Klasy i obiekty

Klasa to złożony typ zdefiniowany przez programistę i składający się ze zmiennych (zwanych *polami*), przechowujących dane, oraz posiadający funkcje (zwane *metodami*), które wykonują zaprogramowane czynności. Tworząc klasę, tworzymy zatem również nowy typ danych. Zmienne utworzone na podstawie zdefiniowanej klasy nazywamy *obiektami*. Każdy obiekt danej klasy posiada oddzielny zbiór opisujących go pól, które rezydują w pamięci operacyjnej. Metody są natomiast kodem wspólnym dla całej klasy, zatem w czasie działania programu istnieje w pamięci tylko jedna ich kopia, wywoływana w razie potrzeby na rzecz różnych obiektów. Klasy tworzy się, używając słowa kluczowego `class`, nazwy klasy oraz jej ciała zawartego w klamrach. Przykładowa definicja klasy wygląda następująco:

```
class Person {
    int m_age;  // pole składowe klasy reprezentujące wiek osoby
    void print();  // metoda wypisuje na ekranie dane osoby
}; // definicję klasy kończymy znakiem średnika, brak powoduje błąd kompilacji
```

Ważne jest, by pamiętać, że definicja klasy musi być zakończona średnikiem. Jest dobrą praktyką, by nazwy klas zaczynać od dużej litery.

Nie zawsze wskazane jest, aby pola były dostępne na zewnątrz klasy. Aby kontrolować zakres ich widoczności, można posłużyć się tak zwanymi kwalifikatorami dostępu. W języku C++ mamy następujące kwalifikatory dostępu do składowych: `public` – dostęp do składników klasy jest dozwolony wszędzie, nawet spoza ciała klasy, `private` – dostęp do składników klasy jest zabroniony spoza ciała klasy, `protected` – dostęp do składników klasy jest dozwolony tylko z ciała klasy (tak jak `private`) oraz w klasach pochodnych naszej klasy bazowej. Kwalifikatory dostępu służą jedynie do projektowania klas zgodnych z pewnymi założeniami programisty. Generalna zasada jest taka, że ważne dla klasy pola powinny być prywatne. Funkcje w klasach (czyli metody) tworzymy na takiej samej zasadzie jak w innych miejscach programu. Metody także podlegają kwalifikatorom dostępu. Możemy więc tworzyć prywatne metody, które nie będą mogły zostać wywołane spoza ciała klasy.

Kiedy w pamięci tworzony jest obiekt danej klasy, zawsze istnieje konieczność wypełnienia jego pól odpowiednimi danymi początkowymi. W tym celu wykorzystywana jest specjalna metoda zwana *konstruktorem*. Konstruktor to metoda, która wykona się zawsze podczas tworzenia obiektu danej klasy. Jego zadaniem jest inicjowanie obiektu i wypełnienie pól początkowymi danymi. Jeżeli nie stworzymy konstruktora, kompilator zrobi to za nas niejawnie. Będzie to wtedy pusty konstruktor bezparametrowy. W jednej klasie może istnieć wiele konstruktorów różniących się użytymi argumentami. Konstruktory muszą spełniać następujące warunki: nazwa konstruktora musi być taka sama jak nazwa klasy, nie może być podany typ zwracany (nawet `void`) oraz deklaracja przynajmniej jednego konstruktora musi znajdować się w sekcji `public`.

```
class Person
{
private:   // specyfikator dostępu pola w klasie, na ogół private
      string m_name;   // pole reprezentujące imię
      int m_age;   // pole reprezentujące wiek
public:  // publiczna część klasy
      Person(const string& name = "", int age = 0) : m_name{name}, m_age{age}
//konstruktor
     {} // w tym przypadku ciało konstruktora jest puste
     void print_info(void) // metoda wypisuje na ekranie dane bieżącego obiektu
     {
          cout<<"My name is "<<m_name<<" and I'm "<<m_age<<" years old."<<endl;
     }
};

int main() {
```

```
Person p("Peter", 20);  // tworzymy nowy obiekt klasy Person za pomocą
zdefiniowanego konstruktora
p.print_info();  // wywołujemy metodę print_info dla obiektu p
return 0; }
```

