Zadanie 0 – set i map (1pkt)

Paulina należy do ESN i zebrała informacje o grupie studentów z programu Erasmus+ i wpisała je do funckji main:

```
#include <vector>
#include <map>
#include <set>
#include <iostream>
#include <string>

int main()
{
        std::vector<std::string> imiona =
        {"Jose", "Mark", "Maria", "Antonio", "Lorenzo", "Anna",
         "Mark", "Anna", "Carol", "Jose", "Maria", "Anna", "Hans"};

        std::vector<std::string> kraje =
         {"Hiszpania", "Holandia", "Portugalia", "Wlochy", "Wlochy",
         "Czechy", "Szwecja", "Hiszpania", "Niemcy", "Hiszpania",
         "Portugalia", "Niemcy", "Norwegia"};

        std::vector<std::string> kierunki =
        {"Socjologia", "Ekonomia", "Psychologia", "Ekonomia",
        "Fizyka", "Biotechnologia", "Psychologia", "Socjologia",
        "Psychologia", "Medycyna", "Ekonomia", "Psychologia", "Fizyka"};

        std::vector<std::string> urodziny =
        {"Maj", "Kwiecien", "Styczen", "Marzec", "Listopad",
        "Grudzien", "Maj", "Kwiecien", "Wrzesien", "Grudzien",
        "Styczen", "Marzec", "Sierpien"};

        /* Tutaj stworz mape, dodaj set stworzone z wektorow i wypisz */
        return 0;
}
```

Uzupełnij funkcję main, wg poniższych wytycznych:

1. Stwórz obiekt typu $std::map$, gdzie klucze będą typu $std::string$ a wartości typu $std::set < std::string >$. Typ $std::string$ to zawarty w pliku $< string >$ kontener na napisy, ułatwiający ich manipulowaniem, np. pozwalający przepisywać napisy tak jak zwykłe zmienne a nie jak tablice znaków. Nie musisz znać jego szczegółów, wiedz tylko, że często da się zrobić niejawną konwersję z łańczucha znaków, np. ”Napis”, na obiekt std::string.

2. Dodaj do mapy dane zebrane przez Paulinę. W tym celu trzeba stworzyć obiekty typu set z istniejących wektorów i dodać je do mapy z odpowiednimi kluczami. Klucze mogą być takie jak nazwy zmiennych (ale musisz je wpisać ręcznie).

3. Wypisz na standardowe wyjście klucze i wartości z mapy w formacie
   KLUCZ : WART WART WART

Output:
Imiona : Anna Antonio Carol Hans Jose Lorenzo Maria Mark

Kierunki : Biotechnologia Ekonomia Fizyka Medycyna Psychologia Socjologia
Kraje : Czechy Hiszpania Holandia Niemcy Norwegia Portugalia Szwecja Włochy
Urodziny : Grudzien Kwiecien Listopad Maj Marzec Sierpien Styczen Wrzesien

Podpowiedź: Mapy i zbiory są opisane w teoretycznym minimum, są na nie przykłady w ćwiczeniach. Obiekty typu set można szybko stworzyć z wektorów za pomocą iteratorów. Aby wypisać elementy z mapy również należy użyć iteratorów, podobnie jak przy wypisywaniu elementów wektora. Różnica polega na tym, że iterator w mapie pokazuje do dwóch obiektów, do klucza i do wartości. Jeżeli iterator nazywa się IT to dostęp do klucza dostajemy przez $IT- > left$ a do wartości przez $IT- > right$. Ponieważ wartościami w mapie są kolekcje set, to przy wypisywaniu trzeba będzie zagnieździć pętle for, żeby wypisać wszystko.

