2 ソフトウェア工学 Software Engineering

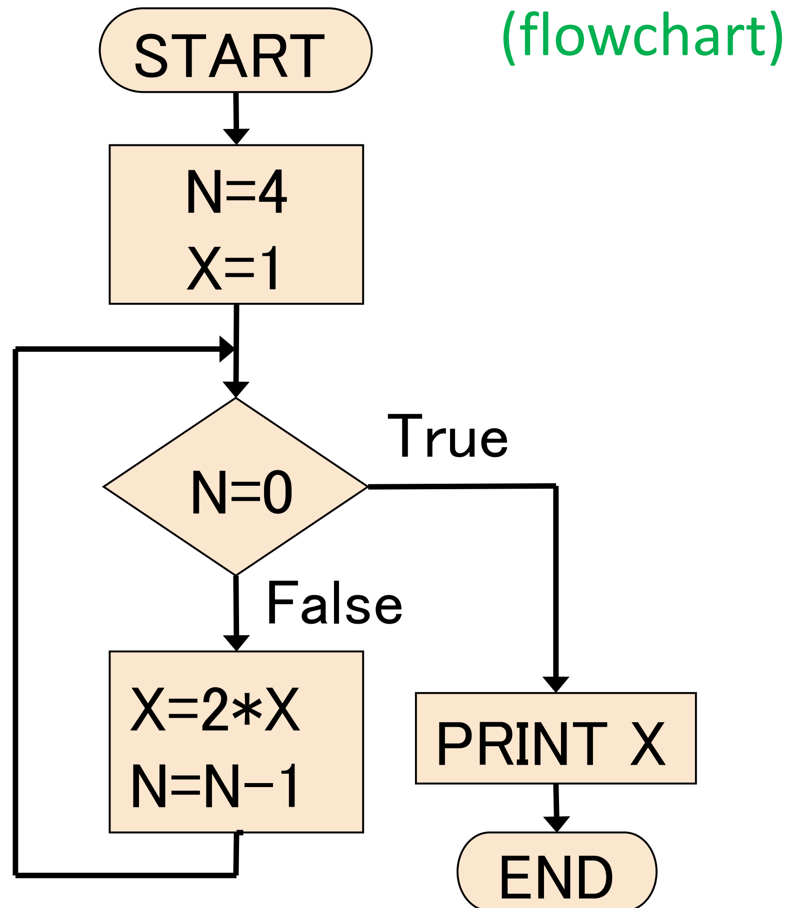
構造化プログラミング

STRUCTURED PROGRAMMING

GO TO 文

GO TO 文：機械語の分岐命令（JMPなど）に対応した制御文

■2 の N 乗を計算する流れ図



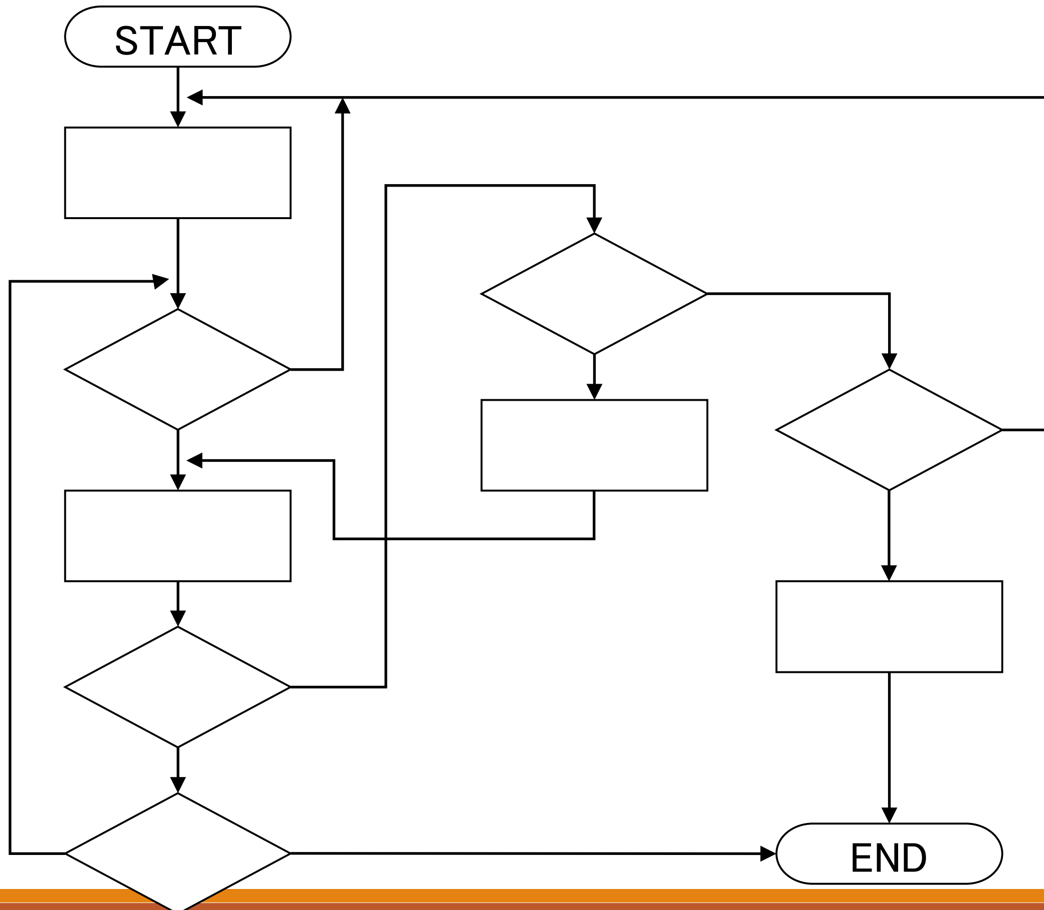
(flowchart)

（初期の頃の）
BASIC言語のプログラム

```
10 LET N=4
20 LET X=1
30 IF N=0 THEN GOTO 70
40 LET X=2*X
50 LET N=N−1
60 GOTO 30
70 PRINT X
80 END
```

スパゲティ･プログラム

スパゲティ･プログラム　(spaghetii program)
GO TO 文を不規則に用い，流れ図の線が交差し，
プログラムの全体を把握しにくいプログラム



GO TO 文有害説 (ダイクストラ, 1968)

E. Dijkstra: Go To Statement Considered Harmful,
Communications of the ACM, vol. 11, no. 3, pp.147-148, 1968.

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1$, $A2$, $\cdots$, An)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

Aus: *Communications of the ACM* **11**, 3 (March 1968). 147-148.

構造化定理 (ダイクストラ, 1967)

どんな流れ図も，3つの基本構造

順次 (sequence)

選択 (selection)

反復 (iteration)

の組合せにより，等価な

構造化流れ図　(structured flowchart)

に変換できる.

