Procedural Programming
It's Back? It Never Went Away
@KevlinHenney
Brycgstow
procedure
procedural?
μονόλιθος
This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.

Doug McIlroy
In Mcllroy's summary, the hard part is his second sentence: Write programs to work together.

John D Cook
In the long run every program becomes rococo — then rubble.

Alan Perlis
1960s
1960s
SOFTWARE ENGINEERING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968
I began to use the term “software engineering” to distinguish it from hardware and other kinds of engineering; yet, treat each type of engineering as part of the overall systems engineering process.

Margaret Hamilton
SOFTWARE ENGINEERING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968
Define a subset of the system which is small enough to bring to an operational state [...] then build on that subsystem.
This strategy requires that the system be designed in modules which can be realized, tested, and modified independently, apart from conventions for intermodule communication.
The design process is an iterative one.

Report on a conference sponsored by the NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October
There are two classes of system designers. The first, if given five problems will solve them one at a time.
The second will come back and announce that these aren’t the real problems, and will eventually propose a solution to the single problem which underlies the original five.
This is the ‘system type’ who is great during the initial stages of a design project. However, you had better get rid of him after the first six months if you want to get a working system.
A software system can best be designed if the testing is interlaced with the designing instead of being used after the design.

Alan Perlis
proc is leap year = (int year) bool:
    skip;
Revised Report
on the Algorithmic Language

Algol 68

Edited by
A. van Wijngaarden, B. J. Mailloux,
Revised Report on the Algorithmic Language Algol 68


int long bool char void union short struct
proc is leap year = (int year) bool:
   false;

[] proposition leap year spec =
 ("Years not divisible by 4 are not leap years",
   void: (assert (not is leap year (1967))))
mode proposition = struct (string name, proc void test);
proc is leap year = (int year) bool:
    false;

[] proposition leap year spec =
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)));

test (leap year spec)
mode proposition = struct (string name, proc void test);

proc test = ([] proposition spec) void:
    for entry from lwb spec to upb spec do
        print (name of spec [entry]);
        test of spec [entry];
        print (new line)
    od;
proc is leap year = (int year) bool:
    year mod 4 = 0;

[] proposition leap year spec =
(
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967))))),
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968))))
);

test (leap year spec)
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0;

[] proposition leap year spec =
   (
    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)))))
    ,
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968)))))
    ,
    ("Years divisible by 100 but not by 400 are not leap years",
     void: (assert (not is leap year (1900))))
    );

test (leap year spec)
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0 or year mod 400 = 0;

[] proposition leap year spec =

    ("Years not divisible by 4 are not leap years",
     void: (assert (not is leap year (1967)))),
    ("Years divisible by 4 but not by 100 are leap years",
     void: (assert (is leap year (1968)))),
    ("Years divisible by 100 but not by 400 are not leap years",
     void: (assert (not is leap year (1900)))),
    ("Years divisible by 400 are leap years",
     void: (assert (is leap year (2000))))

);
proc is leap year = (int year) bool:
    year mod 4 = 0 and year mod 100 /= 0 or year mod 400 = 0;

[] proposition leap year spec =
    ("Years not divisible by 4 are not leap years",
     with (2018, 2001, 1967, 1), expect (false)),
    ("Years divisible by 4 but not by 100 are leap years",
     with (2016, 1984, 1968, 4), expect (true)),
    ("Years divisible by 100 but not by 400 are not leap years",
     with (2100, 1900, 100), expect (false)),
    ("Years divisible by 400 are leap years",
     with (2000, 1600, 400), expect (true)));

test (is leap year, leap year spec)
mode expect = bool;
mode with = flex [1:0] int;
mode proposition = struct (string name, with inputs, expect result);
proc test = (proc (int) bool function, [] proposition spec) void:
  for entry from lwb spec to upb spec do
    print (name of spec [entry]);
    string report := "", separator := " failed for ";
    [] int inputs = inputs of spec [entry];
    for value from lwb inputs to upb inputs do
      if
        bool expected = result of spec [entry];
        function (inputs [value]) /= expected
      then
        report += separator + whole(inputs[value], 0);
        separator := " ";
      fi
    od;
  print (if report = "" then (new line) else (new line, report, new line) fi)
  od;
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
        do
            print (name of spec [entry]);
            string report := "", separator := " failed for ";
            [] int inputs = inputs of spec [entry];
            for value from lwb inputs to upb inputs
                do
                    if
                        bool expected = result of spec [entry];
                        function (inputs [value]) /= expected
                    then
                        report +:= separator + whole(inputs[value], 0);
                        separator := " ";
                    fi
                od;
            print (if report = "" then (new line) else (new line, report, new line) fi)
        od;
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);

        string report := "", separator := " failed for ";
        [] int inputs = inputs of spec [entry];

