2-PROGRAM STRUCTURE
STATEMENTS AND FLOW CONTROL

even[evennumber++] = i;

bifurcate = çatal yapmak, iki kola ayrılmak

{ statement1; statement2; statement3; }
The entire block is considered a single statement (composed itself of multiple substatements)

fulfilled = yerine getirilmiş, başarılmış, uygulanmış
indeed = doğrusu, gerçekten de
concatenate = sıralamak, zincirlemek, ard arda bağlamak
countdown = gerisayım, gerisayma

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

int main (){
    //Sleep for time span           sleep_for
    cout<<"countdown:\n";
    for(int i=5; i>0;){
        cout<<i--<<endl;
        this_thread::sleep_for (chrono::seconds(1));
    }
    return 0;
}
```

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required.

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
```

but rather = -den çok, -den ziyade, aksine
range = sıralamak
Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end.
exclusively = yalnızca, sadece
acquaint = tanıtmak, haber vermek
precede = -den üstün olmak, önce olmak

```
for ( declaration : range ) statement;
```

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str {"Hello Safak!"};
    for (char c : str)
        cout << "[" << c << "]";
}
```

```
for (auto c : str)
  std::cout << "[" << c << "]";
```

Here, the type of c is automatically deduced as the type of the elements in str.

aborted = boşa çıkmış, durduruldu, kesildi
preferably = tercihen, öncelikli olarak
in the presence of = gözü önünde, -in huzurunda
be deemed to = sayılmak, addedilmek
peculiar = garip, olağandışı

FUNCTIONS
bearing in mind that = -i hesaba katarak, -i göz önünde bulundurarak.
in the case of = durumunda, halinde
commutative = değiştirilebilen
absence = bulunmayış, yokluk, yitiklik
there is a catch = dikkat edilmesi gereken bir şey var
implicitly or explicitly(sçıkca) = dolaylı ya da dolaysız

When **main** returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. (on all platforms)

though = -diği halde, -e karşın
altogether = büsbütün, hep beraber
in this regard = bu konuda, bu bakımdan
perceive = algılamak, hissetmek

```
string concatenate (const string& a, const string& b){
                    return a+b;
}
```

By qualifying them as **const,** the function is forbidden to modify the values of neither **a** nor **b,** but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

merely = yalnızca, sadece
invoke = yardım istemek, çağırmak, çalıştırmak
inform = bilgi vermek
in such a way that = şöyle ki, gibi

```
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}

//        6                          //        5
```

```
int protofunction (int first, int second);
int protofunction (int, int );
```

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

```
Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
It is even.
Please, enter number (0 to exit): 1030
It is even.
Please, enter number (0 to exit): 0
It is even.
```

This example is indeed not an example of efficiency. It can be written half code.

concretely = hissedilebilir bir şekilde, fiziksel olarak

OVERLOADS AND TEMPLATES

instantiate = kanıt sunmak, savı örnek vererek desteklemek

```cpp
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

```
11
2.5
```

In this case, we have used **T** as the template parameter name.
We used the function template **sum** twice. The first time with arguments of type **int,** and the second one with arguments of type **double.**
Therefore, result will be a variable of the same type as the parameters **a** and **b,** and as the type returned by the function.

It is possible to instead simply write:
k = sum (i,j)
h = sum(f,g)

deduce = sonuç çıkarmak, anlamak
unambiguous = kesin, belirsizliğe yer vermeyen

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

```
x and y are equal
```

is equivalent to
are_equal < int, double > (10, 10.0)

can also include expressions of a particular type

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

```
20
30
```

NAME VISIBILITY
nevertheless = bütün bunlara rağmen, yine de

Namespaces allow us to group named entities that otherwise would have **global scope** into narrower scopes, giving them **namespace scope.** This allows organizing the elements of programs into different logical scopes referred to by names.

namespace identifier

```
{
        named_entities
}

namespace   myNamespace
{
        int a, b;
}
```

These variables can be accessed from within their namespace normally, with their identifier (either a or b), but if accessed from outside the myNamespace namespace they have to be properly qualified with the scope operator ::.

```
myNamespace::a
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions.

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

There are two functions with the same name: value. One is defined within the namespace **foo,** and the other one in **bar.**

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

This declares three variables: **a** and **c** are in namespace **foo,** while **b** is in namespace **bar.**

using

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```
5
2.7183
10
3.1416
```

Notice how in **main,** the variable x (without any name qualifier) refers to first::x, whereas y refers to second::y, just as specified by the **using** declarations.

The keyword **using** can also be used as a directive to introduce an entire namespace:

```cpp
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

```cpp
// using namespace example
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

```
5
3.1416
```

facilitate = hafifletmek, olanak sağlamak
comprehension = anlama, kavrama

It is common to instead see:
std::cout<< “Hello World!”;

Global variables are automatically initialized to zeroes.