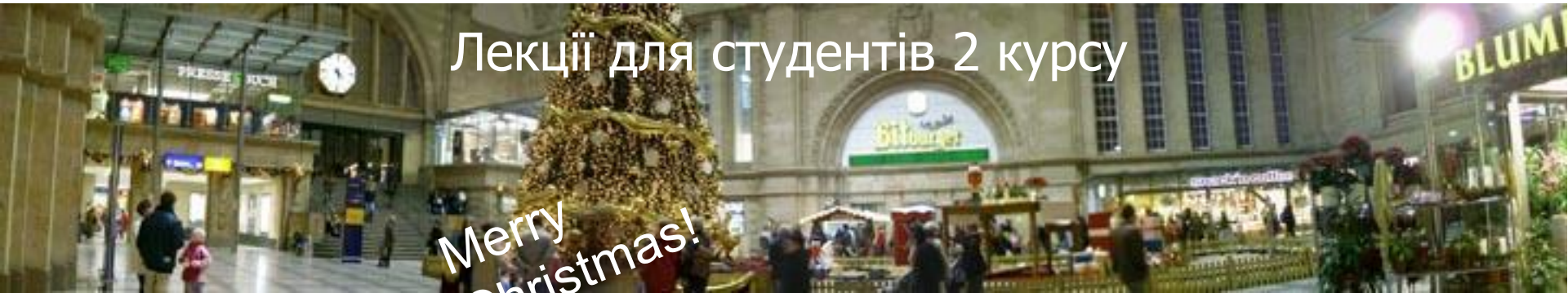
Бублик Володимир Васильович

Об'єктно-орієнтоване програмування

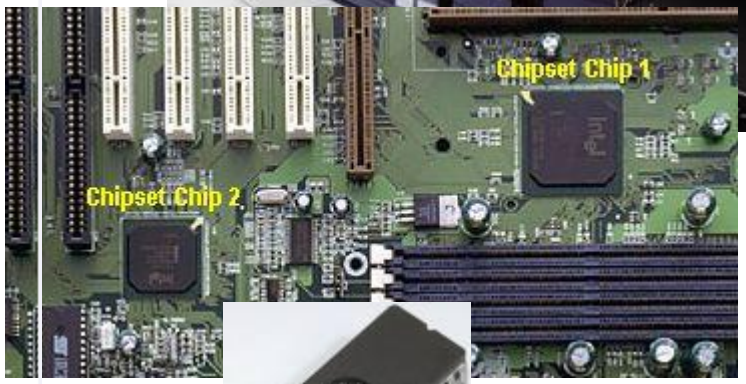
Частина 1. Об'єктне програмування.

