Функции. Встраиваемые функции

В базовом языке С директива препроцессора `#define` позволяла использовать макроопределения для записи вызова небольших часто используемых конструкций. Некорректная запись макроопределения может приводить к ошибкам, которые очень трудно найти. Макроопределения не позволяют определять локальные переменные и не выполняют проверки и преобразования аргументов.

Если вместо макроопределения использовать функцию, то это удлиняет объектный код и увеличивает время выполнения программы.

Кроме того, при работе с макроопределениями необходимо тщательно проверять раскрытия макросов:

```
#define SUMMA(a, b) a + b
rez = SUMMA(x, y)*10;
```

После работы препроцессора получим:

```
rez = x + y*10;
```

Функции. Встраиваемые функции

В C++ для определения функции, которая должна встраиваться как макроопределение используется ключевое слово `inline`. Вызов такой функции приводит к встраиванию кода inline-функции в вызывающую программу. Определение такой функции может выглядеть следующим образом:

```
inline double SUMMA(double a, double b)
      {
       return(a + b);
      }
```

При вызове этой функции

```
rez = SUMMA(x,y)*10;
```

будет получен следующий результат:

```
rez=(x+y)*10
```

Функции. Встраиваемые функции

При определении и использовании встраиваемых функций необходимо придерживаться следующих правил:

- Определение и объявление функций должны быть совмещены и располагаться перед первым вызовом встраиваемой функции.
- Имеет смысл определять `inline` только очень небольшие функции, поскольку любая inline-функция увеличивает программный код.
- Различные компиляторы накладывают ограничения на сложность встраиваемых функций. Компилятор сам решает, может ли функция быть встраиваемой. Если функция не может быть встраиваемой, компилятор рассматривает ее как обычную функцию.

Динамическое выделение памяти

В C работать с динамической памятью можно при помощи соответствующих функций распределения памяти (`calloc`, `malloc`, `free`), для чего необходимо подключить библиотеку

```
#include <malloc.h>
```

C++ использует новые методы работы с динамической памятью при помощи операторов `new` и `delete`:

- `new` — для выделения памяти;
- `delete` — для освобождения памяти.

Динамическое выделение памяти

Оператор new используется в следующих формах:

```
new тип;      // для переменных
new тип[размер];     // для массивов
```

Память может быть выделена для одного объекта или для массива любого типа, в том числе типа, определенного пользователем. Результатом выполнения операции new будет указатель на отведенную память, или нулевой указатель в случае ошибки.

```
int *ptr_i;
double *ptr_d;
struct person *human;
.................
ptr_i = new int;
ptr_d = new double[10];
human = new struct person;
```

Динамическое выделение памяти

Память, выделенная в результате выполнения new, будет считаться выделенной до тех пор, пока не будет выполнена операция delete.

Освобождение памяти связано с тем, как выделялась память – для одного элемента или для нескольких. В соответствии с этим существует и две формы применения delete

```
delete  указатель;   // для одного элемента
delete[] указатель;  // для массива
```

Например, для приведенного выше случая, освободить память необходимо следующим образом:

```
delete ptr_i;
delete[] ptr_d;
delete human;
```

Освобождаться с помощью delete может только память, выделенная оператором new.

Пример динамического выделения памяти

```
#include <iostream>
using namespace std;
int main()
{
 int size;
 int *dan;
 cout << "Ввести размерность массива: ";
 cin >> size;
 dan = new int[size];
 for (int i=0; i<size; i++)
 {
   cout << "dan[" << i << "]= ";
   cin >> dan[i];
 }
 delete[] dan;
 cin.get(); cin.get();
 return 0;
}
```

dan – базовый адрес динамически выделяемого массива, число элементов которого равно size. Операцией delete освобождается память, выделенную при помощи new.

Области видимости в C++

C++ поддерживает три области видимости:

- область видимости файла (глобальная область видимости);
- локальная область видимости;
- область видимости класса (абстрактного типа данных).

Локальная область видимости – это область видимости внутри блока. Каждая функция – это отдельная область видимости. Внутри функции может быть несколько блоков, заключенных в фигурные скобки `{...}`, также образующих отдельные области видимости. Для переменных, объявленных в блоке, область видимости – от точки объявления до конца блока.