W powyższym przykładzie warto zwrócić uwagę na następujące rzeczy: pola składowe są w sekcji `private` (na ogół jest właśnie tak), nazwy pól zaczynają od przedrostka `m_` (jest dobrą praktyką, by nazwy pól zaczynać od takiego lub podobnego przedrostka, gdyż dzięki temu łatwiej odróżnić pola od innych zmiennych mogących występować w metodach klasy), konstruktor jest w sekcji `public` (jest to konieczne), jako argument metody jest użyte wyrażenie `const nazwa_klasy&` (jest to standardowy sposób przekazania innego obiektu do funkcji; słowo `const` oznacza, że wewnątrz klasy nie modyfikujemy stanu przekazanego obiektu, znak `&` oznacza zaś, że przekazujemy do naszej metody sam obiekt, a nie jego kopię), za pomocą operatora `=` została przypisana wartość domyślna dla argumentów metody (jest to rozwiązanie opcjonalne), wartości pól klasy są zainicjowane argumentami konstruktora za pomocą tak zwanej *listy inicjalizacyjnej* (jest to najlepszy sposób), obiekt tej klasy jest utworzony za pomocą jej nazwy, po czym następuje zmienna, pod którą będzie on dostępny w programie (jest to standardowy sposób), . jest operatorem, za pomocą którego mamy dostęp do publicznych składowych.

Obiekty można również tworzyć dynamicznie za pomocą operatora `new`. W przypadku tak utworzonych obiektów dostęp do publicznych pól oraz metod odbywa się za pomocą operatora `->`. W tym przypadku należy również pamiętać o konieczności usunięcia za pomocą operatora `delete` takiego obiektu z pamięci gdy nie jest on już potrzebny. Na przykład:

```
Person* p = new Person("Peter", 20); // tworzymy obiekt dynamicznie, w rezultacie
otrzymujemy wskaźnik do obiektu klasy Person
p->print_info(); // do metody składowej odwołujemy się za pomocą operatora ->
delete p; // koniecznie pamiętajmy o usunięciu obiektu z pamięci gdy nie jest już
potrzebny
```

Zgodnie z zasadami programowania obiektowego pola klasy są prawie zawsze prywatne, dlatego jeśli w programie zachodzi konieczność ich odczytania lub zmodyfikowania, to aby umożliwić wykonanie tych operacji, tworzymy w tej klasie odpowiednie do tego metody zwane *setterami* (z ang. *set* - ustawienie wartości pola) oraz *getterami* (z ang. *get* - pobranie wartości pola). Na przykład dla naszej klasy `Person` gettery i settery będą mieć następującą postać:

```
string get_name(void) const { return m_name; } // getter dla pola m_name
int get_age(void) const { return m_age; } // getter dla pola m_age
void set_name(const string& name) { m_name = name; } // setter dla pola m_name
void set_age(int age) { m_age = age; } // setter dla pola m_age
```

Gettery i settery umieszczamy zawsze w sekcji `public`. Typ zwracany przez getter jest zawsze taki sam jak typ pola, którego wartość zwraca. Słowo `const` występujące w definicji gettera oznacza, że metoda ta nie modyfikuje wartości pól danej klasy. Warto zwrócić uwagę na fakt, że dzięki użyciu przedrostka `m_` w nazwach pól, wewnątrz setterów można łatwo odróżnić pola klasy ich argumentów oraz innych zmiennych, co poprawia czytelność kodu.

Opcjonalnie można zdefiniować w klasie specjalną metodę zwaną *destruktorem*, która wywoływana jest w momencie usuwania z pamięci obiektów tej klasy. Destruktory na ogół używane są do zwalniania zasobów takich jak dynamicznie przydzielona pamięć, otwarte pliki itp. Destruktor ma tę samą nazwę co klasa, poprzedzoną jednak znakiem `~`. Jeśli programista sam nie zdefiniuje destruktora, to kompilator automatycznie wygeneruje destruktor domyślny dla danej klasy. W przeciwieństwie do konstruktorów w każdej klasie może znajdować się tylko jeden destruktor. Na przykład w wypadku naszej klasy `Person` definicja destruktora może wyglądać następująco:

```
~Person() {} //destruktor klasy Person
```

Aby poprawić czytelność oraz organizację kodu źródłowego, można zdefiniować metody na zewnątrz deklaracji klasy. W tym wypadku definicję metody należy poprzedzić wyrażeniem: `nazwa_klasy::`. Na przykład w naszej klasie `Person` możemy zadeklarować metodę `set_age` w obrębie deklaracji samej klasy, a jej definicję umieścić poza tą deklaracją.

```
class Person // ta część będzie na ogół w pliku o nazwie Person.h
{
  public:
    void set_age(int age); // deklaracja settera wewnątrz klasy
};
// zaś, definicje metod będą w pliku o nazwie Person.cpp
void Person::set_age(int age) { m_age = age; } // definicja settera na zewnątrz
klasy, na ogół w pliku o nazwie Person.cpp
```