Лекція 1. Принцип інкапсуляції

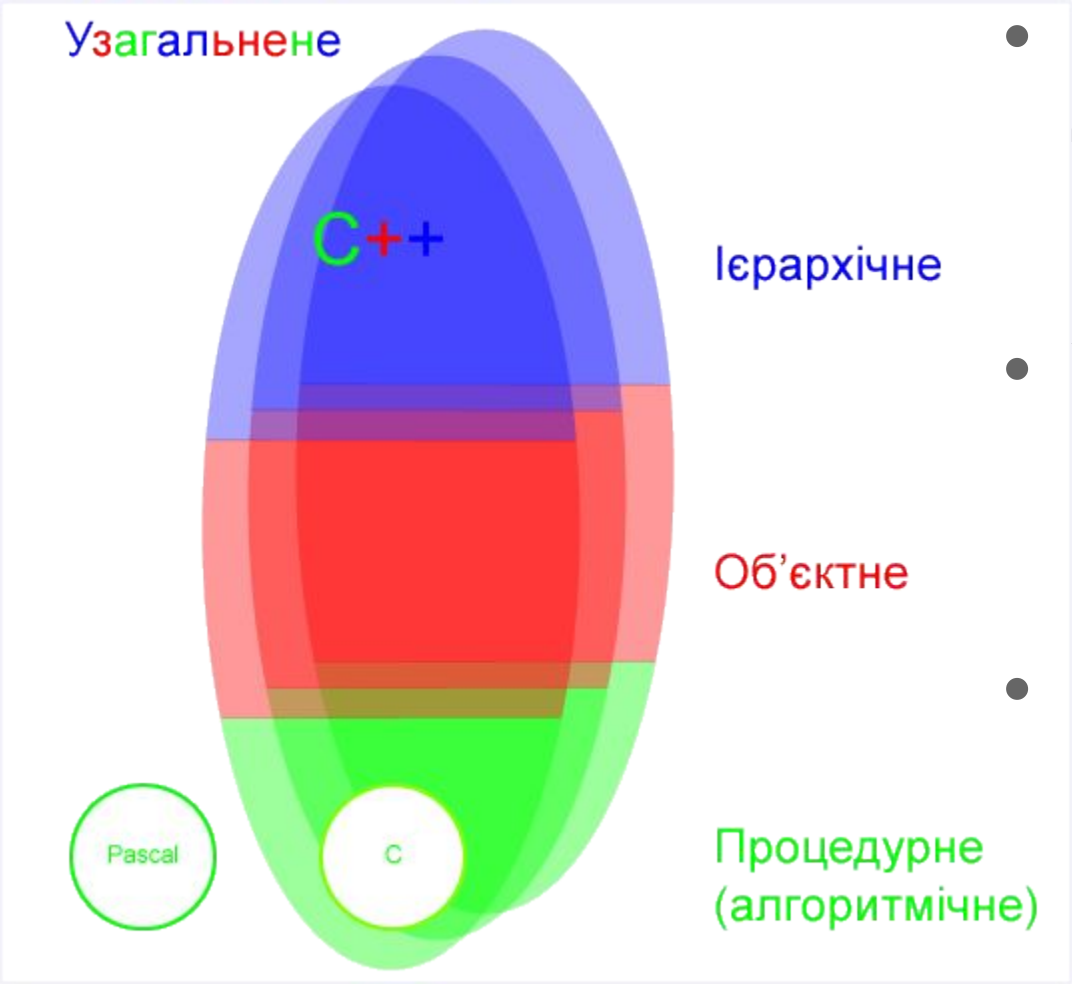


Парадигми програмування (повторення)

Мультипарадигменна мова



Погляд в майбутнє



- Програмувати, думаючи про *нові застосування* (*reuse*)
- Узагальнене програмування дає нові застосування програмних текстів
- Об'єктне і ієрархічне – нові застосування об'єктних кодів (об'єктів і класів)

Інкапсуляція

Під інкапсуляцією розумітимемо спосіб збирання певних елементів в одне ціле з метою утворення нової сутності (або абстракції нового рівня)

1. Команди інкапсульовані в функцію
2. Поля інкапсульовані в структуру

```
struct Point
{
 double _x;
 double _y;
}
```

Дані і функції в структурах (як С моделює С++)

3. Чи можна інкапсулювати функцію в структуру?

Так, можна за допомогою указника функції

QuasiPoint	
double:	_x
double:	_y
doubble (*_f)(QuasiPoint, QuasiPoint)	

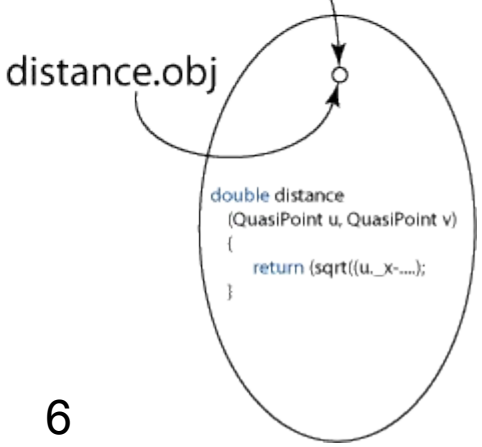
```
struct QuasiPoint
{
    double _x;
    double _y;
// інкапсуляція указника на функцію
    double (*f)(QuasiPoint, QuasiPoint);
};
```

Як викликати f?