GO TO 文の使用制限

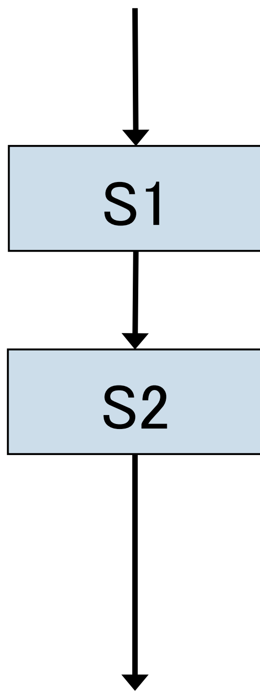
○ 流れ図の線の交差なし

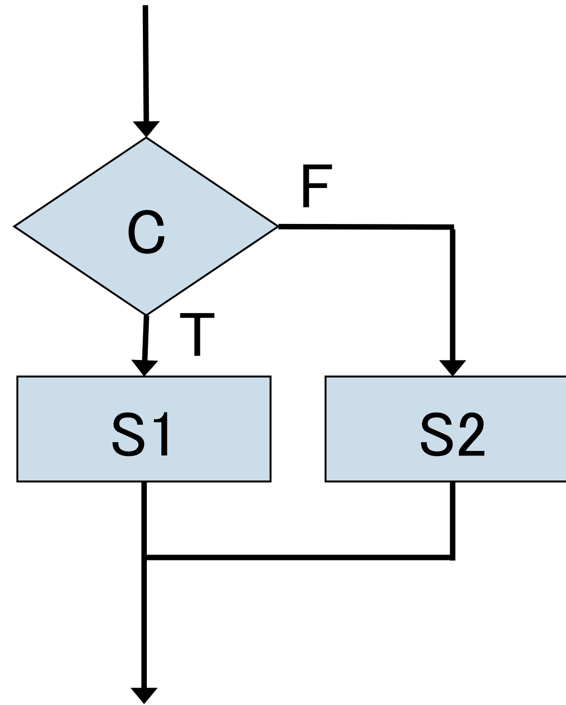
○ 出入り口が1カ所

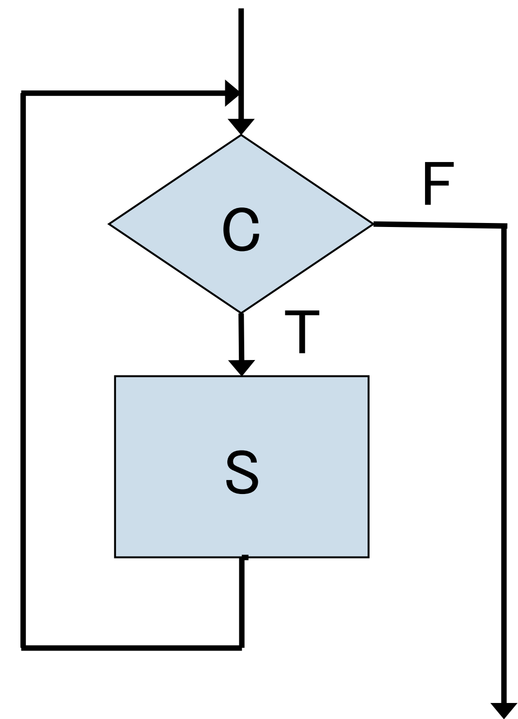
構造化プログラミング
Structured programming

3つの基本構造

順次 (sequence)	選択 (selection)	反復 (iteration)