        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report += separator + whole(inputs[value], 0);
                separator := " ";
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
proc test = (proc (int) bool function, [] proposition spec) void:
  for entry from lwb spec to upb spec do
    print (name of spec [entry]);
    string report := "", separator := " failed for ";
    [] int inputs = inputs of spec [entry];
    for value from lwb inputs to upb inputs do
      if
        bool expected = result of spec [entry];
        function (inputs [value]) /= expected
      then
        report +:= separator + whole(inputs[value], 0);
        separator := " "
      fi
    od;
    print (if report = "" then (new line) else (new line, report, new line) fi)
  od;
proc test = (proc (int) bool function, [] proposition spec) void:
    for entry from lwb spec to upb spec
    do
        print (name of spec [entry]);
        string report := "", separator := " failed for ";
        [] int inputs = inputs of spec [entry];
        for value from lwb inputs to upb inputs
        do
            if
                bool expected = result of spec [entry];
                function (inputs [value]) /= expected
            then
                report += separator + whole(inputs[value], 0);
                separator := " ";
            fi
        od;
        print (if report = "" then (new line) else (new line, report, new line) fi)
    od;
proc test = (proc (int) bool function, [] proposition spec) void:
  for entry from lwb spec to upb spec do
    print (name of spec [entry]);
    string report := "", separator := " failed for ";
    [] int inputs = inputs of spec [entry];
    for value from lwb inputs to upb inputs do
      if
        bool expected = result of spec [entry];
        function (inputs [value]) /= expected
      then
        report += separator + whole(inputs[value], 0);
        separator := " ";
      fi
    od;
    print (if report = "" then (new line) else (new line, report, new line) fi)
  od;
proc test = (proc (int) bool function, [] proposition spec) void:
  for entry from lwb spec to upb spec do
    print (name of spec [entry]);
    string report := "", separator := " failed for ";
    [] int inputs = inputs of spec [entry];
    for value from lwb inputs to upb inputs do
      if
        bool expected = result of spec [entry];
        function (inputs [value]) /= expected
      then
        report += separator + whole(inputs[value], 0);
        separator := " ";
      fi
    od;
    print (if report = "" then (new line) else (new line, report, new line) fi)
  od;
proc test = (proc (int) bool function, [] proposition spec) void:
  for entry from lwb spec to upb spec
    do
      print (name of spec [entry]);
      string report := "", separator := " failed for ";
      [] int inputs = inputs of spec [entry];
      for value from lwb inputs to upb inputs
        do
          if
            bool expected = result of spec [entry];
            function (inputs [value]) /= expected
          then
            report += separator + whole(inputs[value], 0);
          separator := " ";
          fi
        od;
      print ((report = "" | (new line) | (new line, report, new line)))
    od;
We instituted a rigorous regression test for all of the features of AWK. Any of the three of us who put in a new feature into the language [...] first had to write a test for the new feature.

Alfred Aho

There is no such question as testing things after the fact with simulation models, but that in effect the testing and the replacement of simulations with modules that are deeper and more detailed goes on with the simulation model controlling, as it were, the place and order in which these things are done.

Alan Perlis
As design work progresses this simulation will gradually evolve into the real system.

The simulation is the design.
Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program instability, program execution.

Categories: 4.2, 5.3, 5.24

Abstract:

For a number of years I have been familiar with the observation that the quality of programmers is decreasing as a function of the time of day. Based on this, I have come to the conclusion that the go to statement should be abolished (all high-level programming languages (i.e. everything except, perhaps, (dial) machine code). At that time I did not attach too much importance to this discovery. I now, however, consider my conscience for publication because it is now clear in very recent developments (in which the subject participated), I have been forced to change my mind.

