Opracowanie: dr inż. Łukasz Maliński i mgr inż. Błażej Nycz, Politechnika Śląska, 2022

LABORATORIUM 2.2
TWORZENIE INSTANCJI I INTERAKCJA Z OBIEKTAMI

Poruszane zagadnienia z dziedziny programowania:

- Składniki statyczne - **wykład 6, Opus Magnum C++11**
- Konstruktory i destruktor - **wykład 6, Opus Magnum C++11**
- Instancje stałe i pola mutable - **wykład 6, Opus Magnum C++11**

Umiejętności do opanowania:

- definiowanie konstruktorów domyślnych i argumentowych, oraz destruktora,
- stosowanie różnych sposobów na inicjalizację wartości pól klasy,
- praktyczne posługiwanie się statycznymi składnikami klasy,
- przygotowanie klasy do pracy z instancjami chronionymi przed zapisem,
- posługiwanie się wskaźnikami do metody,

Oznaczenia odnośnie samodzielności pracy:

■ – **Ten problem koniecznie rozwiąż w pełni samodzielnie.** Możesz posługiwać się literaturą, wykładami i dokumentacją w celu sprawdzenia niuansów składniowych, ale nie szukaj gotowych rozwiązań i algorytmów. **Jest to bardzo ważne z punktu widzenia nauki i oszukiwanie przyniesie tylko poważne braki w późniejszych etapach nauki!**

▲ – Rozwiązując ten problem możesz posiłkować się rozwiązaniami zapożyczonymi od innych programistów, **ale koniecznie udokumentuj w kodzie źródło.** Nie kopiuj kodu bezmyślnie, tylko dostosuj go do kontekstu zadania. **Koniecznie musisz dokładnie zrozumieć rozwiązanie, które adoptujesz, gdyż inaczej wiele się nie nauczysz!**

● – Rozwianie tego problemu właśnie polega na znalezieniu i użyciu gotowego rozwiązania, **ale koniecznie podaj źródło.** Nie będzie wielkiej tragedii jeśli wykażesz się tylko ograniczonym zrozumieniem rozwiązania, gdyż nie musi być ono trywialne.

Przykładowe zadania do rozwiązania w ramach samodzielnej nauki (nie oceniane):

Zadanie A (z rozwiązaniem):

▲ Zdefiniuj klasę **Notatka** do opisu prostej notatki przypominającej o czymś istotnym. Klasa powinna przechowywać informację o treści notatki (tekst do 1000 znaków), dacie i czasie z nią skojarzonej (dokładność do sekundy) i stanie ważności: { zwykła, ważna, krytyczna }.

Wskazówki: Przyjmij pole tekstowe jako tablicę o 1000 znaków. Zastanów się jak przechować złożoną informację o dacie i czasie. Znajdź w dokumentacji języka C gotowy złożony typ danych. Koniecznie trzymaj się rozwiązań dostępnych dla języka C, gdyż te dla C++ są na tym etapie zbyt skomplikowane. Stan ważności opisz prostym typem wyliczeniowym.

▲ Zdefiniuj dla klasy **Notatka** podstawowe akcesory, ale nie trudź się z implementacją ograniczeń dla daty. Dostosuj getery do pracy z instancjami stałymi.

Wskazówki: W nagłówkach geterów dodaj specyfikator const za nawiasem argumentów, aby zaznaczyć, że taki geter można wywołać dla instancji stałej, lub chronionej przed zapisem.

▲ Zdefiniuj dla klasy **Notatka** konstruktor domyślny, ustawiający treść notatki na: „Brak wpisu”, datę na aktualną (z chwili utworzenia notatki) i zwykłą ważność.

Wskazówki: Jedyną trudniejszą sprawą jest to znalezienie w dokumentacji języka C, jak ustawić aktualną datę na bazie zegara systemowego.

▲ Zdefiniuj dla klasy **Notatka** 2 konstruktory argumentowe:

- pierwszy pozwalający na zadanie samej treści (reszta jak w konstruktorze domyślnym)
- drugi, pozwalający zadać wartości wszystkich pól (datę przesyłaj jako całość).

Wskazówki: Zastanów się jak zastosować zasadę DRY, zminimalizować powtarzanie tego samego kodu w konstruktorach.

▲ Zdefiniuj dla klasy **Notatka** operator << wypisujący notatkę na ekranie w dowolny czytelny sposób.