S1; S2

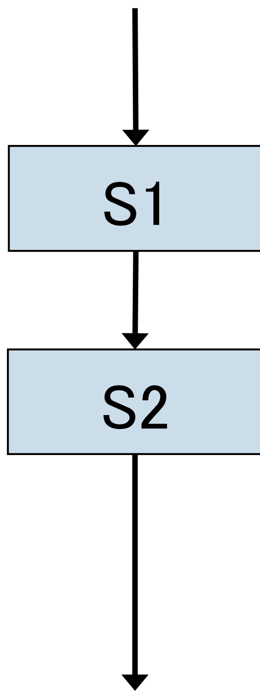
If C then S1 else S2

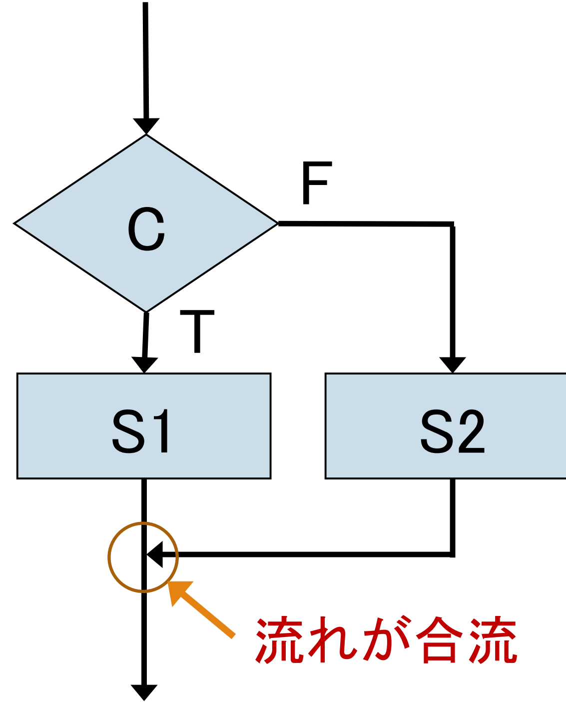
while C do S

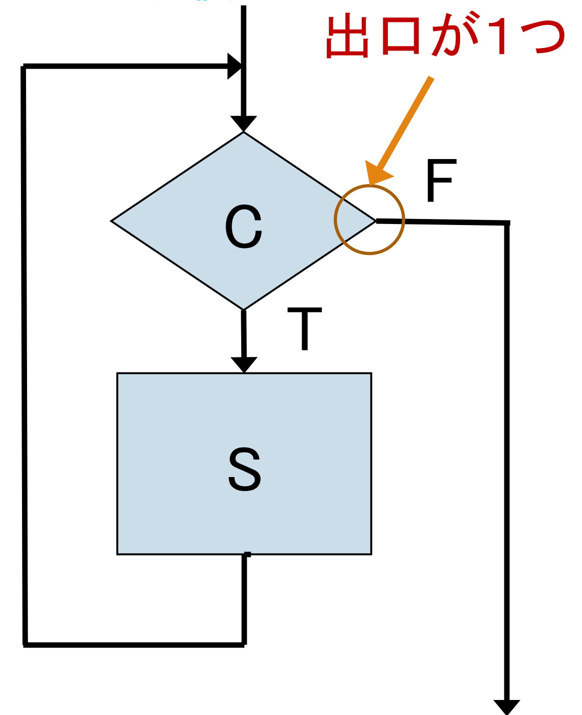
構造化プログラミング言語

構造化流れ図

S, S1, S2は，（再帰的に）ブロックであってもよい．





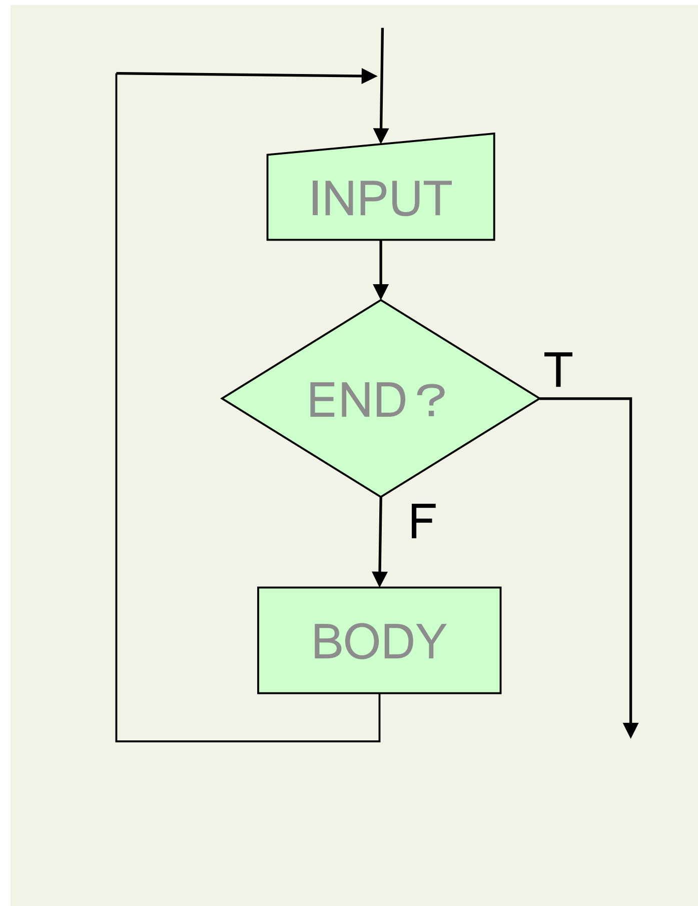


- 出入口はそれぞれ1カ所
- 流れを表す線が交差しない

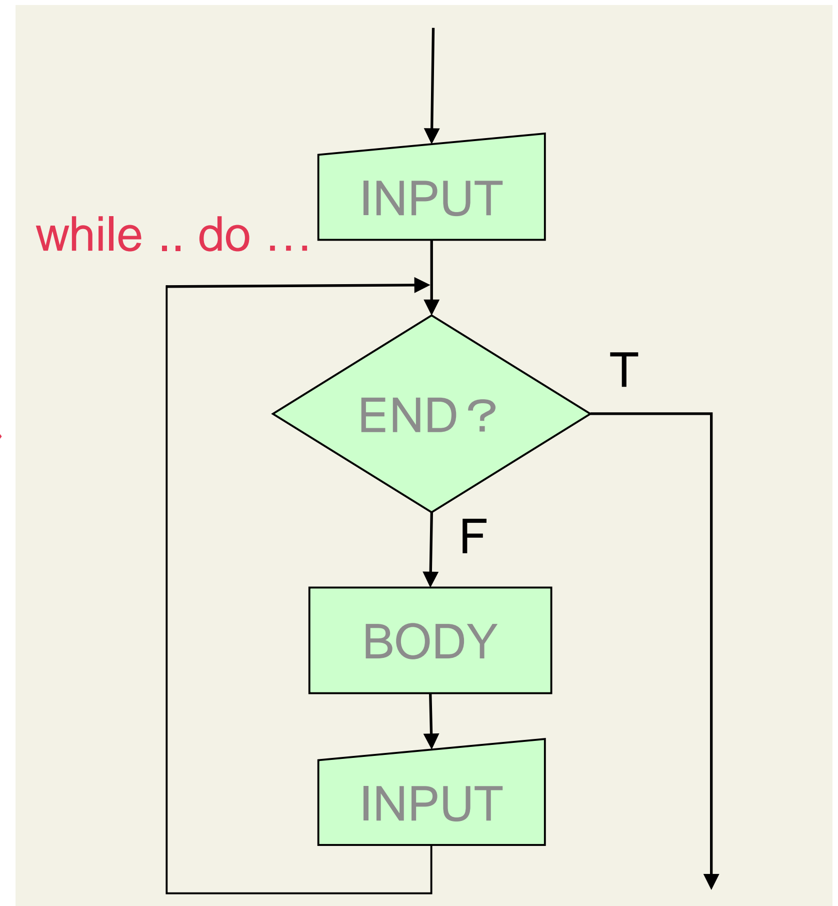
（数学的帰納法で証明できる）

構造化プログラムへの変換

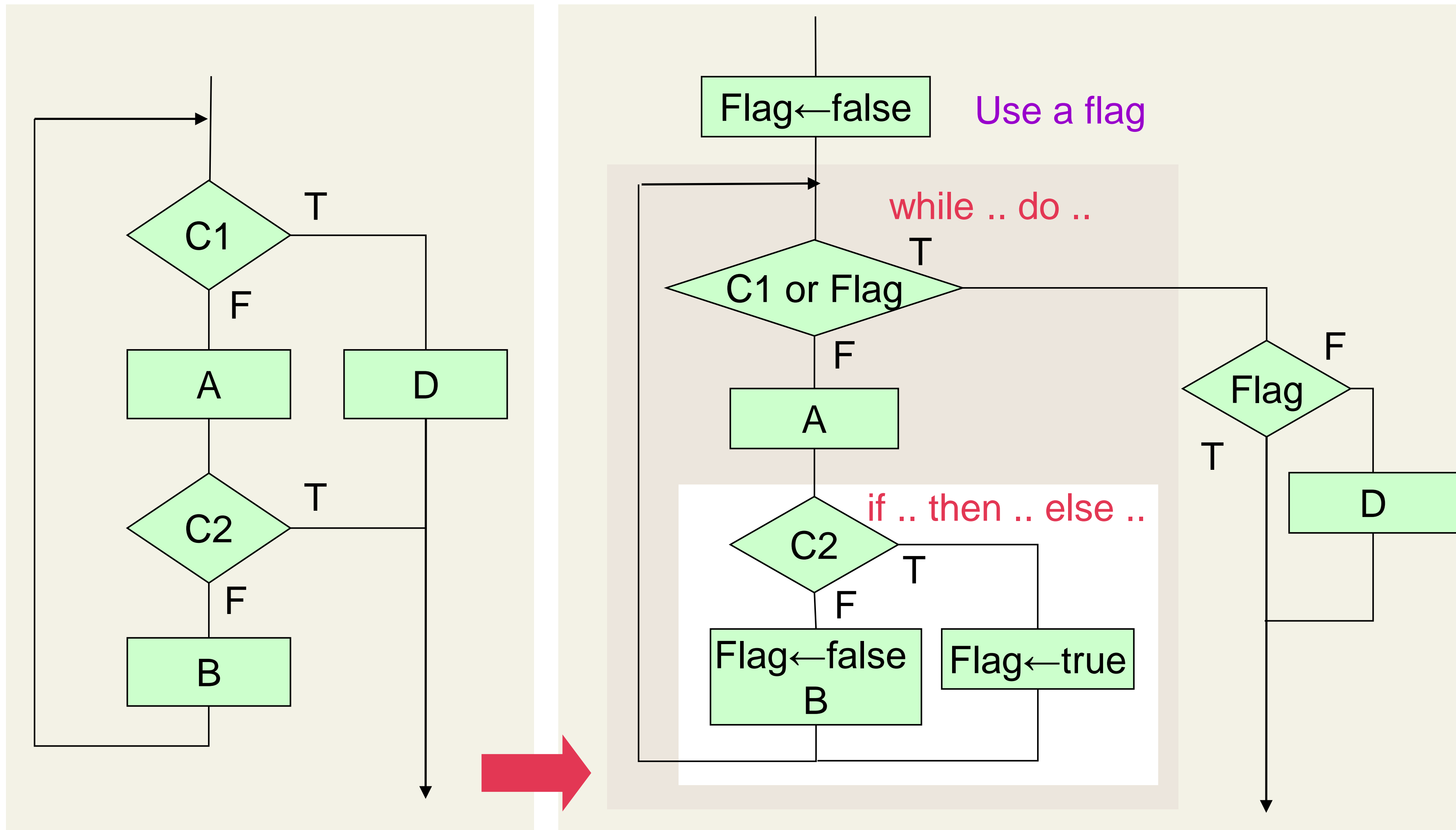
ループの途中に出口



ループの最初に出口