My first reaction was that, although the programmer's activity ends when he has announced a correct program, the person taking place under control of the program is the true subject matter of the activity, but it is this function that has brought the observer to the conclusion that his dynamic behavior is not therefore the desired specification. Yet, once the program has been made, the "coding" of the corresponding process is delegated to the machine.

My second reaction was that our intellectual resources are much greater than the one state relations and that our powers to visualize process entities in time are relatively poorly developed. For both reasons we should do what programmers are aware of our limitations our attempt to shorten the conceptual gap between the state program and the dynamic process, to make the correspondence between the program (open circuit) and the process (closed circuit) as trivial as possible.

Let us now consider how we can characterize the process of a program. You may think about this question in a very concrete manner: suppose that a program, considered as a time succession of actions, is executed after an arbitrary state, what data we have to fix in order that we can fire the program until the very next step? If the program text is in a pure concatenation, my assignment statements (for the purposes of this discussion see it as the program text) is to test the next action of the program (let us call it a "transition" action description). Is it the absence of the next action statement that I can impose upon the program, the system to the last traces of the previous situation.

In the same way with "transition" action descriptions we must associate actions in time space, if we name the "transducer" action description an "innocent" action description. Let us now consider a postulate to a reliable place in the text a "transition" action.

When we include conditional clauses (If then else), alternative clauses (else if else-then), while clauses it is possible to introduce a conditional transducer (C. A. R. Hoare (1969) 101, 122, 224), in conditional expressions as introduced by J. McCarthy (1961). We can also restrict the transducer (Little 1972), the fact remains that the program of the process remains characterized by a single transducer.

As soon as we include in our language processor we must admit that a single transducer is not less efficient. In the case that a transducer points to the interior of a procedure body the dynamic process is only characterized when we also give to which all the procedure we refer. With the inclusion of procedures we can characterize the program of the process by means of a set of transducers, the length of this sequence being equal to the dynamic depths of procedure calling.

Let us now consider repetitive clauses (Like, while, Repeat A unless D). Logically speaking, such clauses are not equivalent, because they are not equivalent with the aid of naive procedures. For reasons of realism I do not wish to examine them. The essential difference in these cases is that there are implemented quite comfortably with present day finite state machines. On the other hand, the LISP clauses are "unlimited" to well equipped to retain our intellectual grasp on the processes generated.

With the view of the repetitive clauses transducers are no longer sufficient to describe the dynamic process. Such sentences as "when" a repetitive clause, however, we can associate a so-called "dynamic index," to the initial value of the corresponding current repetition. As repetition clauses (just in procedure calls) may be applied recursively, but not over the program of the process can always be uniquely characterized by a limited sequence of regular and/or dynamic indices. The main point is that the values of these indices are outside programer's control, they are generated neither by the write-up of his program nor by the execution of the program, not by they are used or not. They provide independent coordinates in which to describe the process of the program.

Why do we need such independent coordinates? The reason is that this seems to be in accordance with our purposes—are we not a program to estimate the values of variable only with respect to the process of the program? It is to avoid the erroneous thought of people. If people in an initially empty room, we can relieve this by increasing by one by one we see them entering the room. In the consecutive moment that we have seen someone entering the room but not yet performed the subsequent increase of 1, it would describe the number of people in the room minus one.

The independent use of the go to statement has an immediate consequence that it becomes very hard to find a meaningful set of coordinates in which to describe the process program. Luckily, people take into account as well as the value of some will close very well, but this is not the case here in this function that it is relative to the program that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the program structure by a singular finite set of actions performed since program starts. But it is not possible to establish a contradiction this is, of course, domain of the subject.

The difficulty is that such a few numbers, although important, is already troublesome. In such a coordinate system it becomes an extremely complicated task to divide all these points of actions, say, in equal number of person in some line means one.

The go to statement is a staple point in programming, it is with teaching an invitation to make a mistake of using a program. Once we disregard and appreciate the clauses considered as heading one's. It is not only that these clauses is considered as wrong, that they will not satisfy all needs, but whatever clauses are suggested can be interpreted to understand 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

Judged by whom my thinking has been influenced? It is fairly obvious that I am not influenced by Peter Landin and Christopher Strachey. Finally I should like to record my own contributions were quite discrete (I am sure) since I made numerous contributions to the study of the game statement to a number of them, but I have not been able to show (or at least part) it has been made by C. A. R. Hoare. In this case, Mark and Peter together make a remark in the same position in the same argument. "Like the rational, it proved the program's structure, in a manner that go to statements are and, with it at least, the introduction of a large number of labels in the program."