Wskazówki: Do wyświetlania stanu rodzaju ważności użyj nazw opisowych. Posłuż się w tym celu tabelaryzacją logiki.

Przykładowe rozwiązanie:

```
#include <ctime>

using tekst = char[1000];
enum class RodzajWaznosci { zwykla, wazna, krytyczna };

class Notatka
{
private:
  tekst m_tresc;
  tm m_data;          // typ danych z j. C do przechowywania daty
  RodzajWaznosci m_waznosc;
public:

  // Akcesory:
  void setTresc(const char* tresc) { strncpy(m_tresc, tresc, 1000); }
  const char* getTresc() const { return m_tresc; }
  void setData(const tm& data) {    m_data = data; }
  tm getData() const { return m_data; }
  void setWaznosc(RodzajWaznosci waznosc) { m_waznosc = waznosc; }
  RodzajWaznosci getWazonsc() const { return m_waznosc; }

  // Konstruktory argumentowe:
  Notatka(const char* tresc, tm& data, RodzajWaznosci waznosc)
    : m_data(data), m_waznosc(waznosc)
  {
    setTresc(tresc);
  }
  Notatka(const char* tresc)
    : m_waznosc(RodzajWaznosci::zwykla)
  {
    setTresc(tresc);
    time_t curTime = time(0);
    m_data = *localtime(&curTime);
  }

  // Konstruktor domyślny:
  Notatka()
    : Notatka("Brak Wpisu")
  {}

  /* Tu użyliśmy tzw. delegacji konstruktora. Konstruktor domyślny zanim się wykona
     zleca wstępne przygotowanie instancji zgodnie z innym konstruktorem.*/
};

// Operator <<:
std::ostream& operator<<(std::ostream &str, const Notatka &notka)
{
  const char* nazwy[3] = {"ZW","WA","KR"};
  tm data = notka.getData();
```

```
  char buffer[100];
  sprintf(buffer, "%s", asctime(&data));
  buffer[strlen(buffer)-1] = 0;

  /* Trzy powyższe linie usuwają znak \n, który jest automatycznie dodawany przez
     funkcję asctime(). Inaczej wszytko po dacie wypisane będzie w nowej linii.
     Funkcja asctime zamiania wpis w strukturze na formatowany c-string, gotowy
     do wypisania na ekranie. */

  str << buffer << " (" << nazwy[(int)notka.getWazonsc()]
      << "): " << notka.getTresc();
  return str;
}

int main()
{
  Notatka nowa_1("Testujemy");
  cout << nowa_1 << endl;

  Notatka nowa_2;
  cout << nowa_2 << endl;

  //Utworzenie daty na jutro:
  time_t curTime = time(0) + 24*3600; //liczba sekund w 24 godzinach
  tm data = *localtime(&curTime);

  Notatka nowa_3("Jakis wpis", data, RodzajWaznosci::krytyczna);
  cout << nowa_3 << endl;

}
```

Zadanie B (z rozwiązaniem):

▲ Rozbuduj klasę **Notatka** (z zadania A), o możliwość zliczania ile w programie jest notatek zwykłych, ważnych i krytycznych.

Wskazówki: Są trzy możliwe stany ważności notatek, więc potrzebujemy trzy statyczne liczniki instancji. Każdy z nich powinien być inicjalizowany wartością 0 przy starcie programu. Każdy konstruktor po ustawieniu pola ważności, powinien zwiększyć właściwy licznik, a destruktor go zmniejszyć. Zauważ, że zmiana ważności notatki, w trakcie jej istnienia także powinna zmienić wartości liczników.

▲ Wyklucz w klasie **Notatka** mechanizmy kopiowania i przenoszenia, aby nie zakłócały działania liczników.

Wskazówki: Na tym etapie nauki kopiowanie i przenoszenie nie było jeszcze omawiane. Ponieważ wykorzystuje ono działanie specyficznych konstruktorów, a nie chcemy aby zakłócało to zliczanie instancji, to wykluczamy możliwość kopiowania i przenoszenia. Szczegóły jak to zrobić są podane na wykładzie.

■ Napisz metodę statyczną wypisującą na ekranie statystyki notatek w postaci: „ ZW / WA / KR ”, gdzie ZW, WA, KR to wartości liczników instancji.

Wskazówki: Absolutnie nie pisz seterów statycznych do liczników. Te wartości nie mogą być ręcznie zmieniane. Getery nie są tu potrzebne, metoda statyczna ma pełny i bezpośredni dostęp do składników prywatnych.