Ініціалізація та виклик інкапсульованої функції

- double distance (QuasiPoint, QuasiPoint);
-

QuasiPoint	u	
double:	_x	0
double:	_y	1
doubble (*_f)(QuasiPoint, QuasiPoint)		

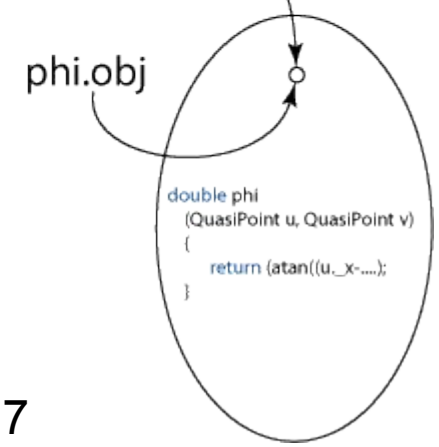


- QuasiPoint u={0, 1, distance};
- cout<<u.f(u,v)<<endl;

Ініціалізація та виклик інкапсульованої функції

- double phi (QuasiPoint, QuasiPoint);

QuasiPoint	v	
double:	_x	1
double:	_y	1
doubble (*_f)(QuasiPoint, QuasiPoint)		



-
- QuasiPoint v={1, 1, phi};
- cout<<v.f(u,v)<<endl;

Інкапсуляція в об'єкті

Це добре чи зле, що в одній і тій же структурі функція-член структури в різних екземплярах позначає різні дії?

u.f(a,b) відстань від a до b

v.f(a,b) кут між векторами 0a і 0b

Як засобами C зробити так, щоб для всіх екземплярів QuasiPoint указник показував завжди одну й ту ж функцію?

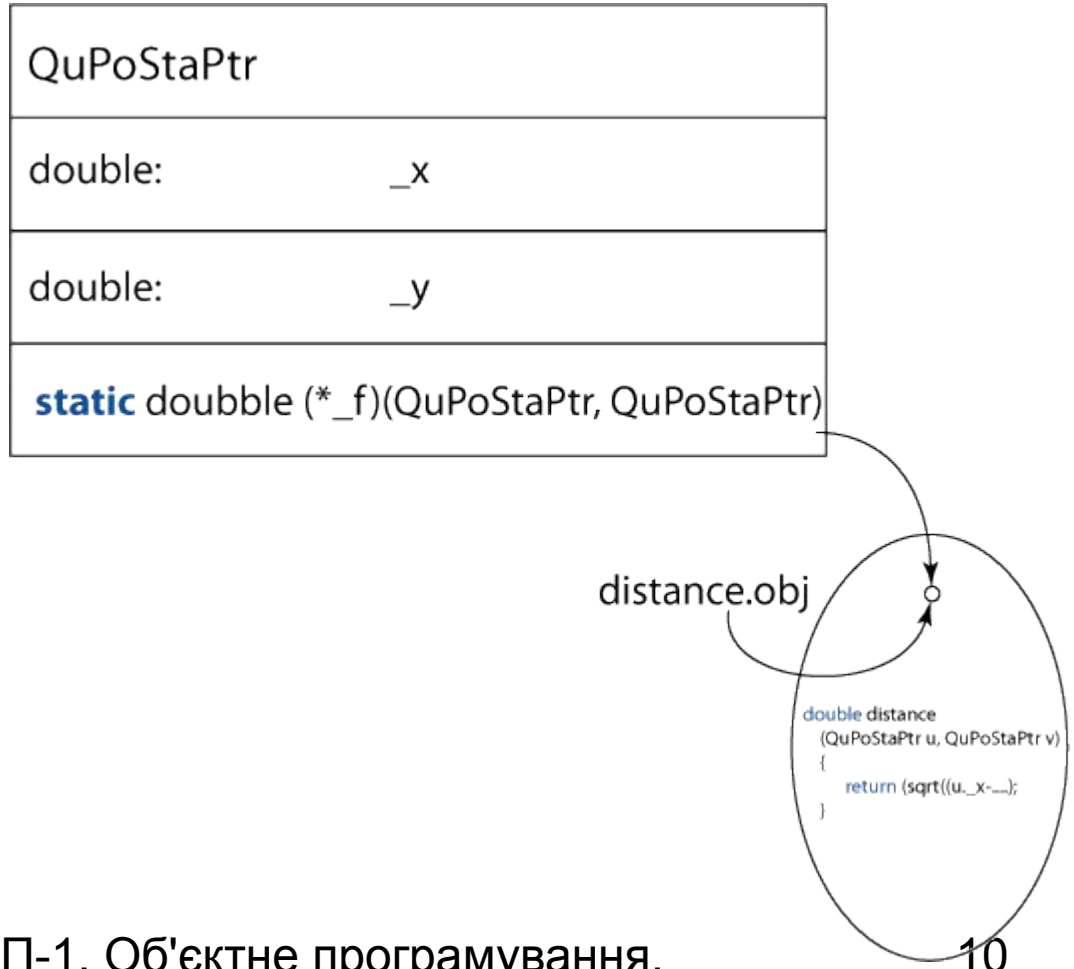
Статичний указник на функцію

```
struct QuPoStaPtr
{
 double _x;
 double _y;
//    інкапсуляція указника на функцію в класі
 static double (*f)(QuPoStaPtr, QuPoStaPtr);
};

//    Один указник для всіх об’єктів
double (* QuPoStaPtr ::f)
            (QuPoStaPtr, QuPoStaPtr) = distance;
cout<< QuPoStaPtr::f(u,v)<<endl;
```