Tego typu podejście stosuje się na ogół w wypadku bardziej rozbudowanych klas. Dodatkowo deklarację samej klasy umieszcza się w pliku nagłówkowym o nazwie `nazwa_klasy.h`, definicje metod zaś w pliku `nazwa_klasy.cpp`. Aby użyć taką klasę w programie, należy na jego początku załączyć plik nagłówkowy za pomocą dyrektywy preprocesora `#include "Person.h"`. W wypadku prostych klas nie ma potrzeby rozdzielania deklaracji klasy i definicji metod na dwa oddzielne pliki.

Niekiedy zachodzi konieczność utworzenia nowego obiektu danej klasy na podstawie istniejącego już wcześniej innego obiektu tej samej klasy. Aby to umożliwić należy stworzyć w klasie specjalny typ konstruktora zwany konstruktorem kopiującym. Konstruktor ten posiada jeden argument, którym będzie stała referencja do obiektu danej klasy. Zadaniem konstruktora kopiującego jest skopiowanie wartości wszystkich pól składowych klasy z obiektu źródłowego do docelowego. Dla naszej klasy `Person`, konstruktor kopiujący będzie miał postać:

```
Person (const Person& rhs)
{
  m_name = rhs.get_name(); // skopiuj pole m_name
  m_age = rhs.get_age(); // skopiuj pole m_age
}
```

*

Często zdarza się, że istnieje w programie konieczność porównania wartości dwóch zmiennych. Ponieważ liczby całkowite mają w pamięci komputera dokładną reprezentację dlatego w przypadku zmiennych typu całkowitego (np. `int, char`) działanie operatorów porównania `>`, `<`, `==`, `!=` jest zawsze dokładne. Oznacz to, że jeśli napiszemy w programie: `int x; if (x==0) {...}` to możemy mieć pewność, że instrukcje w tych klamrach wykonają się jedynie w przypadku gdy wartość zmiennej `x` będzie równa zero. Nieco inaczej rzecz wygląda w przypadku liczb zmiennoprzecinkowych. Ponieważ liczby te nie zawsze mają dokładną reprezentację w pamięci komputera dlatego w przypadku typów zmiennoprzecinkowych (np. `float`, `double`) wynik działania niektórych operatorów porównania może być niezgodny z oczekiwaniami programisty. Rozważmy na przykład poniższy program:

```
float x=1.0/3.0;

if (x==(1.0/3.0))
{
        cout << "true" << endl;
}
else
{
      cout << "false" << endl;
```

```
}
```

Ku zaskoczeniu niedoświadczonych programistów wynikiem działania powyższego programu nie będzie napis `true` ale `false`. Jest tak dlatego, że ułamek ⅓ nie posiada dokładnej reprezentacji w pamięci komputera. Zatem wartość ⅓ jest przechowywana w przybliżeniu czyli z pewnym błędem. W przypadku wyrażenia `float x=1.0/3.0` kompilator najpierw wykona dzielenie liczby 1.0 przez 3.0 za pomocą typu `double` a następnie dokona konwersji wyniku do typu `float`. Ponieważ typ `float` ma mniejszą precyzję niż `double` dlatego konwersja ta spowoduje zwiększenie błędu reprezentacji liczby ⅓. Można sprawdzić, że w powyższym programie zmienna `x` będzie mieć dokładnie wartość `0.33333334326744079590` zaś wyrażenie 1.0/3.0 ma wartość `0.33333333333333331483`. Dlatego warunek `if (x==(1.0/3.0))` nie jest spełniony. Inny przykład:

```
double a = (0.3 * 3.0) + 0.1;
double b = 1.0;
if (a==b)
{
  cout << "true" << endl;
}
else
{
  cout << "false" << endl;
}
```

Powstaje zatem pytanie jak należy porównywać liczby zmiennoprzecinkowe? W przypadku liczb zmiennoprzecinkowych można posłużyć się jedną z poniższych funkcji. Funkcja `approximatelyEqual` implementuje ideę przybliżonej równość zaś funkcja `essentiallyEqual` implementuje ideę istotnej równości dwóch liczb zmiennoprzecinkowych.