Zadanie 1 – podział projektu na kilka plików (1pkt)

Stwórz trzy pliki:

1. jk1_main.cpp
2. jk1.cpp
3. jk1.hpp

gdzie zamiast "jk" wstaw swoje inicjały. Skopiuj do "jk1_main.cpp" podaną niżej funkcję main

```
#include <iostream>//nie stawiaj spacji przed >
#include <cmath>

int main()
{
        TLorentzVector v(5, -2, -1, 3);
        std::cout << v.getX() << " " <<
                v.getY() << " " << v.getZ() << " "
                << v.getT() << std::endl;
        std::cout << v.Mag2() << " " << v.Mag()
                <<std::endl;

        double pi = 3.14159265359;
        v.setX(2*cos(pi/3)*sin(pi/2));
        v.setY(2*sin(pi/3)*sin(pi/2));
        v.setZ(2*cos(pi/2));
        v.setT(0);

        std::cout << v.getX() << " " <<
                v.getY() << " " << v.getZ() << " "
                << v.getT() << std::endl;
        std::cout << "phi=" << v.Phi() <<
        " theta=" << v.Theta() << std::endl;
        return 0;
}
```

W pliku nagłówkowym .hpp zdefiniuj klasę "TLorentzVector"[1]. Klasa ta ma reprezentować cztwerowektor, jakich używamy w teoriach relatywistycznych. Powinna mieć 4 prywatne składowe: x,y,z,t. Dla każdej ze składowych klasa ma mieć publiczne gettery i settery (trywialne!). Dodatkowo klasa ma dwa konstruktory:

[1]Wink wink!

```
TLorentzVector(); //ustawia wszedzie 0
TLorentzVector(double x, double y,
        double z, double t);
```

Konstruktor domyślny ustawia pola na 0. Działanie drugiego konstruktora jest (mam nadzieję) oczywiste. Dodatkowo klasa ma posiadać 4 metody:

```
double Mag2();
```

Metoda Mag2 liczy i zwraca długość czterowektora $s^2 = t^2 - x^2 - y^2 - z^2$.

```
double Mag();
```

Metoda Mag zwraca $\sqrt{s}$ jeśli $s >= 0$. W przeciwnym wypadku zwraca $-1 * \sqrt{-1 * s}$.

```
double Phi();
```

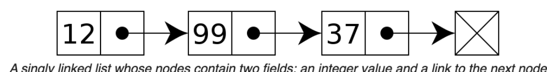
Metoda Phi zwraca kąt azymutalny (między współrzędnymi x i y) z przedziału $[0, 2\pi)$.

```
double Theta();
```

Metoda Theta zwraca kąt polarny, tj. kąt z przedziału $[-\pi, \pi]$, będący kątem pomiędzy osią OZ a kierunkiem wektora w 3D.
Uwaga: Wszystkie metody mają być zdefiniowane w pliku "jk1.cpp". W pliku nagłówkowym jedynie je deklarujemy, tzn. piszemy ich sygnaturę i po niej średnik, czyli pomijamy ciało metody w nawiasach klamrowych. W pliku .cpp musimy: 1) dołączyć plik nagłówkowy za pomocą include 2) definiując metody użyć odpowiedniej przestrzeni nazw, np. definiując gettera piszemy "double TLorentzVector::getX()return x;"
Podpowiedź: Krótki opis jak dzielić program na pliki oraz jak kompilować program z wielu plików jest w teoretycznym minimum. Jest tam również wyjaśnione, co to jest getter/setter.
Dla chętnych: Napisz i przetestuj globalną funkcję, która przyjmuje dwa czterowektory i zwraca ich iloczyn skalarny (w metryce Minkowskiego (+—)). Napisz i przetestuj kolejną funkcję, która zwraca interwał czasoprzestrzenny dla dwóch czterowektorów. **Output:**
5 -2 -1 3
-21 -4.58258
1 1.73205 -2.06823e-13 0
phi=1.0472 theta=1.5708

Zadanie 2 – lista jednokierunkowa (2pkt)

Lista jednokierunkowa to uporządkowana struktura danych, w której każdy element zawiera informację o elemencie następnym. Taka konstrukcja umożliwia iterację po elementach, wystarczy mieć dostęp do pierwszego elementu listy, żeby jeden po drugim dojść do dowolnego elementu położonego dalej. Ideę listy jednokierunkowej ilustruje poniższy obrazek:



Rysunek 1: Przykład listy jednokierunkowej. Każdy element zawiera liczbę całkowitą typu integer oraz łączę (link) do następnego elementu w liście. Ostatni element listy nie wskazuje na kolejny element, tylko na umowny "koniec". Źródło: `https://en.wikipedia.org/`

Zauważmy, że w liście jednokierunkowej możemy iterować jedynie w jednym kierunku (stąd nazwa). Ostatni element listy powinien wskazywać na umowny "koniec", aby zapewnić poprawne działania. Innym rozwiązaniem jest stworzenie listy cyklicznej, w której ostatni element wskazuje na pierwszy.

