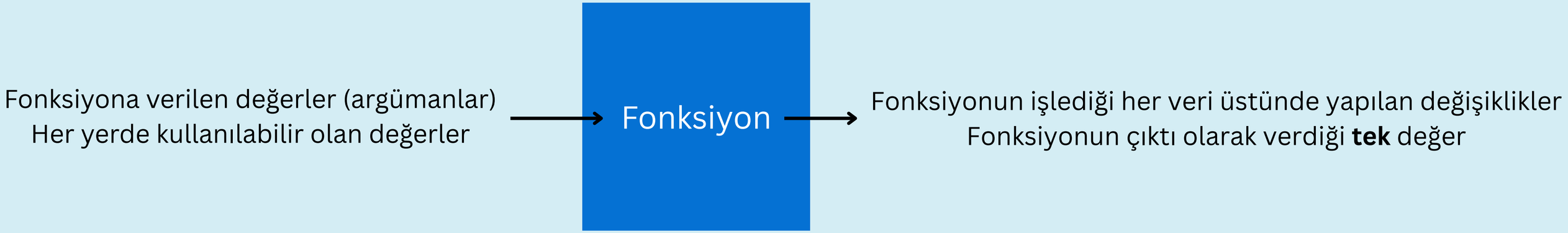
FONKSİYONLAR VE DÖNGÜLER

C++ ile Programlamaya Giriş

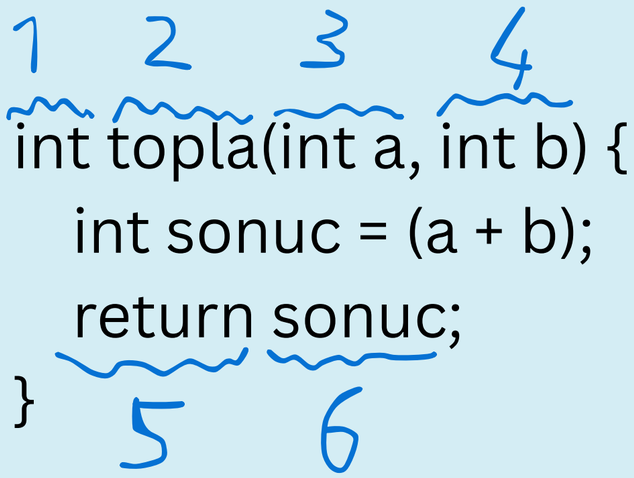
SKYLAB | ALGOLAB

Fonksiyon

Kodun okunabilirliğini, modülerliğini (parçalara ayrılmış hâlini) sağlamak amacıyla programlamada bulunan yapılar/kısayollardır.



Fonksiyonun Yapısı



1 → fonksiyonun çıktı olarak vereceği verinin türüdür, bu fonksiyon bir int yani tam sayı döndürür.

2 → fonksiyonun adıdır, bu fonksiyon programın içerisinde bu adla çağırılır.

3, 4 → fonksiyonun argümanlarıdır, bu argümanların ikisinin de türü int (tam sayı), adları ise a ve b'dir. sadece bu fonksiyonun içerisinde bu ad ile kullanılabilirler.

5, 6 → return ifadesi fonksiyonun bittiğini ve döndüreceği değeri gösterir, bir bloğun içerisinde return varsa o bloğun kalanı çalışmaz, return bitiş noktasıdır. sonuc ise fonksiyonun çıktı olarak döndüreceği tam sayı değeridir.

Örnek fonksiyon ve kullanımı

```cpp
#include <iostream>

using namespace std;

int topla(int a, int b) {
  int sonuc = (a + b);
  return sonuc;
}

int main() {
  cout << topla(2, 5) << endl;
  return 0;
}
```

topla(2, 5) ifadesinin çıktısı 7 olur.

Fonksiyonun çalışmasının işlemci açısından yorumu

```cpp
#include <iostream>

using namespace std;

int topla(int a, int b) {
  int sonuc = (a + b);
  return sonuc;
}

int main() {
  cout << topla(2, 5) << endl;
  return 0;
}
```

Jump/zıplama: İşlemci üstündeki yükü oluşturan etkenlerdendir. Aşırı yoğun zıplama yapılırsa bu programın yavaşlamasına sebep olur. Bu örnekte program bir zıplama yapar.

Çoklu fonksiyon örneği

```cpp
#include <iostream>

using namespace std;

int topla(int a, int b) {
  int sonuc = (a + b);
  return sonuc;
}

int bol(int a, int b) {
  int sonuc = (a / b);
  return sonuc;
}

int main() {
  cout << bol(topla(3, 5), 2) << endl;
  return 0;
}
```

Mantık önceki örnekle büyük oranda aynıdır. bol() fonksiyonu birinci argüman olarak topla() fonksiyonunun çıktısını yani 8'i alır. Ardından bol(8, 2) çalışır ve çıktı olarak 4 döndürür.

Burada 2 tane zıplama vardır.