Przykładowe rozwiązanie (wyeksponowano nowy kod):

```
#include <ctime>

using tekst = char[1000];
enum class RodzajWaznosci { zwykla, wazna, krytyczna };

class Notatka
{
private:
  tekst m_tresc;
```

```
  tm m_data;          // typ danych z j. C do przechowywyania daty
  RodzajWaznosci m_waznosc;

  // Liczniki instancji (tablica):
  static long long s_licznik[3];

public:

  // Akcesory:
  void setTresc(const char* tresc) { strncpy(m_tresc, tresc, 1000); }
  const char* getTresc() const { return m_tresc; }
  void setData(const tm& data) { m_data = data; }
  tm getData() const { return m_data; }

  void setWaznosc(RodzajWaznosci waznosc)
  {
    s_licznik[(int)m_waznosc]--;
    m_waznosc = waznosc;
    s_licznik[(int)m_waznosc]++;
    /* Przed zmianą redukujemy stary licznik, a po zmianie zwiększamy nowy */
  }

  RodzajWaznosci getWazonsc() const { return m_waznosc; }

  // Konstruktory argumentowe:
  Notatka(const char* tresc, tm& data, RodzajWaznosci waznosc)
    : m_data(data), m_waznosc(waznosc)
  {
    setTresc(tresc);
    s_licznik[(int)m_waznosc]++;
  }
  Notatka(const char* tresc)
    : m_waznosc(RodzajWaznosci::zwykla)
  {
    setTresc(tresc);
    time_t curTime = time(0);
    m_data = *localtime(&curTime);
    s_licznik[(int)m_waznosc]++;
  }

  /* Dlaczego nie skorzystamy z setera? Otóż, seter przez zmianą obniża licznik, gdyż
     zakłada, że instancja już istnieje i jest gdzieś zliczona. Konstruktor startuje gdy
     instancja jeszcze  nie jest zliczona, więc użycie setera zakłócało by zliczanie. Nie
     chcemy też nadmiernie komplikować kodu stera tylko po to by go na siłę użyć
     w konstruktorze. */

  // Konstruktor domyślny:
  Notatka()
    : Notatka("Brak Wpisu")
  {}

  /* W konstruktorze domyślnym, nie zwiększamy licznika instancji, gdyż robi to
     już konstruktor, do którego wydelegowaliśmy pracę.*/

  // Destruktor:
  ~Notatka()
  {
    s_licznik[(int)m_waznosc]--;
  }

  // Metoda satyczna do statsystyk:
  static void statystyki()
  {
    cout << s_licznik[0] << " / " << s_licznik[1] << " / " << s_licznik[2]  << endl;
  }
};
```

```
// Incjalizacja liczników instacji:
long long Notatka::s_licznik[3] = {};


// Operator <<:
std::ostream& operator<<(std::ostream& str, const Notatka& notka)
{
	const char* nazwy[3] = { "ZW","WA","KR" };
	tm data = notka.getData();

	char buffer[100];
	sprintf(buffer, "%s", asctime(&data));
	buffer[strlen(buffer) - 1] = 0;

	str << buffer << " (" << nazwy[(int)notka.getWazonsc()]
		<< "): " << notka.getTresc();
	return str;
}

int main()
{
	Notatka nowa_1("Testujemy");
	cout << nowa_1 << endl;

	Notatka nowa_2;
	cout << nowa_2 << endl;

	//Utworzenie daty na jutro:
	time_t curTime = time(0) + 24 * 3600; //liczba sekund w 24 godzinach
	tm data = *localtime(&curTime);

	Notatka nowa_3("Jakis wpis", data, RodzajWaznosci::krytyczna);
	cout << nowa_3 << endl;

	Notatka::statystyki();

	nowa_2.setWaznosc(RodzajWaznosci::wazna);

	Notatka::statystyki();

}
```

Zadania domowe (oceniane)

UWAGA: Można oddać tylko jedno zadanie domowe. Ocena za zadanie domowe jest uwzględniana tylko jeśli uzyska się przynajmniej 1,0 pkt za rozwiązanie zadań podanych na zajęciach!

W żadnym z zadań nie można używać std::string!

Zadanie 1 [1,0 pkt]:

- Dla klasy **Paczka**, zdefiniowanej w pliku „klasy.h” zdefiniuj konstruktor argumentowy, pozwalający na ustawienie wszystkich pól na zadane wartości (przesłane w jego argumentach). Pola, które nie wymagają sprawdzenia ograniczeń, ustaw przy pomocy listy inicjalizacyjnej.

