Zajęcia Wyrównawcze – C/C++

Klasy

Klasa w C++ to definicja własnego typu danych, który łączy **dane (pola)** oraz **funkcje operujące na tych danych (metody)** w jedną logiczną całość. Pozwala opisać obiekty występujące w programie w sposób zbliżony do rzeczywistości.

Najważniejsze cechy klasy:

- **Hermetyzacja** – możliwość ukrywania szczegółów implementacji (np. pola prywatne) i udostępniania tylko wybranych funkcji.
- **Tworzenie obiektów** – klasa jest "szablonem", z którego można tworzyć wiele egzemplarzy (obiektów), każdy z własnym zestawem danych.
- **Konstruktory i destruktory** – specjalne funkcje do inicjalizacji i porządkowania obiektów.
- **Możliwość stosowania dziedziczenia i polimorfizmu** – fundament programowania obiektowego.

Krótko: klasa w C++ pozwala zaprojektować własny typ danych z zachowaniem i stanem, co ułatwia pisanie czytelnych i modularnych programów.

Enkapsulacja

Enkapsulacja (ang. encapsulation) to jedna z podstawowych zasad programowania obiektowego. Polega na ukrywaniu wewnętrznej budowy obiektu (czyli danych i sposobu ich przetwarzania) przed światem zewnętrznym oraz udostępnianiu tylko ściśle określonego interfejsu — zwykle w postaci metod publicznych.

W praktyce oznacza to, że:

- dane obiektu (pola klasy) są prywatne (private),
- dostęp do nich odbywa się tylko przez publiczne metody (public), które kontrolują sposób odczytu i modyfikacji danych.

Dynamiczna alokacja pamięci, czyli działanie na wskaźnikach

Dynamiczna alokacja pamięci w C++ służy do tworzenia obiektów i tablic w trakcie działania programu, kiedy ich rozmiar lub czas życia nie są znane w chwili kompilacji.

Najważniejsze zastosowania:

- Tworzenie struktur o zmiennym rozmiarze – np. tablic, których długość wprowadza użytkownik.
- Kontrola czasu życia obiektów – obiekt istnieje tak długo, jak długo jest potrzebny; można go utworzyć i usunąć w dowolnym momencie.
- Efektywne zarządzanie zasobami – szczególnie w dużych projektach i strukturach danych (np. drzewa, listy, grafy), które wymagają tworzenia elementów "na żądanie".

Do dynamicznej alokacji używa się:

- new / delete – do tworzenia i usuwania pojedynczych obiektów.
- new[] / delete[] – do tworzenia i usuwania tablic.
- Nowocześnie preferuje się także inteligentne wskaźniki (np. std::unique_ptr, std::shared_ptr), które automatyzują zwalnianie pamięci.

Krótko: dynamiczna alokacja pozwala elastycznie zarządzać pamięcią i tworzyć struktury danych dostosowane do potrzeb programu w czasie jego działania.

1) Deklaracje wskaźników
   a. Deklaracja wskaźnika: `int *p`
   b. Przypisanie pamięci do wskaźnika: `p = new int;`
   c. Modyfikacja zmiennej przez wskaźnik: `*p = 3`
   d. Inicjalizacja przez inną zmianną: `p = &a` (adres zmiennej a to: `&a`)
   e. Dereferencja (przypisanie wartości na którą pokazuje wskaźnik): `a2 = *p`

2) Dynamiczna alokacja pamięci

```
// tablica jednowymiarowa int
int *tab = new int * [10];


// tablica dwuwymiarowa int
int **tab2 = new int * [5];
for (int i = 0; i<5; i++){
     tab2[i] = new int[10];

}
```

Konstruktory

Konstruktor to specjalna metoda w klasie, która uruchamia się automatycznie w momencie tworzenia obiektu. Najważniejsze cechy konstruktora:

- Nazywa się tak samo jak klasa

W C++ klasa Book ma konstruktor Book().

- Nie zwraca żadnego typu, nawet void

Dzięki temu kompilator wie, że to konstruktor, a nie zwykła metoda.

- Służy do inicjalizacji obiektu

Ustawia wartości pól, rezerwuje zasoby, przygotowuje obiekt do użycia. To idealne miejsce, aby nadać polom wartości startowe albo wywołać inne metody inicjalizujące.

- Może być przeciążany

Możesz mieć kilka konstruktorów z różnymi parametrami — np. jeden pusty, a drugi ustawiający wartości pól.

Konstruktor kopiujący to specjalny konstruktor w C++, który służy do tworzenia nowego obiektu na podstawie już istniejącego obiektu tej samej klasy.

Jego ogólna postać:

```
ClassName(const ClassName& other);
```

Szczególnie ważny jest w klasach, które zarządzają zasobami, takimi jak:

- pamięć dynamiczna (new/delete),
- uchwyty do plików,
- połączenia sieciowe,
- wskaźniki na tablice, struktury itp.

Domyślny konstruktor kopiujący wykonuje płytką kopię (shallow copy):

- kopiuje wartości pól bit po bicie,

co oznacza, że wskaźniki będą wskazywać na tę samą pamięć.

Pola statyczne i metody statyczne

W języku C++ oprócz zwykłych pól i metod obiektowych możemy w klasie zdefiniować pola statyczne oraz metody statyczne. Są to elementy, które nie należą do konkretnego obiektu, lecz do samej klasy. Oznacza to, że:

- wszystkie obiekty danej klasy współdzielą jedną kopię pola statycznego,
- pole statyczne istnieje, nawet jeśli nie utworzono jeszcze żadnego obiektu,
- metodę statyczną można wywołać bez tworzenia obiektu, używając nazwy klasy.