Zıplamayı optimize etme yolları

```cpp
#include <iostream>

using namespace std;

static inline int topla(int a, int b) {
  int sonuc = (a + b);
  return sonuc;
}

static inline int bol(int a, int b) {
  int sonuc = (a / b);
  return sonuc;
}

int main() {
  cout << bol(topla(3, 5), 2) << endl;
  return 0;
}
```

Fonksiyonun başına static inline ekleyerek derleyiciye zıplama yap demek yerine fonksiyonun kullanıldığı yerde fonksiyonun içerisindeki kodları direkt olarak çalıştır ve burada kullan diyebiliriz.

Not: static inline eklenen fonksiyonlar sadece bulunduğu cpp dosyasında kullanılabilir. Ek olarak kısa fonksiyonlara eklenmeliler.

Fonksiyon argümanları

Eğer fonksiyona belirtilen miktardan az/fazla ya da farklı tipte argüman girilirse programı derlerken hata alırsınız.

Örneğin eksik argüman girdiğimizde:

```
→  test g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:16:20: error: too few arguments to function ‘int topla(int, int)’
   16 |    cout << bol(topla(3), 2) << endl;
      |                    ~~~~~^~~
test.cpp:5:19: note: declared here
    5 | static inline int topla(int a, int b) {
      |                   ^~~~
```

Ayrıca argümanlar belirtilen sıra ile girilmelidir.

İç içe fonksiyon örneği

İç içe fonksiyonları kullanırken içerideki fonksiyonlar düz fonksiyon gibi yazılamaz. Bu tarz fonksiyonlar için lambda fonksiyonlar kullanılır.

N tane isimsiz argüman girmek

Bu mümkündür ancak isimsiz olan argümanların hepsinin aynı tipte olması gerekir. Bu argümanların literatürde ismi “variadic arguments”tir.

```
1  #include <iostream>
2  #include <cstdarg>
3
4  using namespace std;
5
6  double topla(int n, ...) {
7    double sonuc = 0.0;
8
9    va_list args;
10   va_start(args, n);
11
12   for (int i = 0; i < n; ++i) {
13     sonuc += va_arg(args, double);
14   }
15
16   va_end(args);
17   return sonuc;
18 }
19
20 int main() {
21   cout << topla(3, 1.2, 3.1, 6.4) << endl;
22   return 0;
23 }
```

12. satırda for üstünkörü anlatımla kendisine ait bloğun içerisindeki kodu (13. satır) n defa çalıştırır.

Çıktı olarak 1.2+3.1+6.4'ten 10.7 döndürür.

Gördüğünüz gibi argümanların tamamı double olarak alındı.

Döngüler

Döngü bir işlemi şartlara bağlı olarak N defa gerçekleştirmemizi sağlayan yapılardır. Döngüler üçe ayrılır:

1. while
2. do-while
3. for

while döngüsü

```cpp
#include <iostream>

using namespace std;

int main() {
  int i = 0;

  while (i < 20) {
    cout << i << endl;
    i += 1;
  }
  return 0;
}
```

while içerisindeki condition (şart) kontrol edilir.
Eğer condition doğru ise while bloğu çalışır.
Çalışma bittikten sonra tekrar kontrol eder.
Örnekteki kod konsola sırasıyla;
0
1
2
...
19
yazar ve program biter.

do-while döngüsü

```cpp
#include <iostream>

using namespace std;

int main() {
  int i = 0;

  do {
    cout << i << endl;
    i += 1;
  } while (i < 20);

  return 0;
}
```

while döngüsü ile tamamen aynıdır. Tek farkı while ilk çalışırken şartı kontrol ederken do-while ilk seferde şartı kontrol etmeden çalışır.
Örnekteki kod konsola sırasıyla;
0
1
2
...
19
yazar ve program biter.

for döngüsü

```cpp
#include <iostream>

using namespace std;

int main() {
  for (int i = 0; i < 20; i += 1) {
    cout << i << endl;
  }

  return 0;
}
```

for içerisinde 3 tane statement var, ilk statement (int i = 0) sadece bir kere çalışır, genelde assignment için kullanılır. İkinci statement (i < 20) condition’dır, while döngüsündeki gibi bu kısım da sürekli kontrol edilir. Üçüncü statement ise döngü içerisindeki blok çalıştıktan sonra çalıştırılır. Bu da diğer örneklerde olduğu gibi 0'dan 19'a kadar yazdırır.

Döngüyü manipüle etmek

```cpp
#include <iostream>

using namespace std;

int main() {
  for (int i = 0; i < 10; i++) {
    if (i == 2) {
      continue;
    }

    if (i == 6) {
      break;
    }

    cout << i << endl;
  }

  return 0;
}
```

break ifadesi döngüyü bitirir.
continue ifadesi döngünün o kısmı sanki işlenmiş gibi davranır ve osnraki adıma geçer.