W C++ można bardzo łatwo stworzyć listę obiektów. Wystarczy wziąć jakąś klasę, powiedzmy T, i dodać do niej nowe pole o typie wskaźnikowym wskazującym na obiekt tej klasy, czyli pole o typie T*. Tworząc nowe obiekty klasy T, zapisujemy do wskaźników informacje o następnym elemencie aż do ostatniego. Ostatniemu elementowi listy przypisujemy nullptr, jako wskaźnik na kolejny element. Po utworzeniu listy nie potrzebujemy już informacji o położeniu obiektów w pamięci, czyli nie potrzebujemy zmiennych czy wskaźników na wszystkie elementy. Wystarczy nam dostęp do pierwszego elementu listy, z którego możemy dostać się do drugiego, z niego do trzeciego itd. Kiedy natrafimy na element, którego pole-wskaźnik będzie miało wartość nullptr, będziemy wiedzieć, że jesteśmy w ostatnim elemencie listy i będziemy mogli zakończyć pętlę. Łatwo można się domyślić jak zrealizować listę dwukierunkową, wystarczy dodać drugie pole-wskaźnik, tym razem wskazujące na element poprzedzający.
Po co używać list? Dla wydajności i elastyczności. Używając dynamicznej alokacji pamięci możemy tworzyć listy obiektów i nie musimy zachowywać do nich wskaźników, tylko do pierwszego elementu. Następnie, gdy chcemy usunąć element z listy, powiedzmy element k-ty, to zwalniamy po nim pamięć i zmieniamy pole-wskaźnik elementu (k-1)-tego, żeby wskazywało na element (k+1)-szy. Co ważne, nie musimy przesuwać elementów w pamięci! Listy można łączyć i zapętlać w prosty sposób. Inny przykład to sortowanie, w przypadku list nie przenosi ani nie kopiuje się żadnych elementów, a jedynie zmienia połączenia między nimi.
W zadaniu uzupełnij poniższe funkcje, aby zaimplementować funkcjonalność listy jednokierunkowej dla klasy Kontener.

```
void dodajEl(Kontener* biezacy, Kontener* nowy)
```

Ta funkcja ma dodać do listy nowy element o nazwie "nowy". Nowy element ma być dodany za elementem "biezacy". Sprawdź, czy któryś ze wskaźników jest NULL. Funkcja ma w taki sam sposób realizować zarówno dodawanie elementów na końcu listy, jak i w środku. Dodawania na początku nie trzeba uwzględniać.

```
void wypiszListe(Kontener* pocz)
```

Ta funkcja wypisuje na standardowe wyjście wartości pola "no" dla kolejnych elementów listy, zaczynając od "pocz". Kolejne wartości powinny być oddzielone spacjami a na koniec powinno nastąpić przejście do nowej linii (patrz przykładowy output). Tu również zabezpiecz się przed nullptr.

```
Kontener* usunElPoID(Kontener* pocz, unsigned no)

```

To najtrudniejsza z funkcji do napisania. Funkcja ta dostaje (umowny) początek listy dla którego porównuje wartość elementu listy z podanym do niej argumentem "no". Jeżeli obydwie wartości są równe, element jest usuwany za pomocą operatora *delete*, a połączenia w pozostałych są ustawiane tak, żeby lista nadal działała poprawnie. Procedura jest powtarzana dla wszystkich elementów listy, aż do napotkania nullptr. Skutkiem działania jest usunięcie **wszystkich** elementów listy o podanej wartości pola "no", a nie tylko pierwszego. Jeżeli żaden z elementów nie ma pożądanej wartości pola "no" to funkcja zostawi listę niezmienioną. Na koniec funkcja ma zwrócić wskaźnik na początek listy. Jest to nam potrzebne, bo szczególnym przypadkiem jest, gdy pierwszy (lub kilka pierwszych) elementów listy są do usunięcia. Będziesz potrzebował(a) w tej funkcji dużo if/else oraz pętli while/do...while.

