipo = f->getTipoN();
	cout << resMat[tipo] << "sec para quebrar o Bloco" << endl << "Para cancelar
aperte C" << endl;
	int oldTime;
	for(int i = 0; i < resMat[tipo] * 1000; i++)
	{
		Sleep(0.1);
		if(kbhit())
			if(getch() == 'c')
			{
				cout << "Bloco nao foi quebrado" << endl;
				return false;
			}
		int time = i / 1000;
		if(time != oldTime) cout << time + 1 << endl;
		oldTime = time;
	}
	cout << "Bloco quebrado" << endl;
	return this->colherMat[tipo];
}
```

12. Método static – deve ser chamado no main

Método static em Picareta:

```
int Picareta::menuPicareta(){
	int opcao = -1;
	cout << "Menu de criação de Picareta" << endl;
	cout << "Voce deseja criar uma picareta de qual material?" << endl;
	cout << "1 - Madeira" << endl;
	cout << "2 - Pedra" << endl;
	cout << "3 - Ferro" << endl;
	cout << "4 - Diamante" << endl;
	cout << "5 - Ouro" << endl;
	cin >> opcao;
	while(opcao > 5 || opcao < 1) {
		cout << "Por favor, escolha uma opcao valida" << endl;
		cout << "Menu de criação de Picareta" << endl;
		cout << "Voce deseja criar uma picareta de qual material?" << endl;
		cout << "1 - Madeira" << endl;
		cout << "2 - Pedra" << endl;
		cout << "3 - Ferro" << endl;
		cout << "4 - Diamante" << endl;
		cout << "5 - Ouro" << endl;
		cin >> opcao;
	}
	return opcao - 1;
}
```

13. Composição com a classe Data. Fez uso do objeto criado?

Criado um ponteiro pra classe Data na classe Ferramenta que utiliza a data para registrar o dia da criação da ferramenta:

```
Ferramenta::Ferramenta(int tipoFerramenta, int maxRes, int atualRes, Data &
dataDeCriacao) {
        this->tipoFerramenta = tipoFerramenta;
        this->setMaxRes(maxRes);
        this->setAtualRes(atualRes);
        this->dataDeCriacao = new Data(dataDeCriacao);
}
```

E utilizado no método infoItem(), também na classe Ferramenta:

```
void Ferramenta::infoItem() const {
        cout << "Ferramenta criada no dia: ";
        dataDeCriacao->print();
        cout << "Este Item e ";
}
```

14. O que é const deve ser const.

15. Alocação dinâmica de memória. A memória é desalocada?

Atributos:

```
Data * dataDeCriacao; (classe Ferramenta)
Bloco * matMine; (classe Picareta)
```

Ambos destruídos nos destrutores de suas respectivas classes.

16. friend Operator<<

Na classe Bloco:

```
ostream &operator<< (ostream &output, const Bloco &bloco){
        output << bloco.nome;
        return output;
}
```

Na classe Ferramenta:

```
ostream &operator<< (ostream &output, const Ferramenta &ferramenta){
        output << ferramenta.getTipoDaFerramenta();
        return output;
}
```

Na classe Picareta:

```
ostream &operator<< (ostream &output, const Picareta &picareta){
        output << static_cast<Ferramenta>(picareta);
        output << " de " << picareta.getTipoS();
        return output;
}
```

17. Operator=

Na classe Bloco:

```
Bloco Bloco::operator= (Bloco b) {
        Bloco bloco;
        bloco.nome = b.nome;
```

```
            bloco.resMat = b.resMat;
            bloco.colherMat = b.colherMat;
            return bloco;
        }
```

Na classe Ferramenta:

```
        Ferramenta Ferramenta::operator =(Ferramenta fer){
            Ferramenta ferramenta(fer.tipoFerramentaN, fer.maxRes,
            fer.atualRes, *fer.dataDeCriacao);
            return ferramenta;
        }
```

Na classe Picareta:

```
        Picareta Picareta::operator =(Picareta pic){
            Picareta picareta(pic);
            cout << "Dentro de picareta" << endl;
            return picareta;
        }
```

18. vector push_back

Na classe Bloco:

```
        void Bloco::setResMat(float resMat[5]) {
            for(int i = 0; i < 5; i ++){
                this->resMat.push_back(resMat[i]);
            }
        }
```

Requisitos para as classes adicionais (pelo menos duas)

- Operator =
- Alocação dinâmica - se houver vazamento de memória a classe toda é desconsiderada
- Usar o destrutor
- Construtor de cópia
- Operator << friend
- Um const static

Requisitos herança

- Diagrama de classes (obrigatório salvar também o png do diagrama no gitHub)
  - Herança pública

- Construtor de cópia, e sobrecargas dos operadores de atribuição (=) e << (cout << base) para a classe base e derivada

Operator << em Ferramenta:

```
ostream &operator<< (ostream &output, const Ferramenta
&ferramenta){
        output << ferramenta.getTipoDaFerramenta();
        return output;
}
```

Operator << em Picareta:

```
ostream &operator<< (ostream &output, const Picareta &picareta){
        output << static_cast<Ferramenta>(picareta);
        output << " de " << picareta.getTipoS();
        return output;
}
```

Operator = em Ferramenta:

```
Ferramenta Ferramenta::operator =(Ferramenta fer){
        Ferramenta ferramenta(fer.tipoFerramentaN, fer.maxRes,
        fer.atualRes, *fer.dataDeCriacao);
        return ferramenta;
}
```

Operator = em Picareta:

```
Picareta Picareta::operator =(Picareta pic){
        Picareta picareta(pic);
        cout << "Dentro de picareta" << endl;
        return picareta;
}
```

- Usar Protected acessando diretamente os atributos na classe derivada

Na classe Ferramenta:

```
const int maxRes = 0;
float atualRes;
```

Na classe Picareta eles são acessados nos métodos: checarEstado e quebrarBloco.

- Alocação dinâmica de memória na classe base e derivada

Na classe Ferramenta: Data * dataDeCriacao;
Na classe Picareta: Bloco * matMine;

- Sobrescrita de método: chamar dentro do método da classe derivada o método correspondente da classe base usando ::

Em Picareta:

```
void Picareta::infoItem() const{
        Ferramenta::infoItem();
        cout << "uma Picareta de " << this->tipoS << '.' << endl;
```

}

- No main: criar um ponteiro da classe base para alocar memória para a classe derivada e chamar os vários métodos implementados