Wskazówki: Zauważ, że jedno z pól klasy to 3-elementowa tablica.

- Zdefiniuj konstruktor domyślny, ustawiający wszystkie liczbowe informacje w klasie na 1.0, adres na pusty c-string i typ paczki n „zwykły”. Zastosuj przynajmniej 2 różne sposoby na zainicjalizowanie pól klasy.

Wskazówki: Zauważ, że podają poprawne wartości domyślne pól, nie musimy korzystać z seterów. Nie zadawaj wartości polom w ciele konstruktora, to jest nieefektywne.

- Zdefiniuj konstruktor 1-argumnetowy, który ustawi wszystkie pola poza typem paczki na takie same wartości domyślne jak konstruktor domyślny.

Wskazówki: To zadanie da się wykonać bez pisania nowego konstruktora, zastanów się jak to zrobić, z użyciem wartości domyślnych argumentów funkcji.

■ Rozbuduj klasę o możliwość zliczania instancji paczek, których waga przekracza 10.0kg (przyjmij, że pole m_waga przechowuje wagę w kg). Zliczaj tylko paczki spełniające kryterium i zadbaj o to aby zmiana wagi dla paczki już istniejącej był aktualizowała adekwatnie licznik. Dodaj do klasy możliwość odczytu liczby paczek o wadze powyżej 10 kg.

Wskazówki: Zauważ, że zliczanie jest tu warunkowe. Pamiętaj, żeby obniżać licznik gdy instancja paczki spełniającej kryterium, przestanie istnieć. Zastanów się jak zmodyfikować seter wagi. Zastanów się, czy każdy konstruktor powinien modyfikować licznik.

Ocenianie:

Konstruktor argumentowy:	0,2 pkt.
Konstruktor domyślny:	0,2 pkt.
Konstruktor 1-argumnetowy:	0,2 pkt.
Zliczanie paczek (>=10kg):	0,2 pkt.
Ogólna jakość kodu:	0,2 pkt.

Zadanie 2 [1,0 pkt]:

■ Dla klasy **Nagroda** zdefiniowanej w pliku „klasy.h” zdefiniuj konstruktor argumentowy, pozwalający na ustawienie wszystkich pól na zadane wartości (przesłane w jego argumentach) z wyjątkiem pola **m_czyOdebrana** (to pole zawsze powinno być ustawiane na false przy tworzeniu nowej instancji). Zastosuj przynajmniej 2 różne sposoby na zainicjalizowanie pól klasy.

Wskazówki: Zauważ, że dwa pola to wskaźniki i tylko jedno pole w klasie ma zdefiniowane ograniczenie na wartość. Jeśli jakieś pole zawsze ma mieć wartość domyślną to nie dawaj zewnętrznej możliwości zadania jego wartości w konstruktorze.

■ Zdefiniuj dla klasy konstruktor domyślny, który ustawi oba wskaźniki na nullptr, wysokość nagrody na 100, typ na „okolicznościowy”.

Wskazówki: Zauważ że ostanie pole zawsze ma być w stanie false przy tworzeniu instancji, dlatego nie podajemy go w specyfikacji konstruktora domyślnego.

■ Zdefiniuj dla klasy konstruktor 2-argumnetowy, pozwalający tylko ustawić fundatora i beneficjenta. Pozostałe pola powinny być ustawione jak w konstruktorze domyślnym.

Wskazówki: To zadanie da się wykonać bez pisania nowego konstruktora, zastanów się jak to zrobić, z użyciem wartości domyślnych argumentów funkcji.

■ Rozbuduj klasę o możliwość zliczania wszystkich nagród, które nie są nagrodami okolicznościowymi. Zliczaj tylko inne nagrody i zadbaj o to, aby zmiana typu dla nagrody już istniejącej był aktualizowała adekwatnie licznik. Dodaj do klasy możliwość odczytu liczby nagród innych niż okolicznościowe.

Wskazówki: Zauważ, że zliczanie jest tu warunkowe. Pamiętaj, żeby obniżać licznik gdy instancja paczki spełniającej kryterium, przestanie istnieć. Zastanów się jak zmodyfikować seter typu. Zastanów się, czy każdy konstruktor powinien modyfikować licznik.

Ocenianie:

Konstruktor argumentowy:	0,2 pkt.
Konstruktor domyślny:	0,2 pkt.
Konstruktor 2-argumnetowy:	0,2 pkt.
Zliczanie nagród:	0,2 pkt.
Ogólna jakość kodu:	0,2 pkt.