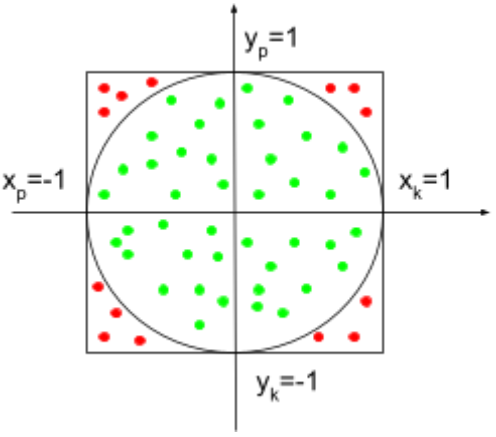
```
bool approximatelyEqual(float a, float b, float epsilon)
{
    return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool essentiallyEqual(float a, float b, float epsilon)
{
    return fabs(a - b) <= ( (fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

Jak jest różnica pomiędzy powyższymi funkcjami? Funkcja `approximatelyEqual` mówi o tym czy różnica pomiędzy `a` i `b` jest mniejsza niż dopuszczalny błąd `epsilon` w odniesieniu do większej wartości spośród `a` i `b`. Innymi słowy, oznacza to, że dwie wartości `a` i `b` są “wystarczająco blisko” i dlatego można powiedzieć, że są w przybliżeniu równe. Funkcja `essentiallyEqual` mówi o tym czy różnica pomiędzy `a` i `b` jest mniejsza niż dopuszczalny błąd epsilon w odniesieniu do mniejszej wartości spośród `a` i `b`. Innymi słowy, oznacza to, że dwie wartości `a` i `b` różnią się mniej niż dopuszczalny błąd `epsilon` w każdych obliczeniach zatem być może nie są one dokładnie równe ale są “istotnie równe przy zadanym błędzie”. Istotna równość implikuje przybliżoną równość ale nie *vice versa*. Zatem istotna równość jest silniejszym warunkiem niż przybliżona równość. Istotna równość nie jest przechodnia, to znaczy, że jeśli `a` jest istotnie równe `b` i `b` jest istotnie równe `c` to a jest w przybliżeniu równe `c` (przy innym `epsilon`).

Podsumowując, należy zawsze założyć, że działania na liczbach zmiennoprzecinkowych są obarczone choćby minimalnym błędem i jest niezależne od języka programowania. Z tego powodu należy być bardzo uważnym przy zastosowaniu operatora `==` dla liczb zmiennoprzecinkowych. Zatem jeśli chcemy sprawdzić równość dwóch liczb zmiennoprzecinkowych zamiast operatora `==` lepiej posłużyć się funkcją `approximatelyEqual` lub `essentiallyEqual`.

zadania

1. Stwórz klasę o nazwie `Point2D` reprezentującą punkt na płaszczyźnie. Klasa powinna posiadać:
   a. pola reprezentujące współrzędne punktu,
   b. parametryczny konstruktor z przypisanymi wartościami domyślnymi argumentów,
   c. gettery i settery dla współrzędnych punktu.
2. Stwórz klasę o nazwie `Circle` reprezentującą okrąg na płaszczyźnie. Klasa powinna posiadać:
   a. pola reprezentujące położenie okręgu oraz jego promień,
   b. parametryczny konstruktor z przypisanymi wartościami domyślnymi argumentów,
   c. metodę `bool is_inside(const Point2D& p)`, która zwraca wartość `true`, gdy punkt `p` znajduje się wewnątrz tego okręgu, oraz `false` w przeciwnym wypadku.
3. Stworzona przez Stanisława Ulama metoda Monte Carlo jest algorytmem numerycznym umożliwiającym oszacowanie wartości całek. Załóżmy, że chcemy obliczyć całkę z funkcji $f(x)$ w przedziale $[x_p, x_k]$. Definicja całki oznaczonej Riemanna mówi nam, że wartość całki równa jest polu obszaru pod wykresem krzywej w zadanym przedziale całkowania. Załóżmy, iż wiemy, że wartości funkcji w obszarze całkowania mieszczą się w przedziale $[y_p, y_k]$. Pole prostokąta wyznaczonego przez przedział całkowania: $[x_p, x_k]$ oraz zakres wartości funkcji $[y_p, y_k]$ w tym przedziale wynosi: $P = |x_k - x_p| \cdot |y_k - y_p|$. Metoda Monte Carlo przebiega w następujący sposób: losujemy *n* punktów w obrębie prostokąta, po czym obliczamy iloraz punktów znajdujących się pod krzywą i całkowitej ilości wylosowanych punktów, na końcu zaś w oparciu o ten iloraz obliczamy wartość całki. W szczególnym