Najczęstsze zastosowania pól statycznych to:

- zliczanie liczby utworzonych obiektów,
- przechowywanie ustawień wspólnych dla całej klasy,
- przechowywanie wartości globalnych logicznie powiązanych z daną klasą.

Metody statyczne natomiast:

- pozwalają wykonywać operacje związane z klasą jako całością (np. wypisywać stan pól statycznych),
- nie mają dostępu do pól obiektów (bo nie działają na obiekcie),
- mogą być wywoływane bez instancji.

Funkcje zaprzyjaźnione

Funkcje zaprzyjaźnione - funkcje, które mimo, że nie są składnikami klasy maja dostęp do wszystkich (nawet prywatnych i chronionych) składników klasy.

To nie funkcja ma twierdzić, ze przyjaźni sie z klasa, ale sama klasa daje jej dostęp do swoich prywatnych i chronionych składników.

- Funkcja zaprzyjaźniona może być przyjacielem więcej niż jednej klasy!

- Jeśli klasa deklaruje przyjaźń ze wszystkimi funkcjami innej klasy, możemy mieć klasę zaprzyjaźnioną.

- słowo kluczowe: friend

Deklarację zaprzyjaźnienia umieszcza się wewnątrz definicji klasy (w pliku .h). Zaprzyjaźnienie dla funkcji globalnej: void WypiszKlase(Klasa A) { … } będzie wyglądać:

`void WypiszKlase(Klasa A);` i należy je umieścić w definicji klasy `Klasa`

Operatory

Przeciążanie operatorów w C++ to mechanizm, który pozwala zdefiniować działanie standardowych operatorów (np. +, ==, [], <<) dla własnych typów danych – najczęściej klas i struktur.
Po co to robimy? Żeby obiekty naszych klas zachowywały się naturalnie i czytelnie, tak jak typy wbudowane. Np. jeśli mamy klasę Wektor2D, to dzięki przeciążeniu operatora + możemy pisać:

```
Wektor2D a, b, c;
c = a + b;
```

Zamiast wywoływać specjalne funkcje typu

```
c = add(a, b);
```

Ułatwia to korzystanie z klasy i poprawia czytelność kodu.

Operator strumieniowy – wyjścia <<

Aby przeciążyć operator<< (tylko jako funkcję globalną) deklarujemy funkcję:

```
ostream& operator<<(ostream& wyjscie, punkt& pp)
{
      Wyjście<<" ….";  //w miejscu kropek wypisujemy wszystko co chcielibyśmy
      żeby było wypisywane
      return wyjscie;
}
```

Jeśli współrzędne punktu są polami prywatnymi w klasie, to funkcję operatorową należy z klasą zaprzyjaźnić.

Operatory +, -, *, /

Mogą być przeładowane w wersji jedno- [wewnątrz klasy] lub dwuargumentowej [na zewnątrz klasy]. Tak jak w przypadku każdej innej funkcji nie istnieją ograniczenia co do zwracanych wartości: operator taki może zwracać obiekt tej samej klasy, którą dodaje (np. możemy napisać klasę która dodając do siebie dwa obiekty klasy człowiek zwróci również człowieka), lub dowolny inny, jaki sobie wymyślimy (dodając do siebie dwóch ludzi zwróci nam np. sumę ich wieku).

Operator przypisania =

Dwuargumentowy operator przypisania = służy do przypisanie jednemu obiektowy klasy treści drugiego obiektu tej samej klasy.

Jeśli operator= nie zostanie zdefiniowany, to zostanie on automatycznie wygenerowany (podobnie jak konstruktor kopiujący!), przepisane zostaną pola “składnik po składniku” (podobnie jak w przypadku automatycznie generowanego konstruktora kopiującego). W rezultacie takiego przypisania będą dwa obiekty o bliźniaczej treści.

Kłopoty natomiast mogą pojawić się wtedy, kiedy składnikami klasy są wskaźniki lub jeśli klasa używa operatora new do rezerwacji miejsca w zapasie pamięci.

Operator przypisania składa się z części:
(1) sprawdzenie, czy nie jest kopiowane siebie samego
(2) części "destruktorowej" (np. zwalnianie pamięci)
(3) części "konstruktorowej", przypominającej konstruktor kopiujący

Przykład użycia: (jeśli operator znajduje się wewnątrz klasy):

```
Nazwa & operator=(Nazwa & na){;
 if (&na == this) return *this; //(1) sprawdzenie, czy nie kopiuje samego siebie

 delete X; //(2) zwalniamy pamięć dla rzeczy, które mogły mieć poprzednio zaalokowaną pamięć

 //(3)alokujemy nową pamięć i przepisujemy wartości tak samo jak w konstruktorze kopiującym
    return *this;
 }
```

Operator [ ]

Operator [ ] to operator odwołania się do elementu tablicy, jest dwuargumentowy. Chcąc napisać funkcję operatorową [ ], tak, aby operator [ ] mógł stać po obu stronach znaku = musimy zadeklarować, że funkcja ta będzie zwracała referencję do tego, czemu mamy przypisywać!

Przykład implementacji operatora [ ]:

```
int& operator[](int i) { return a[i]; }
```