Przykład:

```
#include <iostream> //nie ma spacji przed >
using namespace std;

class Kontener
{
        public:
```

```
                unsigned no;
                Kontener* nast;
                Kontener()
                {
                        this->no = 0;
                        this->nast = nullptr;
                };
                Kontener(unsigned no)
                {
                        this->no = no; //ta sama nazwa rozne rzeczy!
                        this->nast = nullptr;
                };
};

void dodajEl(Kontener* biezacy, Kontener* nowy)
{
        // dodaj element "nowy" do listy za "biezacy"
}

void wypiszListe(Kontener* pocz)
{
        //wypisuje pola no dla elementow listy
        //oddziela je spacja, na koniec przechodzi do nowej linii
}

Kontener* usunElPoID(Kontener* pocz, unsigned no)
{
 //usuwa wszystkie elementy o podanym no
 //i zwraca wskaznik na poczatek listy
}

int main()
{
        //pierwszy element listy
        Kontener* pocz = new Kontener();
        Kontener* biezacy = pocz;
        Kontener* dzies = nullptr;

        // dodaj elementy do listy
        for(unsigned ii=1; ii<33; ii++)
        {
                Kontener* nowy = new Kontener(ii%10);
                dodajEl(biezacy, nowy);
                biezacy = nowy;
                if(ii==10)
                        dzies = biezacy;
        }
        //wypisz cala liste
        wypiszListe(pocz);
```

```
        //dodaj po elemencie ii=10
        dodajEl(dzies, new Kontener(100));
        wypiszListe(pocz);
        //wypisz od elementu ii=10
        wypiszListe(dzies);

        //usun wszystkie elementy z id=2 po elemencie ii=10
        usunElPoID(dzies, 2);
        wypiszListe(pocz);

        //dodaje element z id=0 po pierwszym
        dodajEl(pocz, new Kontener(0));
        wypiszListe(pocz);
        for(unsigned ii=0; ii<=11; ii++)
        {
                //usun wszystkie elementy z id=ii
                pocz = usunElPoID(pocz, ii);
                wypiszListe(pocz);
        }
        //usuwamy ostatni element jaki zostal
        pocz = usunElPoID(pocz, 100);
        //probujemy wypisac pusta liste
        wypiszListe(pocz);
        return 0;
}
```

Output:
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 100 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 100 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 100 1 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1
0 0 1 2 3 4 5 6 7 8 9 0 100 1 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1
1 2 3 4 5 6 7 8 9 100 1 3 4 5 6 7 8 9 1 3 4 5 6 7 8 9 1
2 3 4 5 6 7 8 9 100 3 4 5 6 7 8 9 3 4 5 6 7 8 9
3 4 5 6 7 8 9 100 3 4 5 6 7 8 9 3 4 5 6 7 8 9
4 5 6 7 8 9 100 4 5 6 7 8 9 4 5 6 7 8 9
5 6 7 8 9 100 5 6 7 8 9 5 6 7 8 9
6 7 8 9 100 6 7 8 9 6 7 8 9
7 8 9 100 7 8 9 7 8 9
8 9 100 8 9 8 9
9 100 9 9
100
100
100
[ERROR] Nullptr!

Uwaga: Nie zmieniać definicji klasy ani maina. Nie tworzyć żadnych tablic ani kolekcji. Zmieniając listę zawsze pamiętaj o przełączeniu pól-wskaźników w odpowiednich elementach. Pole-wskaźnik ostatniego elementu ma być nullptr.

Podpowiedź: Uważaj na wskaźnik nullptr! W tym zadaniu należy go używać, ale trzeba być przy tym precyzyjnym. Zawsze się upewnij, że dany element istnieje zanim spróbujesz odczytać jego pole.

Dla chętnych: Pobaw się listami. Zaimplementuj listę dwukierunkową i/lub cykliczną i dostosuj napisane funkcje. Napisz funkcję, która usuwa n-ty element listy licząc od początku (lista jednokierunkowa/dwukierunkowa) albo od końca (konieczna lista dwukierunkowa albo 2 iteracje).

Zadanie 3 – Polimorfizm i fabryka (2pkt)

Polimorfizm i metody wirtualne

1. Napisz definicję klasy o nazwie "Owoc". Klasa ta ma być **abstrakcyjna** i posiadać jedną publiczną metodę o nazwie "powitanie". Powitanie nie przyjmuje argumentów ani nie zwraca żadnej wartości, jedyne co robi, to wypisuje na $std::cout$ napis "Jestem sobie pysznym owocem!". Powitanie ma być metodą **wirtualną!** Nie musisz definiować innych pól i konstruktorów dla tej klasy.