Области видимости в C++

Если переменная в локальной области видимости переопределяет переменную из глобальной области видимости, то можно явно указать область видимости, используя оператор разрешения контекста (области видимости) `::`.

Оператор разрешения контекста имеет самый высокий приоритет и применяется в двух формах:

- унарная — ссылается на внешний контекст;
- бинарная — ссылается на контекст класса.

Унарная форма записи:

:: Идентификатор

Бинарная форма записи:

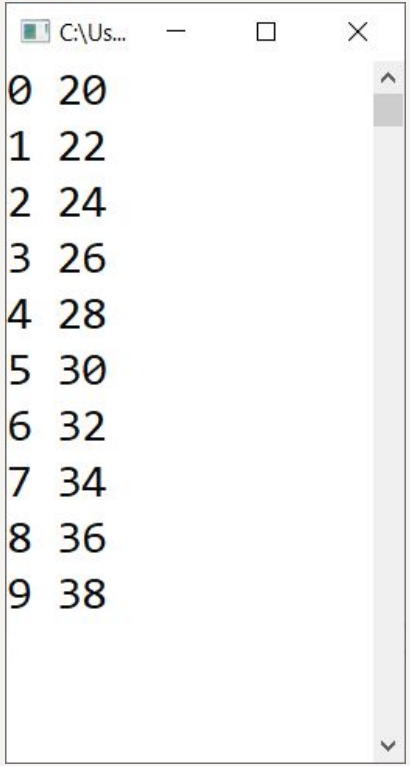
ИмяКласса :: Идентификатор

Области видимости в C++

Унарная форма используется для обращения к имени, относящемуся ко внешнему контексту и скрытому локальным контекстом или контекстом класса.

```
#include <iostream>
using namespace std;
int count = 20; // (*)
void func()
{
    for (int count = 0; count < 10; count++)
    {
        cout << count << " " << ::count << endl;
        ::count += 2; // увеличивает на 2 (*)
    }

}
int main()
{
    func();
    cin.get();
    return 0;
}
```

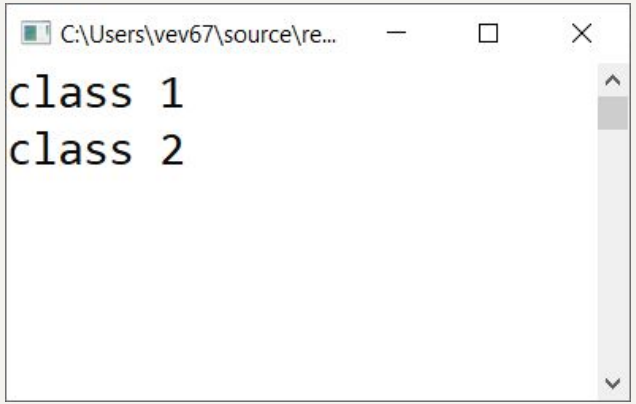


Области видимости в C++

Бинарная форма используется для ссылки на контекст класса с целью устранения неоднозначности имен, которые могут повторно использоваться внутри класса.

```
#include <iostream>
using namespace std;
struct cl1 { void f(); };
struct cl2 { void f(); };
void cl1::f() { cout << "class 1" << endl; }     // обращение к f() из cl1
void cl2::f() { cout << "class 2" << endl; }     // обращение к f() из cl2
int main()
{
  cl1 c;
  c.f();
  cl2 d;
  d.f();

  cin.get();
  return 0;
}
```



Пространства имен в C++

Области видимости могут быть вложенными.

Для разделения областей видимости используются так называемые пространства имен.