In 1961 (Göpel, personal) seem to have proved the dependences of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a process one, however, is not to be recommended. The resulting flow diagrams cannot be expected to be more transparent than the original one.

References:


Kaj Thumbnails: Technological University Emotions (1961), 505-51.

Communications of the ACM
**snowclone, noun**

- clichéd wording used as a template, typically originating in a single quote
- e.g., "X considered harmful", "These aren't the Xs you're looking for", "X is the new Y", "It's X, but not as we know it", "No X left behind", "It's Xs all the way down", "All your X are belong to us"
A Case against the GO TO Statement.

by Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am 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. Later I discovered why the use of the go to statement has such disastrous effects and did I become 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.
FUNCTION ISLEAP(YEAR)
  LOGICAL ISLEAP
  INTEGER YEAR
  IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
  IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
  IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
  10  ISLEAP = .FALSE.
      RETURN
  20  ISLEAP = .TRUE.
      END
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
    IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
    IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
    10   ISLEAP = .FALSE.
        RETURN
    20   ISLEAP = .TRUE.
        RETURN
END
FUNCTION ISLEAP(YEAR)
  LOGICAL ISLEAP
  INTEGER YEAR
  IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
  IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
  IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
  10  ISLEAP = .FALSE.
  GOTO 30
  20  ISLEAP = .TRUE.
  30  RETURN
END
FUNCTION ISLEAP(YEAR)
  LOGICAL ISLEAP
  INTEGER YEAR
  IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
  IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
  IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
  10   ISLEAP = .FALSE.
        GOTO 30
  20   ISLEAP = .TRUE.
        GOTO 30
  30   RETURN 
END
FUNCTION ISLEAP(YEAR)
  LOGICAL ISLEAP
  INTEGER YEAR
  IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
  IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
  IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
  10    ISLEAP = .FALSE.
       GOTO 30
  20    ISLEAP = .TRUE.
       GOTO 30
  30    CONTINUE
       RETURN
END
FUNCTION ISLEAP(Year)
   LOGICAL ISLEAP
   INTEGER YEAR
   IF (MOD(YEAR, 400) .EQ. 0) THEN
      ISLEAP = .TRUE.
   ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
      ISLEAP = .FALSE.
   ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
      ISLEAP = .TRUE.
   ELSE
      ISLEAP = .FALSE.
   END IF
END IF
END
FUNCTION ISLEAP(Year)
  LOGICAL ISLEAP
  INTEGER YEAR
  IF (MOD(YEAR, 400) .EQ. 0) THEN
    ISLEAP = .TRUE.
  ELSE IF (MOD(YEAR, 100) .EQ. 0) THEN
    ISLEAP = .FALSE.
  ELSE IF (MOD(YEAR, 4) .EQ. 0) THEN
    ISLEAP = .TRUE.
  ELSE
    ISLEAP = .FALSE.
  END IF
END IF
END
A goto completely invalidates the high-level structure of the code.
FUNCTION ISLEAP(YEAR)
    LOGICAL ISLEAP
    INTEGER YEAR
    IF (MOD(YEAR, 400) .EQ. 0) GOTO 20
    IF (MOD(YEAR, 100) .EQ. 0) GOTO 10
    IF (MOD(YEAR, 4) .EQ. 0) GOTO 20
    10  ISLEAP = .FALSE.
        RETURN
    20  ISLEAP = .TRUE.
        END
send(to, from, count)
register short *to, *from;
register count;
{
    register n=(count+7)/8;
    switch(count%8){
        case 0: do{ *to = *from++; 
        case 7: *to = *from++; 
        case 6: *to = *from++; 
        case 5: *to = *from++; 
        case 4: *to = *from++; 
        case 3: *to = *from++; 
        case 2: *to = *from++; 
        case 1: *to = *from++; 
            }while(--n>0);
    }
}
send(to, from, count)
register short *to, *from;
register count;
{
  register n=(count+7)/8;
  switch(count%8){
    case 0: do{ *to = *from++;
    case 7:     *to = *from++;
    case 6:     *to = *from++;
    case 5:     *to = *from++;
    case 4:     *to = *from++;
    case 3:     *to = *from++;
    case 2:     *to = *from++;
    case 1:     *to = *from++;
    }while(--n>0);
  }
}

I feel a combination of pride and revulsion at this discovery.