2. Stwórz trzy klasy dziedziczące po klasie Owoc ze specyfikatorem dostępu "public". Klasy noszą nazwy: "Jablko", "Gruszka" i "Banan". Dla każdej z klas zdefiniuj własną publiczną metodę o identycznej nazwie i sygnaturze jak $Owoc::powitanie$. Nie musisz definiować żadnych innych składowych.

3. Przetestuj działanie programu na mainie z poniższego szablonu

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
        //wektor wskaznikow na klase abstrakcyjna?
        vector<Owoc*> v;
        v.push_back(new Jablko);
        v.push_back(new Gruszka);
        v.push_back(new Banan);
        for(auto owoc : v)
                owoc->powitanie();
        //dostep do metody w klasie abstrakcyjnej
        v[1]->Owoc::powitanie();
        //zawsze pamietamy o zwolnieniu pamieci
        for(int ii=v.size(); ii>0; ii--)
                delete v[ii];
        return 0;
}
```

Output:
Jestem sobie pysznym jablkiem!
Jestem sobie pyszna gruszka!
Jestem sobie pysznym bananem!
Jestem sobie pysznym owocem!

Fabryka

Nie ma sensu wymyślać koła od nowa, dlatego w programowaniu używa się *wzorców projektowych*. Wzorce projektowe to gotowe rozwiązania często pojawiających się problemów, które są niezależne od używanego

języka (ale przybierają zależne od języka implementacje). Przykładem wzorca projektowego, wystepującego w wielu odmianach jest fabryka.
W pierwszej części zadania stworzyliśmy 3 klasy pochodne opisujące owoce. Łatwo sobie wyobrazić, że takich klas moglibyśmy mieć o wiele więcej. Nie chcemy za każdym razem tworzyć obiektów danej klasy, wolelibyśmy mieć jakiś interfejs, który w zależności od podanego parametru zwróci nam to co trzeba. Idealnie by było, gdyby w trakcie działania programu można było zdecydować jakie obiekty będą utworzone, a nie w momencie pisania programu. Powyższe zadania realizuje właśnie wzorzec fabryki. Implementacja fabryki w C++ polega na napisaniu klasy, która ma metodę przyjmującą identyfikator określający rodzaj obiektu do utworzenia. Metoda tworzy na stosie i zwraca zadany obiekt.
Napisz klasę o nazwie "Fabryka", która zawiera jedną **publiczną metodę statyczną**:

```
Owoc* uprawiajOwoc(string rodzaj)
```

W zależności od podanego napisu, metoda utworzy i zwróci wskaźnik na obiekt typu Jablko, Gruszka albo Banan. W przypadku podania nieznanego identyfikatora, metoda zwroci wskaźnik pusty. Działanie fabryki pozwoli przetestować zmodyfikowana funkcja main, w której ostatni z elementów tworzony jest w trakcie wykonywania programu na podstawie identyfikatora podanego przez użytkownika.

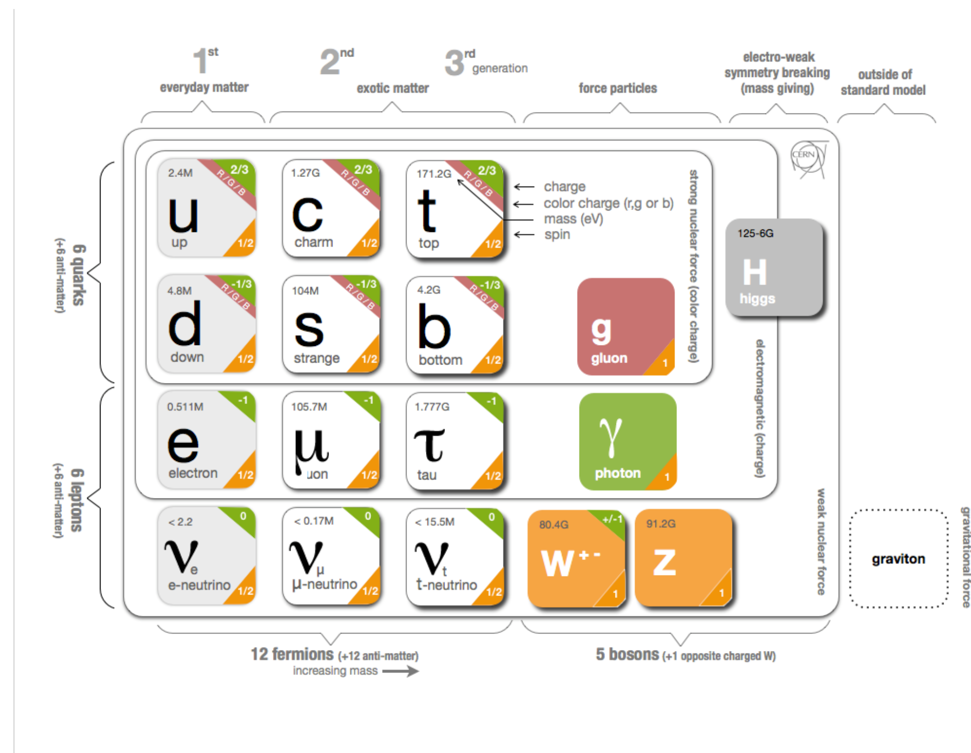
```
int main()
{
        //wektor wskaznikow na klase abstrakcyjna?
        vector<Owoc*> v;
        // v.push_back(new Jablko);
        // v.push_back(new Gruszka);
        // v.push_back(new Banan);
        v.push_back(Fabryka::uprawiajOwoc("jablko"));
        v.push_back(Fabryka::uprawiajOwoc("gruszka"));
        v.push_back(Fabryka::uprawiajOwoc("banan"));
        cout << "Podaj nazwe..." << endl;
        string nazwa;
        cin >> nazwa;
        v.push_back(Fabryka::uprawiajOwoc(nazwa));

        for(auto owoc : v)
        {
                if(owoc)
                        owoc->powitanie();
        }
        //dostep do metody w klasie abstrakcyjnej
        v[0]->Owoc::powitanie();
        //zawsze pamietamy o zwolnieniu pamieci
        for(int ii=v.size(); ii>0; ii--)
        {
                if(v[ii])
                        delete v[ii];
        }

        return 0;
}
```

Zadanie dodatkowe – Model Standardowy (3pkt)

Wykorzystaj dziedziczenie aby zaprogramować strukturę hierachiczną cząstek modelu standardowego. W tym celu należy stworzyć kilka klas opisanych poniżej, zdefiniować dla nich dziedziczenie (specyfikator dostępu public) oraz odpowiednie metody. Zadanie nie jest trudne, ale wymaga więcej kodu niż zazwyczaj. Ważne jest, żeby zachować porządek wcięć, klamer, nazw itd. inaczej zadanie zrobi się dużo trudniejsze.



Rysunek 2: Cząstki standardowe w Modelu Standardowym. Źródło: Google Image

Particle

Particle to klasa od której zaczniemy. Ma być abstrakcyjna, tzn. taka, której obiektu nie można utworzyć (patrz teoretyczne minimum). Powinna posiadać chronione pola:

```
protected:
        double m; //GeV
        double eCharge;
        long int PDG;
        bool red;
        bool green;
        bool blue;
        bool antiRed;
        bool antiBlue;
        bool antiGreen;
```

Gdzie kolejne pola zawierają informację o masie cząstki, jej ładunku w jednostkach ładunku elementarnego, kodzie cząstki. Dalej są zmienne, które opisują ładunek/antyładunek kolorowy[2]. Dla wszystkich pól należy zdefiniować metody typu get/set. Ponadto należy stworzyć 3 konstruktory:

[2]Nie jest to najbardziej elegancka implementacja, ale najprostsza.