wypadku krzywą, pod którą obliczamy pole, może być okrąg o środku w punkcie (0,0) i promieniu równym 1, jak to jest pokazane na poniższym rysunku. Przyjmijmy, że n_i oznacza liczbę wylosowanych punktów znajdujących się wewnątrz okręgu. Zgodnie z metodą Monte Carlo będzie zachodzić następująca proporcja: $\frac{P_{koła}}{P_{kwadratu}} = \frac{n_i}{n}$, z której można oszacować pole koła. Posługując się klasami `Point2D` oraz `Circle` oraz metodą Monte Carlo, napisz program, który wyznaczy przybliżoną wartość liczby π.



Aby zapewnić równomierny rozkład wylosowanych punktów, posłuż się generatorem liczb pseudolosowych Mersenne Twister, który jest zadeklarowany w pliku nagłówkowym `random`, oraz zegarem wysokiej rozdzielczości zadeklarowanym w pliku `chrono`. Najpierw tworzymy generator za pomocą instrukcji

`mt19937 gen(chrono::system_clock::now().time_since_epoch().count());`. Następnie za pomocą instrukcji `uniform_real_distribution<double> distribution{-1.0,1.0};` tworzymy dystrybuantę rozkładu równomiernego w przedziale $[-1; 1]$. Kolejne liczby pseudolosowe otrzymujemy za pomocą instrukcji `distribution(gen);`.

4. Stwórz klasę o nazwie `TQuadEq` reprezentującą trójmian kwadratowy. Klasa powinna zawierać:
    a. pola reprezentująca współczynniki trójmianu
    b. Konstruktor domyślny oraz parametryczny
    c. Gettery oraz settery dla wszystkich pól klasy
    d. metodę `double ComputeDelta(void) const` zwracającą wartość delty trójmianu
    e. metodę `int GetNumRoots(const double delta) const` zwracającą liczbę pierwiastków rzeczywistych trójmianu
    f. metodę `GetRoots(double& root1, double& root2) const` zwracającą rzeczywiste pierwiastki trójmianu.
    g. W funkcji main() napisz prosty program demonstrujący wykorzystanie klasy `TQuadEq`. Program wczytuje z klawiatury współczynniki trójmianu a następnie wyświetla na ekranie jego pierwiastki.
5. Utwórz klasę o nazwie `SmartArray`, która reprezentuje tablicę liczb całkowitych. Klasa powinna zawierać następujące pola i metody:
    a. konstruktor, którego argumentem jest rozmiar tablicy i który wypełnia tablicę liczbami losowymi oraz wypisuje na ekranie komunikat informujący o utworzeniu obiektu wraz z wartościami argumentów konstruktora,
    b. konstruktor, którego argumentem jest inny obiekt tej samej klasy (tak zwany konstruktor kopiujący) i który też wypisuje na ekranie komunikat o utworzeniu obiektu za jego pomocą,
    c. destruktor, który wypisuje na ekranie komunikat informujący o wywołaniu destruktora,
    d. metodę o nazwie `at`, która zwraca element tablicy pod wskazanym indeksem. (Posłuż się makrem `assert`, by sprawdzić, czy podany indeks jest poprawny).
    e. metodę, która umożliwia ustawienie elementu tablicy pod wskazanym indeksem,
    f. metodę, która zwraca wartość maksymalną w tablicy,
    g. metodę o nazwie `print`, która wypisuje na ekranie zawartość tablicy,

    W funkcji `main` utwórz obiekt klasy `SmartArray` o liczbie elementów 16. Za pomocą metody `print` wypisz na ekranie zawartość tablicy. Zmodyfikuj dowolny element w tablicy, a następnie wypisz na ekranie maksymalną wartość w tablicy. Ponownie wypisz na ekranie zawartość tablicy. Następnie utwórz nowy obiekt, posługując się konstruktorem kopiującym. Wypisz na ekranie zawartość nowego obiektu. Sprawdź działanie makra `assert` w metodzie `at`, podając jako argument niepoprawną wartość indeksu.