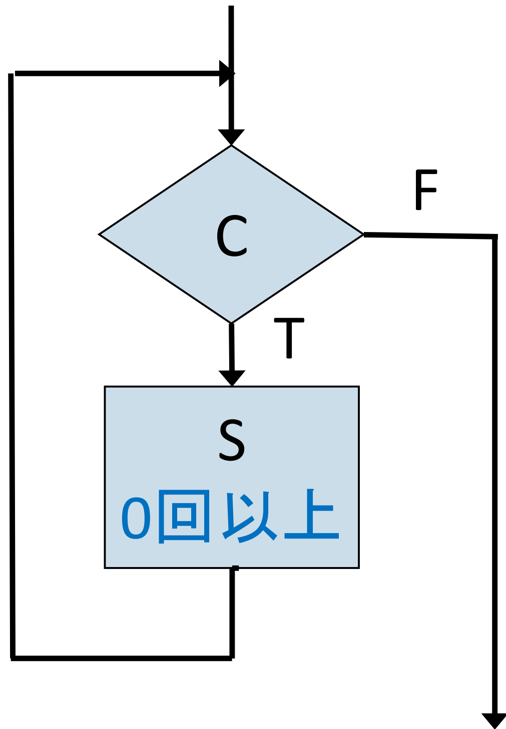


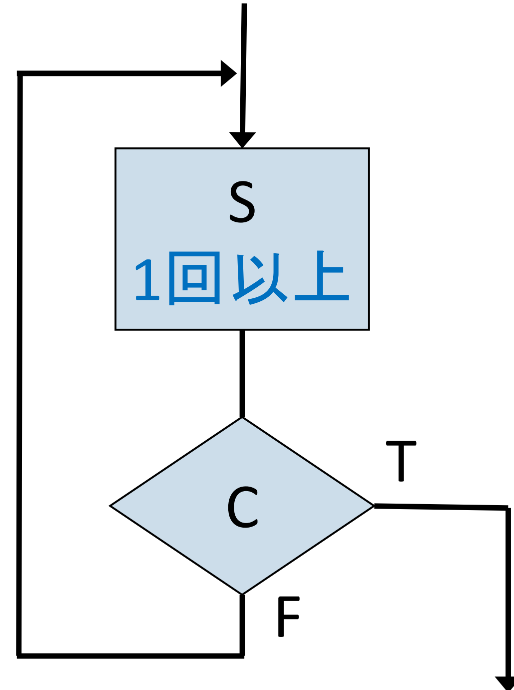
やや複雑な変換



必ずしもわかりやすくなるわけではない

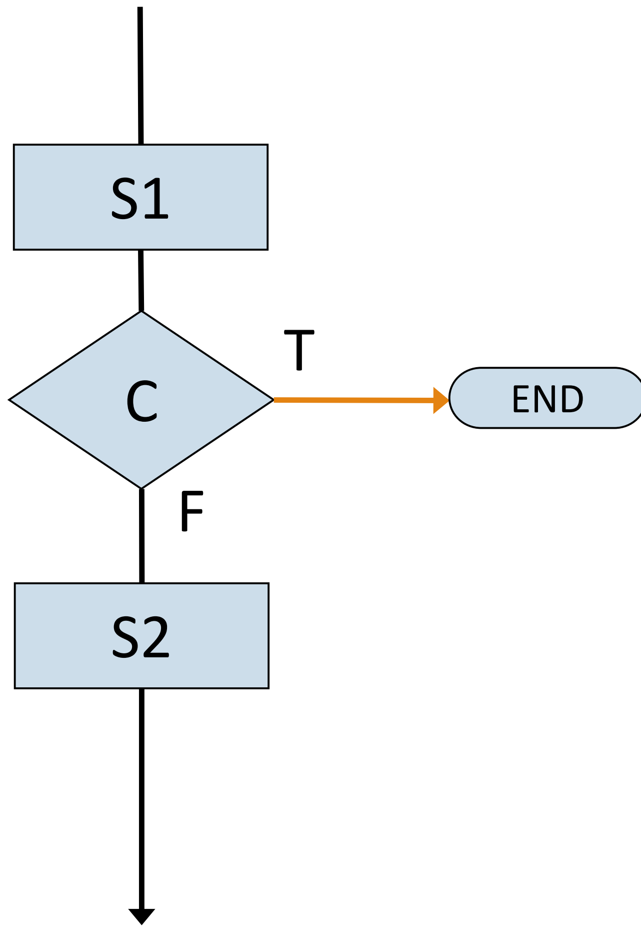
現実には後判定反復も許す





現実には return文も許す

return 文: 関数の出口に飛ぶ特別な GO TO 文

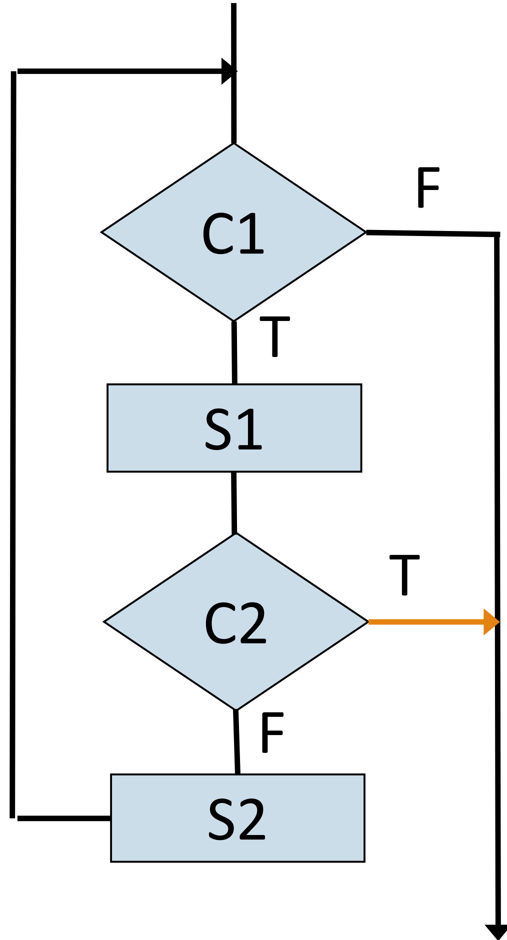



```
S1;
if C then return;
S2;
```

if 文の枝分かれが合流せず
出口が2カ所以上となるが
制御構造はあまり複雑化しない

break文も許す

break 文 : ループの出口に飛ぶ特別な GO TO 文



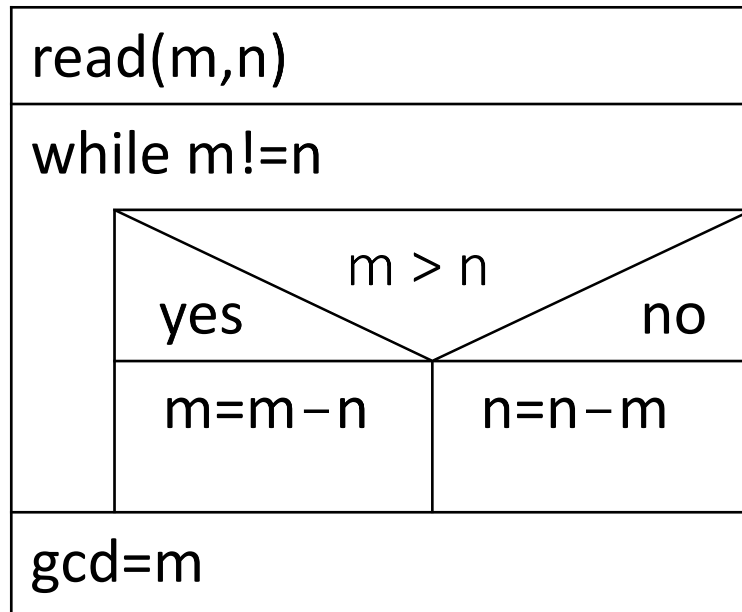

```