Zadanie 3 [2,0 pkt]:

Opracowanie: dr inż. Łukasz Maliński i mgr inż. Błażej Nycz, Politechnika Śląska, 2022

Uwaga: to zadanie będzie dalej rozwijane (w ramach zadania ambitnego) na tych samych zajęciach!

■ Rozbuduj klasę **Point** zdefiniowaną w pliku „klasy.h” o możliwość współdzielenia informacji o położeniu wszystkich instancji punktów w programie. Przyjmij, że program ma górny limit 1000 instancji.

Wskazówki: Zastanów się co trzeba mieć, aby znać informację o położeniu instancji w programie. Następnie zastanów się co zrobić by móc takich informacji przechować 1000 na raz i jak zrobić, aby każda instancja klasy miała do nich dostęp. Przypomnij sobie co należy zrobić aby oznaczyć, że informacja o płożeniu nie jest aktualnie przypisana.

■ Rozbuduj klasę o możliwość zadania rozmiarów ekranu. Rozmiary ekranu powinny być wspólne dla wszystkich instancji klasy i domyślnie być ustawione na 1920x1080. Daj możliwość ustawiania obu wymiarów jednym składnikiem funkcjonalnym klasy tak aby oba były liczbą naturalną parzystą. Wszystkie punkty, które po zmianie wymiarów, znajdą się poza ekranem, powinny zostać przesunięte tak aby znalazły się w granicach ekranu. Odczytywanie wymiarów nie jest konieczne.

Wskazówki: Zastanów się jakiego typu powinny być pola przechowujące wymiary ekranu i jak zrobić aby dotyczyły wszystkich instancji jednocześnie. Zastanów się co zrobić, gdy ktoś poda wymiary ekranu niezgodne ze specyfikacją.

■ Zmodyfikuj setery pól współrzędnych punktu, które pozwolą na zadanie współrzędnych tylko w granicach ekranu. Przyjmij, że lewy górny róg to piksel (0,0), a prawy dolny ma współrzędne wynikające z zdanych wymiarów ekranu dla klasy **Point**.

Wskazówki: Zauważ, że trzeba tu tak przerobić setery, aby uwzględniały zmienne ograniczenia. Progi ograniczeń są przechowywane w klasie i mogą się zmieniać. Seter powinien nie pozwolić zmienić współrzędnych tak, aby wychodziły poza ekran. Pomyśl, jakie powinny być współrzędne dolnego prawego rogu, jeśli lewy górny to (0,0), a wymiary ekranu to po prostu liczba pikseli w poziomie i pionie.

■ Zdefiniuj dla klasy konstruktor argumentowy, pozwalający na ustawienie wszystkich pól na zadane wartości i konstruktor domyślny, ustawiający współrzędne punktu zawsze na środku ekranu i symbol na znak NUL. Znak NUL oznacza brak przypisanego symbolu. Zdefiniuj też destruktor i zastanów się co powinno się w nim znaleźć, gdyż na pewno nie powinien pozostać pusty.

Wskazówki: Zauważ, że tym razem konstruktor domyślny nie polega na wartościach zadanych na sztywno w czasie kompilacji, gdyż zawsze powinien dopasować współrzędne do aktualnych rozmiarów ekranu. Zauważ, że konstruktor tworzy nową instancję, zatem powinien też jakoś uzupełnić współdzieloną informację o położeniu wszystkich instancji punktów w programie. Destruktor natomiast powinien, zrobić coś dokładnie odwrotnego.

■ Zaprezentuj działanie klasy w programie głównym. Utwórz w tym celu kilka punktów, a następnie tak zmień rozmiary ekranu, aby część z nich musiała zostać przesunięta. Pokaż, że zaszło przesuniecie.

Wskazówki: Warto jest napisać sobie prosty operator <<, który wyświetli symbol i współrzędne punku. Pomocne też będzie wyświetlanie przynajmniej kilku pozycji danych przechowujących położenia instancji.

Ocenianie:

Przechowywanie położenia instancji:	0,6 pkt.
Wymiary ekranu w klasie:	0,4 pkt.
Modyfikacja seterów:	0,2 pkt.
Konstruktory i destruktor:	0,4 pkt.
Prezentacja w programie głównym:	0,2 pkt.
Ogólna jakość kodu:	0,2 pkt.