Інкапсуляція указника на функцію в класі

- Діаграма класу, а не об'єкта



Створення і видалення екземпляру структури

```
struct PreWrappedVector
{
 static const int _n;
 double * _v;
};

//   Функція створення вектора a._v
void construct (PreWrappedVector& a);

//   Функція видалення вектора a._v
void destroy (PreWrappedVector& a);
```

Створення і видалення екземпляру структури

- // Типовий сценарій обробки
- PreWrappedVector v;

- // Хто гарантує наявність конструктора і
- // створення вектора до першого вживання?
- construct(v);
- .....
- destroy(v);
- // Хто гарантує наявність деструктора і видалення
- // вектора після завершення його обробки?

Конструктор і деструктор

```
struct WrappedVector
{
 static const int _n;
 double * _v;
//    Це конструктор, він створює об'єкт
//    при його визначенні
 WrappedVector();
//    Це деструктор, він автоматично видаляє об'єкт
 ~WrappedVector();
};
```

Реалізація конструктора

```
WrappedVector::WrappedVector()
{
 cout<<"Constructor WrappedVector"<<endl;
//    1. Виділити пам'ять
 _v = new double[_n];
//    2. Можливо обробити аварійну ситуацію
//    3. Ініціалізувати масив
 for (int i=0; i<_n; i++)
        _v[i] = 0;
 return;
}
```

Реалізація деструктора

```
WrappedVector::~WrappedVector()
{
 cout<<"Destructor WrappedVector"<<endl;
 delete [] _v;
//    Чи варто обнулити указник _v?
 return;
}
```

Головне правило об'єктного програмування

- Кожній структурі надаються конструктор і деструктор

Автоматичний виклик конструктора і деструктора

```
int main()
{
//    Визначення об'єктів приводить
//    до виклику конструкторів.
//    Ось він:
 WrappedVector w1, w2;
.................................................
//    Життя об'єктів завжди закінчується
//    автоматичним викликом їх деструкторів
//    ост тут:
 return 0;
}
```

Хто викликає конструктор і деструктор?

Це робить система програмування

```
// new: конструктор; delete: деструктор
int main()
{
 WrappedVector *pw = new WrappedVector;
//    Створення об'єкту: неявний виклик конструктора
.....................................................
//    Видалення об'єкту: неявний виклик деструктора
 delete pw;
 return 0;
}
```

Конструктор і деструктор за замовчуванням

Чи мала б структура PreWrappedVector конструктора і деструктора, якщо їх не визначити явно?

```
struct PreWrappedVector
{
 static const int _n;
 double * _v;
};
```

Конструктор і деструктор за замовчуванням

Так! Компілятор генерує порожні конструктор і деструктор для кожної структури, яка не має власних

```
// Конструктор за замовчуванням
PreWrappedVector:: PreWrappedVector(){ };

// Деструктор за замовчуванням
PreWrappedVector:: ~PreWrappedVector(){ };

// Краще б їх не було, але так досягається
// сумісність C і C++
```

Друге правило об'єктного програмування

- Ніколи не користується конструкторами за замовчуванням, згенерованими системою програмування

- Чому?

- ВВ поставить двійку
- Визначивши власні конструктор і деструктор ви повністю контролюєте створення і видалення ваших об'єктів, а не передоручаєте це комусь (віддаєте дітей в дитячий будинок)

Дані-члени структур (атрибути) і функції(методи)

```
• struct Point
• {
//  Атрибути
•   double _x, _y;
//  Методи
•   Point (double x=0, double y=0): _x(x),_y(y) { };
•   ~Point(){ };
//  Функції доступу до атрибутів
•   double& x()   {return _x;}
•   double& y()   {return _y;}
• };
```

Виклик конструктора з параметрами

// Замість

- Point a = Point(1,2);

// пишемо скорочено

- Point a1(1,2);
-
- Point b = Point(1);
- Point b1(1);
-
- Point c = Point();
- Point c1;

Навіщо потрібні функції доступу?

- Для того щоб контролювати всі випадки використання атрибутів у кожному об'єкті

Виклик методів

Виклик методів відрізняється від виклику звичайних функцій

// Застосувати до екземпляру **a** структури Point
// функцію **x()**

- **a.x()** = 10;
- cout<<**a.x()**<<endl;

Варіант функцій доступу: утиліти

```
• struct Point
• {
//  Атрибути
•   double _x, _y;
//  Конструктор
•   Point (double x=0, double y=0): _x(x),_y(y) { };
//  Деструктор
•   ~Point(){};
• }
//  Утиліти доступу до атрибутів
•   double& x(Point & a)    {return a._x;}
•   double& y(Point & a)    {return a._y;}
```

Виклик утиліти

Виклик утиліт є звичайним викликом функцій

// Передати до функції x() параметр a

- **x(a)** = 10;
- cout<<**x(a)**<<endl;

Прямий доступ

Замість функкції x(a) або методу a.x() можна було б напряму звертатися до члена структури _x

- **a._x** = 10;
- cout<<**a._x**<<endl;

Для чого потрібні методи доступу?

```
struct Point
{
private:   //закрили прямий доступ до атрибутів
 double _x;
 double _y;
public:    //відкрили доступ до методів
 Point (double x=0, double y=0): _x(x),_y(y) { };
 ~Point(){};
 double& x()  {return _x;}
 double& y()  {return _y;}
};
```

Права доступу

Як і раніше, кожен, хто бачить об'єкт може скористатися його відкритим методом

// Застосувати до екземпляру **a** структури Point
// відкритий метод **x()**

- **a.x()** = 10;
- cout<<**a.x()**<<endl;

Права доступу

Сам метод, завдяки своїй належності до структури сам бачить її закриту частину

// Як і раніше, ім'я **_x** в тексті функції

- double& x() {return **_x**;}

// позначає поле **_x** того екземпляру,

// до якого застосовано функцію

Права доступу

Але для сторонніх атрибути стали невидимими

```
int main ()
{
  Point a;
  a._x = 10;
  cout<<a._x<<endl;
  return 0;
}
double& x(Point & a){return a._x;}
double& y(Point & a){return a._y;}
```

Клас

```
class Point
{
private:   //закрита частина класу
 double _x;
 double _y;
public:    //відкрита частина класу
 Point (double x=0, double y=0): _x(x),_y(y) { };
 ~Point(){};
 double& x()  {return _x;}
 double& y()  {return _y;}
};
```

Структури і класи

Структуру, яку поділено на **відкриту і закриту частини** називатимемо **класом**.

Структуру розглядатимемо як архаїзм від C. В структурі все звичайно вважається відкритим.

Екземпляри структур і класів називатимемо **об'єктами**. Кожен об'єкт створюється в результаті виклику конструктора, а видаляється деструктором.

Структура може не мати власних конструктора і деструктора. Ваші класи завжди повинні мати власні конструктори (можливо декілька) і деструктор (не забудьте про ВВ).

Клас vs. структура

Правила доступу — це поділ класу на відкриту (public) і закриту (private) частини. Закрита частина доступна лише з середини класу, відкрита — звідусіль.

Формально клас відрізняється від структури лише правилом замовчування прав доступу:

1. Все, що явно не відкрите в класі, вважається закритим
2. Все, що явно не закрите в структурі, вважається відкритим

Структури звичайно вживають як сукупність даних, класи — як сукупність даних (атрибутів) і функцій-членів класу (методів)

Повторення. Два способи запису ініціалізації

- double x = 1.0;
- double y = x;
- double u = 2.0;

// але можна й так

- double v(u);

// це те ж саме, що

// double v = u;

Ініціалізація атрибутів в конструкторі

```
class Complex
{
private:
  double _re;
  double _im;
public:
  Complex (double re, double im):
//  ініціалізація атрибутів (добра!)
        _re(re), _im(im) { };
}
```

Ініціалізація атрибутів в конструкторі

```
class Complex
{
private:
  double _re;
  double _im;
public:
  Complex (double re, double im):
//  ініціалізація атрибутів (погана!)
  {
      _re = re; _im = im;
  };
```