```
        Particle();
        Particle(double mass, double charge, long int pDG);
        Particle(double mass, double charge, long int pdg,
                bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Pierwszy konstruktor (domyślny) ustawia wszystkie pola na 0. Drugi konstruktor ustawia wartości pól numerycznych, a pola odpowiadające kolorom na 0 (false). Ostatni konstruktor ustawia wszystkie pola. Ponieważ Particle ma być abstrakcyjna, to będzie potrzebować czysto wirtualnego destruktora

```
virtual ~Particle() = 0;
```

Boson

Boson to abstrakcyjna klasa reprezentująca bozony. Ma dziedziczyć (ze specyfikatorem public) po klasie Particle. Dodatkowo, ma posiadać zabezpieczone pole int spin, oraz odpowiedni getter i setter. Ma posiadać 3 konstruktory, podobnie do klasy Particle:

```
Boson();
Boson(double mass, double charge, long int pDG, int sp);
Boson(double mass, double charge, long int pdg, int sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu w konstruktorze domyślnym ma być 0.

Fermion

Boson to abstrakcyjna klasa reprezentująca fermiony. Ma dziedziczyć (ze specyfikatorem public) po klasie Particle. Dodatkowo, ma posiadać zabezpieczone pole double spin, oraz odpowiedni getter i setter. Ma posiadać 3 konstruktory, podobnie do klas Particle i Boson:

```
Fermion();
Fermion(double mass, double charge, long int pDG, double sp);
Fermion(double mass, double charge, long int pdg, double sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu w konstruktorze domyślnym ma być 1/2.

Lepton

Klasa Lepton reprezentuje leptony, fermiony nieoddziałujące silnie. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma tylko dwie metody-konstruktory.

```
Lepton();
Lepton(double mass, double charge, long int pdg, double sp);
```

Domyślną wartością spinu ma być 1/2. Wszystkie kolory mają być ustawione na 0.

Hadron

Klasa Hadron reprezentuje hadrony, fermiony oddziałujące silnie. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Hadron();
Hadron(double mass, double charge, long int pdg, double sp);
Hadron(double mass, double charge, long int pdg, double sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu ma być 1/2. Domyślnie kolory mogą być ustawione na 0.

Scalar

Klasa Scalar reprezentuje bozony skalarne, czyli o spinie 0. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Scalar()
Scalar(double mass, double charge, long int pdg);
Scalar(double mass, double charge, long int pdg,
 bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Wartość spinu ma zawsze być 0.

Vector

Klasa Vector reprezentuje bozony wektorowe, czyli o spinie 1. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Vector();
Vector(double mass, double charge, long int pdg);
Vector(double mass, double charge, long int pdg,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Wartość spinu ma zawsze być 1.

Przykładowy main

```
#include <iostream> //nie dawa spacji przed >
#include "zadanie2.hpp"//zmien nazwe na odpowiednia

using namespace std;

template<class T>
void print(T* p)
{
        cout << p->getM() << " " << p->getCharge() <<
                " " << p->getPDG() << " " << p->getSpin() <<
                " " << p->getRed() << " " << p->getGreen() <<
                " " << p->getBlue() << " " << p->getAntiRed() <<
                " " << p->getAntiGreen() << " " << p->getAntiBlue()
                << endl;
}

int main()
{
        //gauge vectors
        Vector gamma;
        gamma.setPDG(22);
        Vector Z(91.1876, 0, 23);
        print(gamma);
        print(Z);
        //leptons
        Lepton e(0.00051, -1, 11, 0.5);
        print(&e);
```

```
        Lepton mu(1.776, -1, 13, 0.5);
        print(&mu);
        //quarks
        Hadron u(0.00022, 2./3, 2, 0.5);
        u.setGreen(true);
        print(&u);
        Hadron ab(4.18, 1./3, -5, 0.5);
        ab.setAntiRed(true);
        print(&ab);
        //Higgs
        Scalar h(125, 0, 25);
        print(&h);
        //proton
        Hadron p(0.938, 1, 2212, 0.5);
        print(&p);
        //gravitino!
        Lepton gr(3000, 0, 1000039, 3./2.);
        print(&gr);


        return 0;
}
```

Uwaga: Klasy Particle, Boson i Fermion powinny być abstrakcyjne. Relacje dziedziczenie powinny być takie jak opisano, ze specyfikatorem dostępu public. Nie nadpisywać zdefiiowanych wyżej w hierarchi pól. Klasy najniższego rzędu, a więc Lepton, Hadron, Scalar i Vector nie mają nowych pól a jedynie konstruktory. Zadanie można zrobić w 2/3 plikach, ale można też w jednym.

Podpowiedź: Używaj kopiuj-wklej pisząc nowe klasy. W teoretycznym minimum jest opisane jak zrobić klasę abstrakcyjną.