Tom Duff
send(to, from, count)
register short *to, *from;
register count;
{
register n=(count+7)/8;
switch(count%8){
case 0: do{ *to = *from++;
        case 7:     *to = *from++;
        case 6:     *to = *from++;
        case 5:     *to = *from++;
        case 4:     *to = *from++;
        case 3:     *to = *from++;
        case 2:     *to = *from++;
        case 1:     *to = *from++;
        while(--n>0);
    }while(0);

Many people have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.  

Tom Duff
def isLeapYear(year):
    return year % 4 == 0 && year % 100 != 0 || year % 400 == 0
def isLeapYear(year):
    
    year % 4 == 0 && year % 100 != 0 || year % 400 == 0

}
def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    else if (year % 100 == 0)
        return false
    else if (year % 4 == 0)
        return true
    else
        return false
}

def isLeapYear(year)
{
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
}
def isLeapYear(year):
    if (year % 400 == 0)
        true
    else if (year % 100 == 0)
        false
    else if (year % 4 == 0)
        true
    else
        false

def isLeapYear(year):
    if (year % 400 == 0)
        return true
    if (year % 100 == 0)
        return false
    if (year % 4 == 0)
        return true
    return false
def isLeapYear(year):
    if (year % 400 == 0):
        return true
    else if (year % 100 == 0):
        return false
    else if (year % 4 == 0):
        return true
    else:
        return false
def isLeapYear(year):
    if (year % 400 == 0):
        return true
    else if (year % 100 == 0):
        return false
    else if (year % 4 == 0):
        return true
    else:
        return false

def isLeapYear(year):
    if (year % 400 == 0):
        return true
    if (year % 100 == 0):
        return false
    if (year % 4 == 0):
        return true
    return false
def isLeapYear(year):
    if (year % 400 == 0):
        ...
    else if (year % 100 == 0):
        ...
    else if (year % 4 == 0):
        ...
    else:
        ...

def isLeapYear(year):
    if (year % 400 == 0):
        ...
    else if (year % 100 == 0):
        ...
    else if (year % 4 == 0):
        ...
    return False

def isLeapYear(year):
    if (year % 400 == 0):
        return true
    else if (year % 100 == 0):
        return false
    else if (year % 4 == 0):
        return true
    else:
        return false

def isLeapYear(year):
    if (year % 400 == 0):
        return true
    if (year % 100 == 0):
        return false
    if (year % 4 == 0):
        return true
    return false
def isLeapYear(year):
    {
        if (year % 400 == 0)
            true
        else if (year % 100 == 0)
            false
        else if (year % 4 == 0)
            true
        else
            false
    }
proc is leap year = (int year) bool:
    if year mod 400 = 0 then
        true
    elif year mod 100 = 0 then
        false
    elif year mod 4 = 0 then
        true
    else
        false
    fi;
isLeapYear year =
    if year `mod` 400 == 0 then
        True
    else if year `mod` 100 == 0 then
        False
    else if year `mod` 4 == 0 then
        True
    else
        False
function IsLeapYear(Year: Integer): Boolean;
begin
    if Year mod 400 = 0 then
        IsLeapYear := True
    else if Year mod 100 = 0 then
        IsLeapYear := False
    else if Year mod 4 = 0 then
        IsLeapYear := True
    else
        IsLeapYear := False
end;
STRUCTURED PROGRAMMING

O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE
One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare

"Hierarchical Program Structures"
sequence
selection
iteration
Main Program and Subroutine

The goal is to decompose a program into smaller pieces to help achieve modifiability. A program is decomposed hierarchically.

Len Bass, Paul Clements & Rick Kazman
Software Architecture in Practice
There is typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.
You cannot teach beginners top-down programming, because they don't know which end is up.

C A R Hoare
Everything should be built top-down, except the first time.

Alan Perlis
We propose [...] that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

David L Parnas

"On the Criteria to Be Used in Decomposing Systems into Modules"
An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects.

Barbara Liskov
"Programming with Abstract Data Types"
A programmer [...] is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation.