while C1 do
   S1;
   if C2 then break;
   S2;
```

ループからの出口が2カ所以上になるが制御構造はあまり複雑化しない

構造の図式(1/4):NSチャート

構造図式： 図式化することにより，
構造の全体を把握した後に，細部を読むことができる

NSチャート (Nassi-Shneiderman diagram)　積み木を積み上げる感じで表現

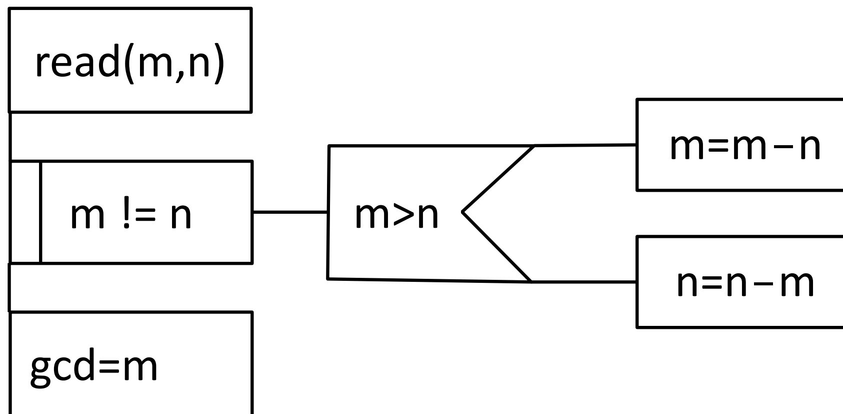



```
read(m,n);
while m!=n
   if m>n then m=m-n
           else n=n-m;
gcd=m;
```

構造の図式(2/4)：PAD

PAD (Problem Analysis Diagram)

木構造により表現
（日本人が考案し世界標準化）




```
read(m,n);
while m!=n
   if m>n then m=m-n
          else n=n-m;
gcd=m;
```

構造の図式(3/4)：インデント

字下げ (indentation) 細かな流儀がいろいろ。一貫して使うこと。

良い例

```
int main(void){
    scanf("%d %d", &m, &n);
    while(m!=n){
        if(m>n){
            m=m-n;
        }
        else{
            n=n-m;
        }
    }
    printf("%d¥n", m);
}
```

構造の図式(4/4)：インデント

悪い例

```
int main(void){
  scanf(”%d %d”, &m, &n);
  while(m!=n){
    if(m>n){
      m=m-n;
  }
  else{
n=n-m;
}
  }
  printf(”%d¥n”, m);
}
```

演習問題２

つぎのCプログラムの構造を
(1) NSチャート
(2) PAD
(3) ソースコードのインデント
により，それぞれ表示しなさい。

```
int print_id(char line[ ], int start){
while((start < strlen(line))
 && !alpha(line[start]))
start++;
if(start >= strlen(line)) return -1;
else {printf("%c", line[start]);
start++;
while((start < strlen(line)) &&
(alpha(line[start])||num(line[start])))
{
printf("%c", line[start]);
start++;}
printf("¥n");
return start;
}}
```