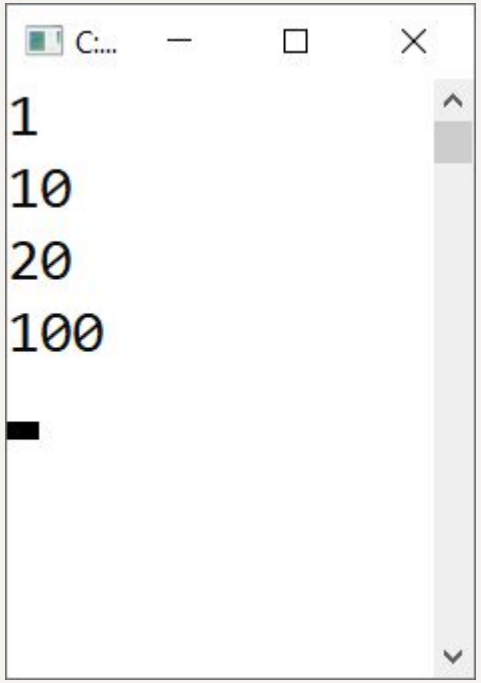
Пространство имён (`namespace`) — некоторое множество, под которым подразумевается модель, абстрактное хранилище или окружение, созданное для логической группировки уникальных идентификаторов (имён).

Объявление пространства имен имеет синтаксис:

```
namespace Идентификатор
{
  Содержимое
}
```

Пространства имен в C++

```
#include <iostream>
using namespace std;
int X = 100;
namespace A {
    int X = 10;
    namespace B {
        int X = 20;
    }
}
int main() {
    int X = 1;
    cout << X << endl;
    cout << A::X << endl;
    cout << A::B::X << endl;
    cout << ::X << endl;
    cin.get();
    return 0;
}
```



Абстрактные типы данных (АТД)

Язык C++ позволяет создавать типы данных, которые ведут себя аналогично базовым типам языка C. Такие типы обычно называют **абстрактными типами данных** (АТД).

Для реализации АТД в языке C используются структуры. Но использование данных структурного типа значительно ограничено по сравнению с использованием базовых типов данных. Например, структурные данные нельзя использовать как операнды в различных операциях (сложение, вычитание). Для манипуляции с подобными данными надо писать набор функций, выполняющих различные действия, и вместо операций вызывать эти функции.

Кроме того, элементы структуры никак не защищены от случайной модификации. То есть любая функция (даже не из набора средств манипуляции структурными данными) может обратиться к элементу структуры. Это противоречит одному из основных принципов объектно-ориентированного программирования — **инкапсуляции данных**: никакие другие функции, кроме специальных функций манипуляции этим типом данных, не должны иметь доступ к элементам данных.

Абстрактные типы данных (АТД)

Рассмотрим реализацию понятия даты с использованием struct для того, чтобы определить представление даты date и множества функций для работы с переменными этого типа:

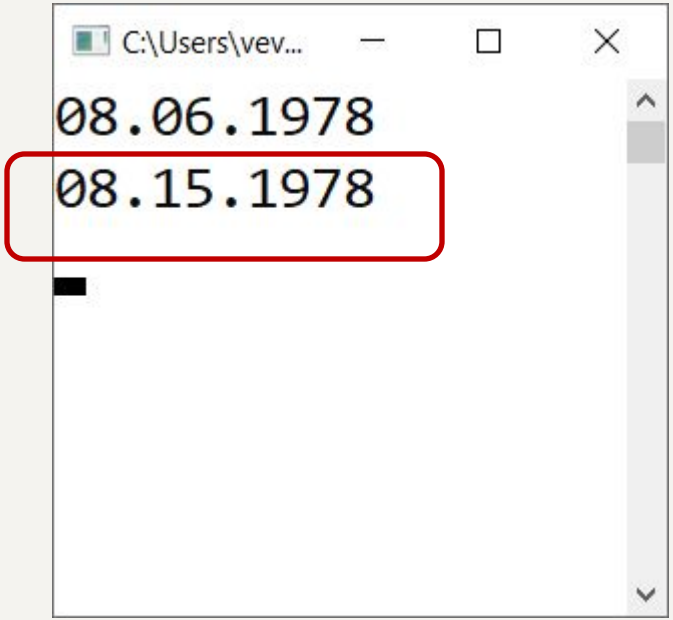
```
#include <iostream>
using namespace std;
struct date
{
    int day;  // день
    int month;// месяц
    int year; // год
};
```

Абстрактные типы данных (АТД)

```
void set(date* d, int Day, int Month, int Year)
{
    d->day = Day;
    d->month = Month;
    d->year = Year;
}
void out(date d)
{
    cout.fill('0');
    cout.width(2);
    cout << d.day << ".";
    cout.width(2);
    cout << d.month << "." << d.year << endl;
}
```

Абстрактные типы данных (АТД)

```
int main()
{
    date D;
    set(&D, 8, 6, 1978);
    out(D);
    D.month = 15;
    out(D);
    cin.get(); cin.get();
    return 0;
}
```



Поля структуры доступны для непосредственного изменения, что нарушает принцип инкапсуляции ООП.

Абстрактные типы данных (АТД)

Никакой явной связи между функциями и типом данных в этом примере нет.

Такую связь можно установить, описав функции как члены структуры. Эти функции могут действовать на данные, содержащие в самой структуре.

По умолчанию при объявлении структуры ее данные и функции являются общими, то есть у объектов типа структура нет ни инкапсуляции, ни защиты данных.

Абстрактные типы данных (АТД)

```
#include <iostream>
using namespace std;
struct date {
    int day;  // день
    int month;// месяц
    int year; // год
    void set(int Day, int Month, int Year) {
        day = Day;
        month = Month;
        year = Year;
    }
    void out() {
        cout.fill('0');
        cout.width(2);
        cout << day << ".";
        cout.width(2);
        cout << month << "." << year << endl;
    }
};
```

```
int main()
{
    date D;
    D.set(8, 6, 1978);
    D.out();
    D.month = 15;
    D.out(D);
    cin.get();
    return 0;
}
```

Права доступа

Концепция АТД в языке C++ (в отличии от C) позволяет членам АТД быть общими, частными или защищенными:

`public` – общие;
`private` – частные;
`protected` – защищенные.

Использование ключевого слова `protected` связано с понятием наследования, и подробно будет рассмотрено позже.

Использование ключевого слова `private` ограничивает доступ к членам, которые следуют за этой конструкцией. Члены `private` могут использоваться только несколькими категориями функций, в привилегии которых входит доступ к этим членам.

Ключевое слово `public` образует интерфейс к объекту структуры.

Классы

Ключевым понятием абстрактного типа данных в языке C++ является класс.

Классы в C++ — это абстракция описывающая поля (свойства) и методы (действия), ещё не существующих объектов.

Классы в C++ определяются ключевым словом `class`. Они представляют собой форму АТД, у которой спецификация доступа по умолчанию – `private` (скрытые).

Структуры принято использовать в тех случаях, когда сокрытие данных неуместно.

Функции, описанные внутри класса, называются **функциями-членами** или **методами** класса.

Данные (чаще всего - переменные), описанные внутри класса, называются **данными-членами** или **полями** класса.

Классы

Если данные-члены размещаются в открытой области видимости, то они называются еще **свойствами** класса.

В примере далее поля класса размещены в скрытой области видимости, в то время как методы размещены в открытой области видимости.

В общем случае часть методов может располагаться в скрытой области видимости (скрытые или внутренние методы).

Методы, размещенные в открытой области видимости, называются **интерфейсом класса**.

Внутри метода класса имена полей того же класса могут использоваться без явной ссылки на объект. В этом случае имя относится к полю того объекта, для которого метод был вызван.

Классы

```
#include <iostream>
using namespace std;
class date
{
private:
    int day;
    int month;
    int year;
public:
    void set(int Day, int Month, int Year)
    {
        day = Day;
        month = Month;
        year = Year;
    }
    void out()
    {
        cout.fill('0');
        cout.width(2);
        cout << day << ".";
        cout.width(2);
        cout << month << "." << year
        << endl;
    }
};
int main()
{
    date D;
    D.set(8, 6, 1978);
    D.out();
    cin.get(); cin.get();
    return 0;
}
```

Поля и методы класса

Определение методов может осуществляться двумя способами:

- описание функции непосредственно при описании класса (см.пример выше);
- описание функции вне класса.

Методы, которые определены внутри класса, являются неявно встроенными (`inline`). Как правило, только короткие, часто используемые методы должны определяться внутри класса.

Поскольку разные классы могут иметь методы с одинаковыми именами, при определении метода необходимо указывать имя класса, связывая их с помощью оператора разрешения контекста `::`

В методах имена полей класса могут использоваться без явной ссылки на объект. В этом случае имя относится к полю того объекта, для которого функция была вызвана.

Методы одного и того же класса могут быть полиморфными, то есть перегруженными.

Описание методов вне класса

```
#include <iostream>
using namespace std;
class date
{
    int day;
    int month;
    int year;
public:
    void set(int Day, int Month, int Year);
    void out();
};
void date::set(int Day, int Month, int Year)
{
    day = Day;
    month = Month;
    year = Year;
}
```

```
void date::out()
{
    cout.fill('0');
    cout.width(2);
    cout << day << ".";
    cout.width(2);
    cout << month << "." << year
    << endl;
}
int main()
{
    date D;
    D.set(8, 6, 1978);
    D.out();
    cin.get();
    return 0;
}
```

Объекты класса

Объекты — конкретное представление абстракции, имеющее свои свойства и методы.

Созданные объекты на основе одного класса называются **экземплярами** этого класса.

Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса.

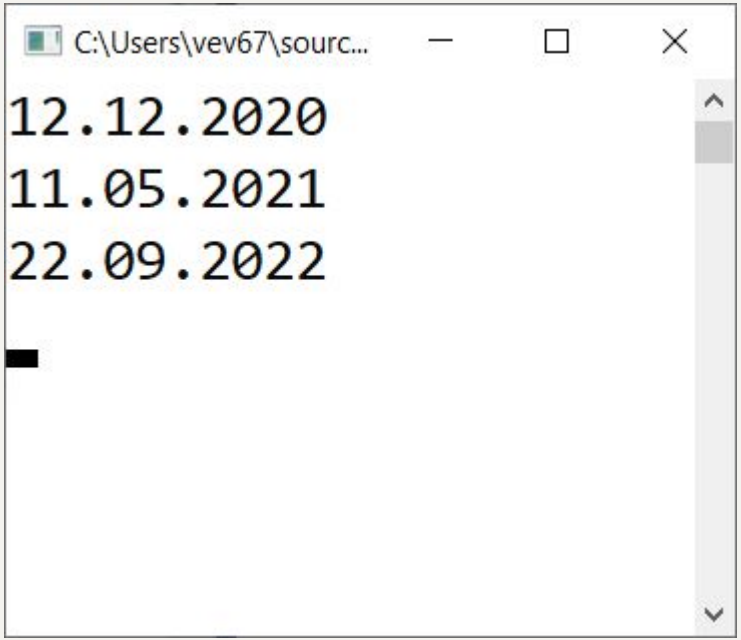
Каждый объект обладает уникальными значениями полей. Однако набор полей и методов остается одинаковым для всех объектов указанного класса.

Поскольку методы класса оперируют полями этого класса, то каждый экземпляр класса при вызове метода получает разные значения в зависимости от полей этого класса.

Создание объекта на основе описанного ранее класса называется **инстанцированием** (от instance – сущность).

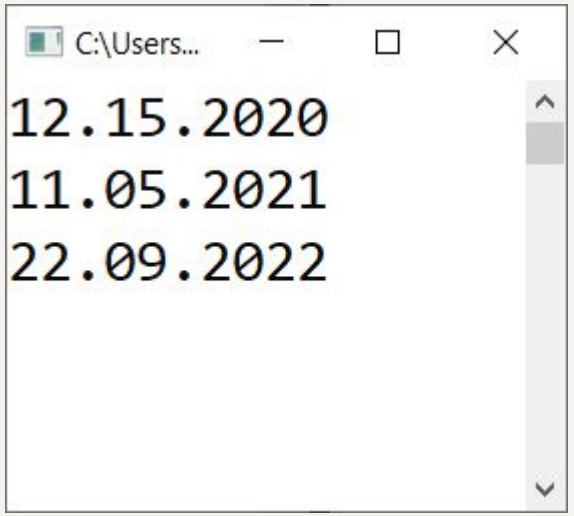
Объекты класса

```
int main()
{
    date A;              // объект A класса date
    date B;              // объект B класса date
    date C;              // объект C класса date
    A.set(12, 12, 2020);
    B.set(11, 5, 2021);
    C.set(22, 9, 2022);
    A.out();
    B.out();
    C.out();
    cin.get();
    return 0;
}
```



Инкапсуляция в действии

```
int main()
{
    date A;              // объект A класса date
    date B;              // объект B класса date
    date C;              // объект C класса date
    A.set(12, 15, 2020);
    B.set(11, 5, 2021);
    C.set(22, 9, 2022);
    A.out();
    B.out();
    C.out();
    cin.get();
    return 0;
}
```



Инкапсуляция в действии

```
#include <iostream>
using namespace std;
class date
{
    int day;
    int month;
    int year;
    void defaultDate() { day = 1; month = 1; year = 2000; }
public:
    void set(int Day, int Month, int Year);
    void out() const;
};
```

Инкапсуляция в действии

```
void date::set(int Day, int Month, int Year)
{
    if((Year < 1900) || (Year > 2100)) {defaultDate(); return; }
    if (Day < 1) { defaultDate(); return; }
    switch (Month)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            if(Day > 31) { defaultDate(); return; }
            break;
```

Инкапсуляция в действии

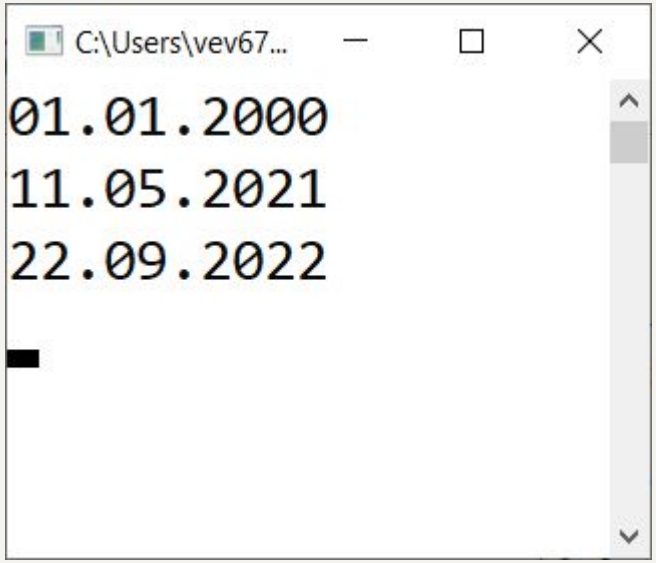
```
        case 4:
        case 6:
        case 9:
        case 11:
            if (Day > 30) { defaultDate(); return; }
            break;
        case 2:
            if(Day > 29) { defaultDate(); return; }
            if((Year % 4 > 0) && (Day > 28)) { defaultDate(); return; }
            break;
        default:
             defaultDate(); return;
    }
    day = Day;
    month = Month;
    year = Year;
}
```

Инкапсуляция в действии

```
void date::out() const
{
    cout.fill('0');
    cout.width(2);
    cout << day << ".";
    cout.width(2);
    cout << month << "." << year << endl;
}
```

Инкапсуляция в действии

```
int main()
{
    date A;              // объект A класса date
    date B;              // объект B класса date
    date C;              // объект C класса date
    A.set(12, 15, 2020);
    B.set(11, 5, 2021);
    C.set(22, 9, 2022);
    A.out();
    B.out();
    C.out();
    cin.get(); cin.get();
    return 0;
}
```



Статические члены класса

Класс – это тип, а не объект данных, и в каждом объекте класса имеется своя собственная копия данных – полей этого класса. Однако некоторые типы требуется реализовать так, что все объекты этого типа могут совместно использовать (разделять) некоторые данные. Такие разделяемые данные должны быть описаны как часть класса.

Статические данные относятся ко всем объектам класса. Такие данные используются, если

- требуется контроль общего количества объектов класса;
- требуется одновременный доступ ко всем объектам или части их;
- требуется разделение объектами общих ресурсов.

В этом случае в определение класса могут быть введены статические члены.

Статические поля класса

Статические данные описываются с помощью ключевого слова `static`, которое может использоваться при объявлении член-данных и член-функций.