Barbara Liskov
"Programming with Abstract Data Types"
DEFINITION MODULE Stacks;

TYPE Stack;

PROCEDURE New(VAR self: Stack);
PROCEDURE Delete(VAR self: Stack);
PROCEDURE Push(self: Stack; top: ARRAY OF CHAR);
PROCEDURE Pop(self: Stack);
PROCEDURE Depth(self: Stack): CARDINAL;
PROCEDURE Top(self: Stack; VAR top: ARRAY OF CHAR);

END Stacks.
#ifdef __cplusplus
extern "C"
{
#endif

typedef struct stack stack;
stack * stack_new(void);
void stack_delete(stack *);
void stack_push(stack *, const char *);
void stack_pop(stack *);
size_t stack_depth(const stack *);
const char * stack_top(const stack *);
#endif
}
struct stack {
    const char ** items;
    size_t depth;
};

stack * stack_new(void) {
    stack * result = (stack *) malloc(sizeof(stack));
    result->items = (const char **) malloc(0);
    result->depth = 0;
    return result;
}

void stack_delete(stack * self) {
    free(self->items);
    free(self);
}

void stack_push(stack * self, const char * new_top) {
    self->items = (const char **) realloc(self->items, (self->depth + 1) * sizeof(char *));
    self->items[self->depth] = new_top;
    ++self->depth;
}

void stack_pop(stack * self) {
    self->items = (const char **) realloc(self->items, (self->depth - 1) * sizeof(char *));
    --self->depth;
}

size_t stack_depth(const stack * self) {
    return self->depth;
}

const char * stack_top(const stack * self) {
    return self->items[self->depth - 1];
}
extern "C"
{
    struct stack
    {
        std::vector<std::string> items;
    };

    stack * stack_new()
    {
        return new stack;
    }

    void stack_delete(stack * self)
    {
        delete self;
    }

    void stack_push(stack * self, const char * new_top)
    {
        self->items.push_back(new_top);
    }

    void stack_pop(stack * self)
    {
        self->items.pop_back();
    }

    size_t stack_depth(const stack * self)
    {
        return self->items.size();
    }

    const char * stack_top(const stack * self)
    {
        return self->items.back().c_str();
    }
}
Hamlet: To be, or not to be, that is the question.
Ophelia: 'Tis in my memory locked, and you yourself shall keep the key of it.
Hamlet: Yea, from the table of my memory I'll wipe away all trivial fond records.
One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"
begin
  ref (Book) array books(1:capacity);
  integer count;
  procedure Push(top);
  procedure Pop;
  boolean procedure IsEmpty;
  boolean procedure IsFull;
  integer procedure Depth;
  ref (Book) procedure Top;
  count := 0
end;
A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"
class Stack(capacity);
integer capacity;
begin
    ref (Book) array books(1:capacity);
    integer count;
    procedure Push(top); ...
    procedure Pop; ...
    boolean procedure IsEmpty; ...
    boolean procedure IsFull; ...
    integer procedure Depth; ...
    ref (Book) procedure Top; ...
    count := 0
end;
const newStack = () => {
  const items = []
  return {
    depth: () => items.length,
    top: () => items[0],
    pop: () => { items.shift(); },
    push: newTop => { items.unshift(newTop); },
  }
}
const newStack = () => {
  const items = []
  return {
    depth: () => items.length,
    top: () => items[items.length - 1],
    pop: () => { items.pop() },
    push: newTop => { items.push(newTop) },
  }
}
Concatenation is an operation defined between two classes $A$ and $B$, or a class $A$ and a block $C$, and results in the formation of a new class or block.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"
Concatenation consists in a merging of the attributes of both components, and the composition of their actions.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"
const stackable = base => {
  const items = []
  return Object.assign(base, {
    depth: () => items.length,
    top: () => items[items.length - 1],
    pop: () => { items.pop() },
    push: newTop => { items.push(newTop) }
  })
}
const newStack = () => stackable({})
const clearable = base => {
  return Object.assign(base, {
    clear: () => {
      while (base.depth())
        base.pop()
    }
  })
}
const newStack = () => clearable(stackable({}))
const newStack = () => compose(clearable, stackable)({})

const compose = (...funcs) =>
  arg => funcs.reduceRight((composed, func) => func(composed), arg)
Concept Hierarchies

The construction principle involved is best called *abstraction*; we concentrate on features common to many phenomena, and we abstract away features too far removed from the conceptual level at which we are working.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"
A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra.