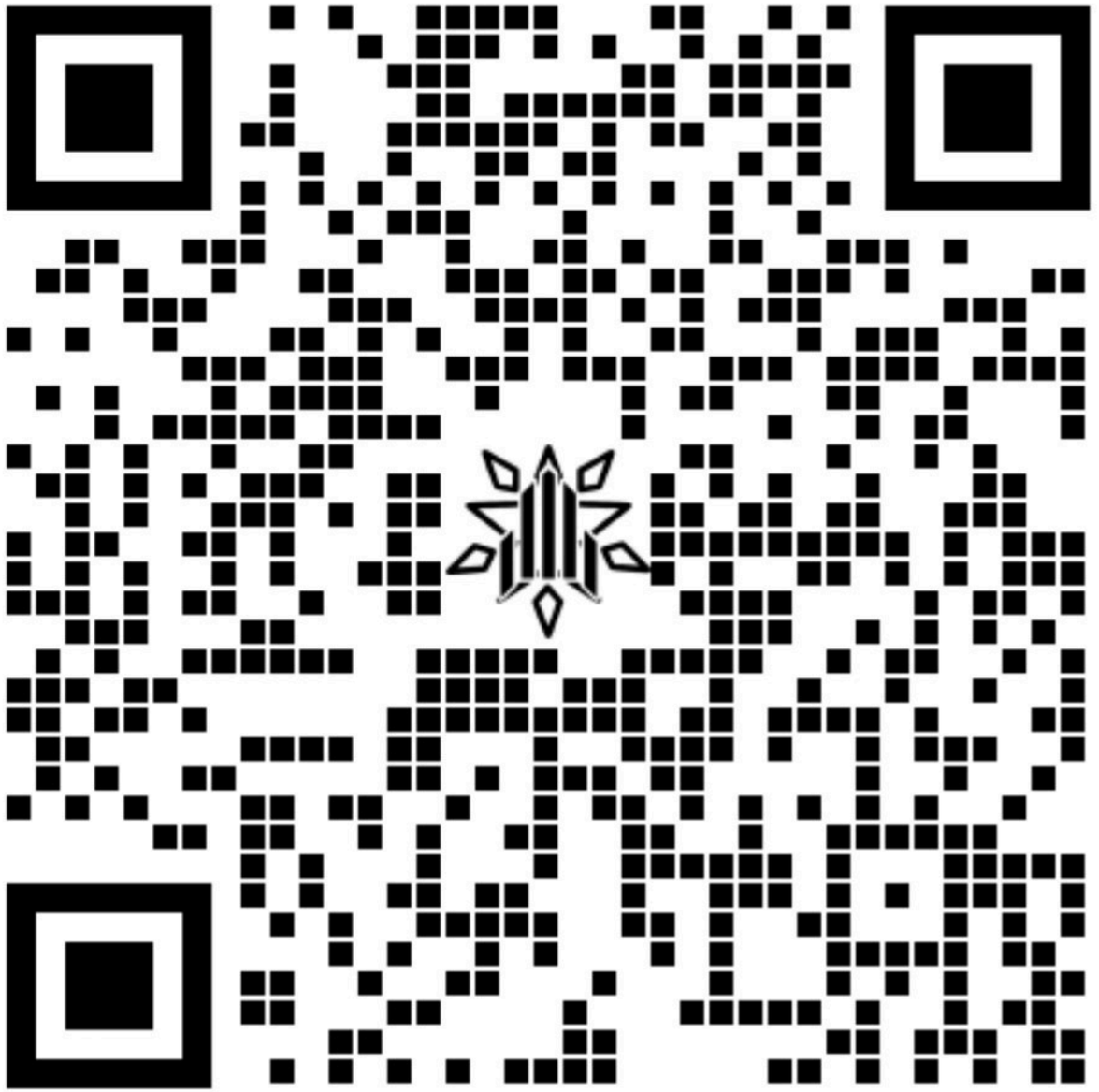
Çıktı;
0
1
3
4
5

Döngüler, jumping ve zaman karmaşıklığı

while ve do-while döngülerinde condition'ı her kontrol ederken bir kere jump yapar.
for döngüsünde ise şart kontrolünde bir jump, blok sonunda 3. statement çalıştırılmasında ise bir jump daha yapar. Bu yüzden while ve do-while döngüleri 1 jump, for döngüsü ise 2 jump ile çalışır.

Algoritma çözerken verdiğimiz verinin miktarına göre yaptığımız jump sayısına göre zaman karmaşıklığını belirleriz. 10 tane sayı verdiğimizde jump sayısı 13, 100 veri verdiğimizde ise bu sayı 103 ise bu algoritmanın/programın zaman karmaşıklığı O(N)'dir, buradaki N değeri verdiğimiz verinin boyutunu temsil eder. Aradaki 3 jump vb. değerler önemsizdir, amacımız verinin artışını temsil eden değişkeni bulmaktır. $O(N^2)$ içinse:
5 veri -> 27 jump
10 veri -> 102 jump

ARGE ekipleri katılım formu:



AGC katılım formu: