

# ТЕМА: Конструктори і деструктори класів.

## Зміст і будова конструктора

Як правило, поля об'єкта потрібно ініціалізувати перед першим використанням. Наприклад, для попереднього прикладу після визначення об'єкта Stat A,B; не можна зразу виконати метод A.Read(); чи A.Print(); , бо не відомий початковий розмір масиву і не надано місце в пам'яті для елементів масиву.

Для розв'язання цієї проблеми в мові C++ передбачений механізм конструкторів, який дозволяє ініціалізувати об'єкти в момент їх створення.

Конструктор – це особлива функція класу. Ім'я функції співпадає з іменем класу.

```
class Stat // визначення класу
{
    . . . . .
public:
    Stat() // конструктор 1 – визначений всередині класу
    {
        size = 10; // фіксуємо деякий початковий розмір
        mas = new int[size];
        for(int i=0; i<size; i++) mas[i]=0; // поч. знач. =0
        max=ave=0; // потрібно визначати всі поля
    }
    . . . . .
} ;
```



Для мови C++ щодо конструкторів маємо такі правила:

- 1) ім'я конструктора співпадає з іменем класу;
- 2) конструктор не повертає результату;
- 3) може бути перевантажений з різними сигнатурами;
- 4) виконується автоматично в момент створення об'єкта, статичного чи динамічного, і лише один раз при вході у відповідний блок операторів (для глобального – до початку виконання функції `main()` );
- 5) може бути визначений за схемою звичайного методу – всередині класу і окремо поза класом. Якщо поза класом, тоді:

```
class Stat // визначення класу
{
. . . . .
public:
    Stat(); // конструктор 1 – визначений поза класом
. . . . .
} ;

Stat::Stat()
{
    // такий самий текст
}
```

Якщо тепер виконати, наприклад, такий фрагмент програми:

```
void main()
{
    Stat A; // статичний об'єкт,
           // автоматично виконують конструктор
    A.Print();
    A.Read(); A.Print();
    A.Repl(3,-1); A.Print();
    cout << A.GetAve() << " " << A.GetMax() << endl;
    . . . . .
}
```

то побачимо, що методи `Print()` і `Read()` працюють коректно.

Визначивши конструктор, ми автоматично надаємо пам'ять елементам масиву, тому треба корегувати метод `Create()`, щоб звільнити попередню пам'ять:

```
void Stat::Create(int S, int val)
// створення масиву розміру S, значення val
{
    delete [] mas; // за наявності конструктора треба
                  // звільнити попередню пам'ять
    . . . . . // далі так само, як раніше для Create
}
```



## Правило замовчування для визначення конструктора

Конструктор можна не визначати, як це було в найпершому варіанті програми на попередній лекції. В цьому разі компілятор сам будує пустий конструктор

```
Stat() {}
```

який не виконує жодних операцій. Його називають конструктором за замовчуванням.

Якщо ж визначити хоча б один свій конструктор, тоді конструктор за замовчуванням не будують. Це правило потрібно врахувати для створення об'єктів, бо можливі різні варіанти ситуацій.

Висновок: конструктор завжди буде виконаний – свій або пустий.



## Перевантаження конструкторів

У переважній більшості реальних задач буває потреба в різних способах ініціалізації об'єкта в момент його створення. Це можна реалізувати, побудувавши ще один конструктор чи декілька конструкторів. Конструктори можна перевантажувати так само, як звичайні методи.

Наприклад, можна визначити другий конструктор з параметрами:

```
Stat(int dim, int x)  // конструктор 2
{
    size = dim;  // розмір масиву
    mas = new int[size];
    for(int i=0; i<size; i++) mas[i]=x;
                                // x - початкові значення
    max=ave=x; // потрібно визначати всі поля
}
```

Тепер можна будувати об'єкти двома способами:

```
void main()
{
    . . . . .
    Stat C;    // виконують конструктор 1
    . . . . .
    Stat D(6,25);    // виконують конструктор 2
    . . . . .
}
```

Конструктор 2 виконано скороченою формою запису – як показано вище, шляхом запису значень фактичних параметрів.

Друга форма є повною:

```
Stat D = Stat(6,25);
```

---

Рекомендація: завжди явно визначати конструктор 1 без параметрів, а також інші необхідні конструктори.

Це може бути особливо актуальним для майбутньої будови дочірніх класів.



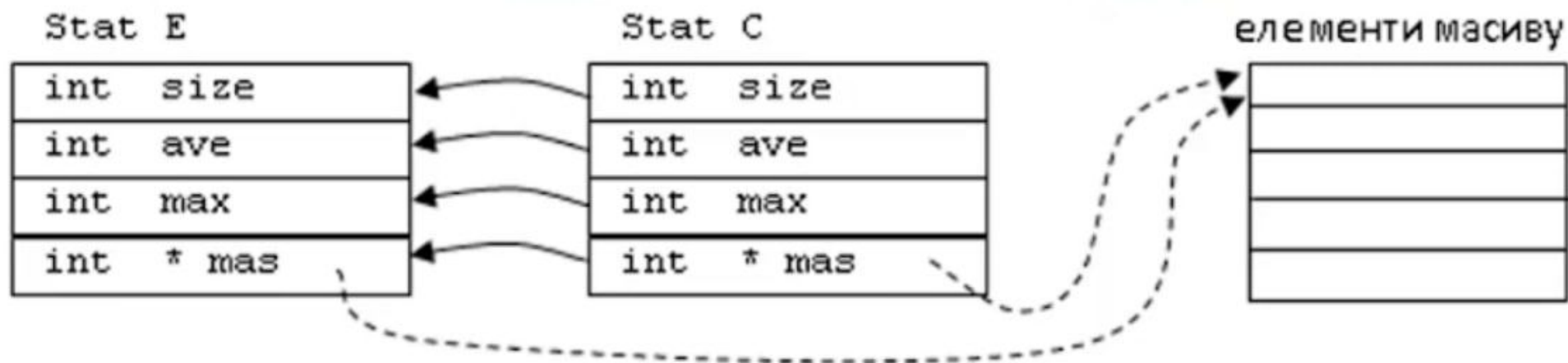


## Конструктор копіювання

Мова C++ дозволяє ініціалізувати один об'єкт іншим. Наприклад:

```
Stat C; // виконують конструктор 1
C.Read(); C.Print();
Stat E(C); // виконують конструктор копіювання
E.Print();
```

В цьому разі виконують так званий конструктор копіювання, який за замовчуванням створюється автоматично для кожного класу. Проте такий конструктор виконує побітову копію лише статичних полів об'єкта:




Отже, об'єкт E буде мати посилання на ту саму ділянку пам'яті, яка була надана об'єкту C для елементів масиву, не маючи своєї.

Очевидно, що це зовсім не те, що нам потрібно.

Для вирішення цієї проблеми потрібно ще раз перевантажити конструктор, побудувавши свій конструктор копіювання. Цей конструктор не буде виконувати побітового копіювання, а явно визначати всі необхідні присвоєння.

Звернути увагу на стандартну форму заголовка конструктора копіювання, параметром якого є посилання на об'єкт-оригінал (знак &) :




```
Stat ( const Stat & ob )    // конструктор копіювання
{
    size = ob.size;  ave = ob.ave;  max = ob.max;
    mas = new int[size];  // надаємо свою ділянку пам'яті

    // і виконуємо копіювання елементів масиву
    for(int i=0; i<size; i++) mas[i]=ob.mas[i];
}
```

Насамкінець підкреслимо, що конструктор копіювання застосовують лише для ініціалізації об'єктів, але не для створення.



## Деструктори

 Деструктор є антиподом конструктора. Це спеціальний метод, який так само автоматично викликають, але в момент припинення існування об'єкта – кінець блоку або оператор delete. Якщо бути точним, деструктор виконують безпосередньо перед знищенням об'єкта.

Ім'я деструктора так само співпадає з іменем класу, але з передуючим знаком тильда. В багатьох випадках об'єкт перед знищенням має виконати деякі заключні операції, наприклад, записати зміст об'єкта у файл, закрити робочі файли, звільнити пам'ять.

Приклад деструктора:

```
~Stat() { delete [] mas; cout << "Object destroyed.\n"; }
```

Викликом деструктора ми не керуємо – на відміну від конструктора. Тому:

- 1) деструктор завжди один;
- 2) параметрів не має;
- 3) результату не повертає.

Для деструктора є аналогічне правило замовчування – якщо не визначати, буде створений порожній без всяких дій.

Місце виклику деструктора:

```
{ // початок деякого блоку
  Stat C; // виконують конструктор 1
  . . . . .
  Stat E(C); // виконують конструктор копіювання
  . . . . .
} // <- тут викликають деструктори автоматично
```

Для випадку глобальних об'єктів деструктори викликають після закінчення функції `main()`.