Такие члены классов называются **статическими**, и независимо от количества объектов данного класса, существует только одна копия статического элемента.

Обращение к статическому элементу осуществляется с помощью оператора разрешения контекста и имени класса

ИмяКласса :: ИмяЭлемента

Если x – статическое поле класса `cl`, то к нему можно обращаться как

```
cl::x
```

При этом не имеет значения количество объектов класса `cl`.

Инициализироваться статические поля класса должны за пределами класса с использованием оператора разрешения контекста ::

Инициализация статических полей класса

Статические поля класса можно рассматривать как глобальную переменную класса. Но в отличие от обычных глобальных переменных, на статические члены распространяются правила видимости `private` и `public`. Поместив статическую переменную в часть `private`, можно ограничить ее использование.

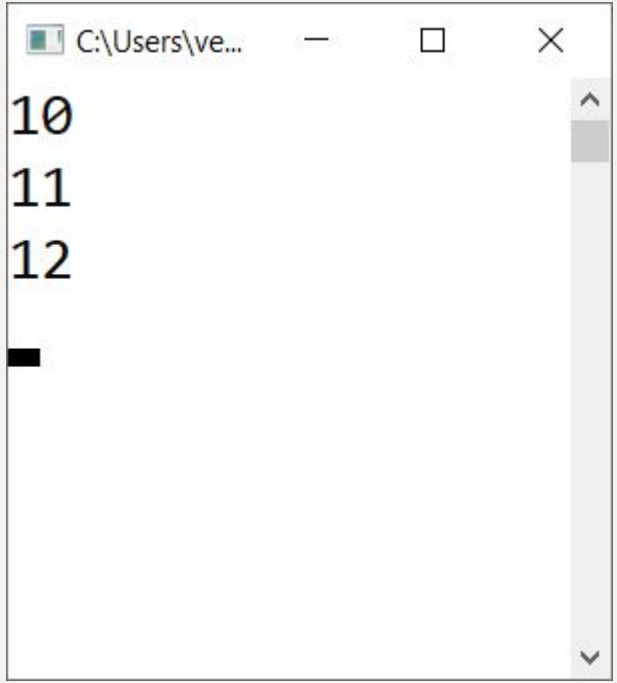
Объявление статического члена в объявлении класса не является определением, то есть это объявление статического члена не обеспечивает распределения памяти и инициализацию.

Статические поля нельзя инициализировать в теле класса, а также в методах класса (даже статических). Статические поля должны инициализироваться аналогично глобальным переменным в области видимости файла.

Статические поля и методы класса

Аналогично можно обращаться к статическому методу.

```
#include <iostream>
using namespace std;
class X
{
    static int A; // статическое поле
public:
    static int getA() { return A++; } // метод
};
int X::A = 10; // инициализация статического поля
int main()
{
    cout << X::getA() << endl; // вызов
    cout << X::getA() << endl; // статического
    cout << X::getA() << endl; // метода
    cin.get();
    return 0;
}
```



Константные объекты

Константные члены класса описываются с помощью ключевого слова `const`. Объявление константного члена гарантирует, что его значение в области видимости не будет меняться.

С константным объектом могут работать методы класса, объявленные с ключевым словом `const`. Такие функции-члены не изменяют данные-члены класса. Слово `const` помещается между списком аргументов и телом функции-члена.

Если метод определяется вне тела класса, то `const` указывается и в объявлении и в определении функции.

Константные объекты

```
#include <iostream>
using namespace std;
class date
{
    int day;
    int month;
    int year;
public:
    void set(int Day, int Month, int Year);
    void out() const;
};
void date::set(int Day, int Month, int Year)
{
    day = Day;
    month = Month;
    year = Year;
}
```

Константные объекты

```
void date::out() const
{
    cout.fill('0');
    cout.width(2);
    cout << day << ".";
    cout.width(2);
    cout << month << "." << year << endl;
}
```

```
int main()
{
    date A;
    date B;
    date C;
    A.set(12, 12, 2020);
    B.set(11, 5, 2021);
    C.set(22, 9, 2022);
    A.out();
    B.out();
    C.out();
    cin.get(); cin.get();
    return 0;
}
```