Приклад 1. Person.h

```
class Person
{
private:
  const int _len;
  char * _name;
public:
//  реалізація винесена в cpp-файл
  Person(int, char []);
  ~Person();
};
```

Приклад 1. Person.cpp (конструктор)

```
• Person::Person (int len, char name[]):
•  _len(len),
•  _name (new char[_len+1]);
• {
•  for (int i=0; i<_len; i++)
•      _name[i] = name[i];
•  _name[_len]='\0';
•  cout<<"A person "<<_name<<" was created"<<endl;
•  return;
• }
```

Приклад 1. Person.cpp (деструктор)

```
• Person::~Person()
• {
•  cout<<"A person "<<_name<<" was deleted"<<endl;
•  delete [] _name;
//  _name = 0; зайве, оскільки сам об'єкт,
//  а значить і його атрибут _name,
//  припиняють своє існування
•  return;
• }
```

Приклад 2. WrappedVector.h

```
class WrappedVector
{
private:
 static const int _n;
 double * _v;
public:
 WrappedVector();
 ~WrappedVector();
};
```

Приклад 2. WrappedVector.cpp

```
const int WrappedVector::_n = 100;
WrappedVector::WrappedVector():
 _v (new double[_n];)
{
 cout<<"Constructor WrappedVector"<<endl;

 for (int i=0; i<_n; i++)
      _v[i] = 0;
 return;
}
```

Приклад 2. WrappedVector.cpp

```
WrappedVector::~WrappedVector()
{
 cout<<"Destructor WrappedVector"<<endl;
 delete [] _v;
 return;
}
```

Селектори і модифікатори

Як добратися до атрибутів, якщо вони закриті? — За допомогою методів доступу: селекторів і модифікаторів

1. Два в одному

- double& x() { return **_x**;}

2a. Селектор

- double getX() { return **_x**; };

2b. Модифікатор

- void setX (double x) { _x = x;}

Приклад 2. WrappedVector. Селектор-модифікатор

```
class WrappedVector
{
private:
 static const int _n;
 double * _v;
public:
 class BadIndex { };
 double& getSet (int i);
 WrappedVector();
 ~WrappedVector();
};
```

Приклад 2. WrappedVector. Селектор-модифікатор

```
• double& WrappedVector::getSet (int i)
• {
•  if ((i<0) !! (i>=_len))
•       throw BadIndex;
•  return _v[i];
• }

• WrappedVector u;
• u.getSet(0) = 500;
• cout<<u.getSet(0)<<endl;
//А хотілося б u[i]. — Далі буде
```

Приклад 2. WrappedVector. Селектор і модифікатор

```
double WrappedVector::get (int i)
{
 if ((i<0) !! (i>=_len)) throw BadIndex;
 return _v[i];
}
void WrappedVector::set (int i, double x)
{
 if ((i<0) !! (i>=_len)) throw BadIndex;
 _v[i] = x;
 return;
}
```

Чому віддавати перевагу

Окремий модифікатор дозволяє контролювати кожну спробу зміни значення атрибуту, а селектор — кожне використання його значення.

Модифікатор-селектор

1. не відрізняє зміну значення від читання;
2. порушує інкапсуляцію (як?)

Але кожна мова програмування пропонує оператор індексування [ ] — по суті селектор-модифікатор.

Що вживати: клас чи структуру?

Слідкуємо за створенням і видаленням об'єктів, регламентуємо доступ до його частин — вживаємо клас.

Обов'язкові конструктор(и) і деструктор, модифікатори і селектори для кожного призначеного для використання зовні атрибуту.

Правило доступу: Атрибути, **як правило**, закриті; методи **можуть бути** відкриті.

В інших випадках можна обходитися структурами

Об'єкт – екземпляр класу

Об'єкт характеризується ідентичністю, станом і поведінкою.

Ідентичність — це властивість, що відрізняє об'єкт від усіх інших об'єктів. Об'єкт набуває ідентичності при створенні його конструктором і втрачає її при видаленні його деструктором.

Стан визначається набором значень атрибутів об'єкту.

Поведінка визначається набором методів.

Висновок

Вивчили

1. Як створити об'єкт в заданому початковому стані
2. Як змінити стан об'єкту
3. Як визначити стан об'єкту
4. Як видалити об'єкт

Наступна задача:

- наділити об'єкти поведінкою