Barbara Liskov
"Data Abstraction and Hierarchy"
What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"
const nonDuplicateTop = base => {
  const push = base.push
  return Object.assign(base, {
    push: newTop => {
      if (base.top() !== newTop)
        push(newTop)
    }
  })
}
tests = {

  'A non-empty stack becomes deeper by retaining a pushed item as its top': () => {
    const stack = newStack()
    stack.push('ACCU')
    stack.push('2018')
    stack.push('2018')
    assert(stack.depth() === 3)
    assert(stack.top() === '2018')
  },

  ...

}
const newStack = () => compose(clearable, stackable)({})

tests = {
  ...
  'A non-empty stack becomes deeper by retaining a pushed item as its top': () => {
    const stack = newStack()
    stack.push('ACCU')
    stack.push('2018')
    stack.push('2018')
    assert(stack.depth() === 3)
    assert(stack.top() === '2018')
  },
  ...
}
const newStack = () => compose(nonDuplicateTop, clearable, stackable)({})

tests = {
...
'A non-empty stack becomes deeper by retaining a pushed item as its top': () => {
    const stack = newStack()
    stack.push('ACCU')
    stack.push('2018')
    stack.push('2018')
    assert(stack.depth() === 3)
    assert(stack.top() === '2018')
},
...
};
What is wanted here is something like the following substitution property: If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \), then \( S \) is a subtype of \( T \).

Barbara Liskov

"Data Abstraction and Hierarchy"
We can build a complete programming model out of two separate pieces—the computation model and the coordination model.

David Gelernter + Nicholas Carriero
"Coordination Languages and their Significance"
Algorithms + Data Structures = Programs

Niklaus Wirth
Coordination + Computation = Programs
ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

*Make* also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978
Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared immutable data needs no synchronisation

Shared immutable data needs no synchronisation
The Synchronisation Quadrant

- **Mutable**
  - Unshared mutable data needs no synchronisation
  - Shared mutable data needs synchronisation

- **Immutable**
  - Unshared immutable data needs no synchronisation
  - Shared immutable data needs no synchronisation
**Procedural Comfort Zone**

- **Mutable**
  - Unshared mutable data needs no synchronisation
  - Shared mutable data needs synchronisation

- **Immutable**
  - Unshared immutable data needs no synchronisation
  - Shared immutable data needs no synchronisation
Procedural Comfort Zone

Mutable

Unshared mutable data needs no synchronisation

Shared mutable data needs synchronisation

Unshared immutable data needs no synchronisation

Shared immutable data needs no synchronisation

Procedural Discomfort Zone
Threads and locks — they’re kind of a dead end, right?

Bret Victor
"The future of programming"
So, I think if [...] we’re still using threads and locks, we should just, like, pack up and go home, ’cause we’ve clearly failed as an engineering field.

Bret Victor

"The future of programming"
Procedural Comfort Zone

- Unshared mutable data needs no synchronisation
- Shared mutable data needs synchronisation
- Unshared immutable data needs no synchronisation
- Shared immutable data needs no synchronisation
The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation.

David Gelernter + Nicholas Carriero

"Coordination Languages and their Significance"
The coordination model is the glue that binds separate activities into an ensemble.

David Gelernter + Nicholas Carriero

"Coordination Languages and their Significance"
single-threaded activity

no shared mutable state

coordination
Summary—what's most important.

1. We should have some way of coupling programs like garden hose—screw in another segment when it becomes clear it becomes necessary to manage data in another way.

This is the way of 10 also.

2. Our leader should be able to do link-loading and controlled establishment.

3. Our library filing system should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are system components) for buggering around with.

K. B. Kollroy
Oct. 11, 1964
bounded buffered asynchronous
N = \infty
unbounded
buffered
asynchronous
N = 1
bounded buffered asynchronous

promise future
N = 0
bounded
unbuffered
synchronous
Toutes choses sont dites déjà; mais comme personne n'écoute, il faut toujours recommencer.

André Gide
Everything has been said before; but since nobody listens, we must always start again.

André Gide