Programming and Paradigms:

Imperative Programming

Edward (Ned) Blurock

Definitions

IMPERATIVE PROGRAMMING

Wikipedia

In computer science,
imperative programming is a programming paradigm
that **uses statements that change a program's state**.

In much the same way that
the imperative mood in natural languages expresses
commands, an *imperative program* consists of
commands for the computer to perform.
Imperative programming focuses
on **describing how a program operates**..

https://en.wikipedia.org/wiki/Imperative_programming

Blog discussion

Imperative Programming is what most professional **programmers use in their day-to-day jobs**. It's the name given to languages like C, C++, Java, COBOL, etc. In imperative programming, you tell the computer what to do. "Computer, add x and y," or "Computer, slap a dialog box onto the screen." And (usually) the computer goes and does it. This is where most of us spend our lives, in looping structures and if-then-else statements and the like.

The focus is on **what steps the computer should take rather than what the computer will do** (ex. C, C++, Java).

An imperative language specfies **a series of instructions** that the computer executes in sequence (do this, then do that).

expressions describe **sequence of actions** to perform (associative)

http://stackoverflow.com/questions/602444/what-is-functional-declarative-and-imperative-programming

SKIP

Imperative Programming Course

- Translate basic functional idioms into imperative ones.
- Design simple loops, using invariants to explain why they work correctly.
- Use subroutines and modules to structure more complex programs.
- Specify a module as an abstract datatype, and formalise the relationship between that specification and an implementation.
- Design simple data structures.
- Understand the imperative implementation of some common algorithms.

- Basic imperative programming constructs: assignments, conditionals, procedures and loops. Comparison of imperative and functional programming. Examples.
- Method of invariants: correctness rules for while loops; proof of termination. Examples including summing an array, slow and fast exponentiation. Unit testing; debugging.
- Examples: string comparison, printing numbers in decimal.
- Binary search.
- Quicksort.
- Programming with abstract datatypes.
- Objects and classes as modules; specification; data abstraction.
- Reference-linked data structures: linked lists, binary trees.

SKIP

Caml Language Chapter

This style of programming is directly inspired by assembly programming.

You find it in the earliest general-purpose programming languages (Fortran, C, Pascal, etc.). In Objective Caml the following elements of the language fit into this model:
• modifiable data structures, such as arrays, or records with mutable fields;
• input-output operations;
• control structures such as loops and exceptions.

http://caml.inria.fr/pub/docs/oreilly-book/pdf/chap3.pdf

Comment

the choice between several programming styles
offers the greatest flexibility for writing algorithms,
which is the principal objective of any programming language.

Besides, a program written in a style
which is close to the algorithm used
will be simpler,
and hence will have a better chance
of being correct (or at least, rapidly correctable).

http://caml.inria.fr/pub/docs/oreilly-book/pdf/chap3.pdf

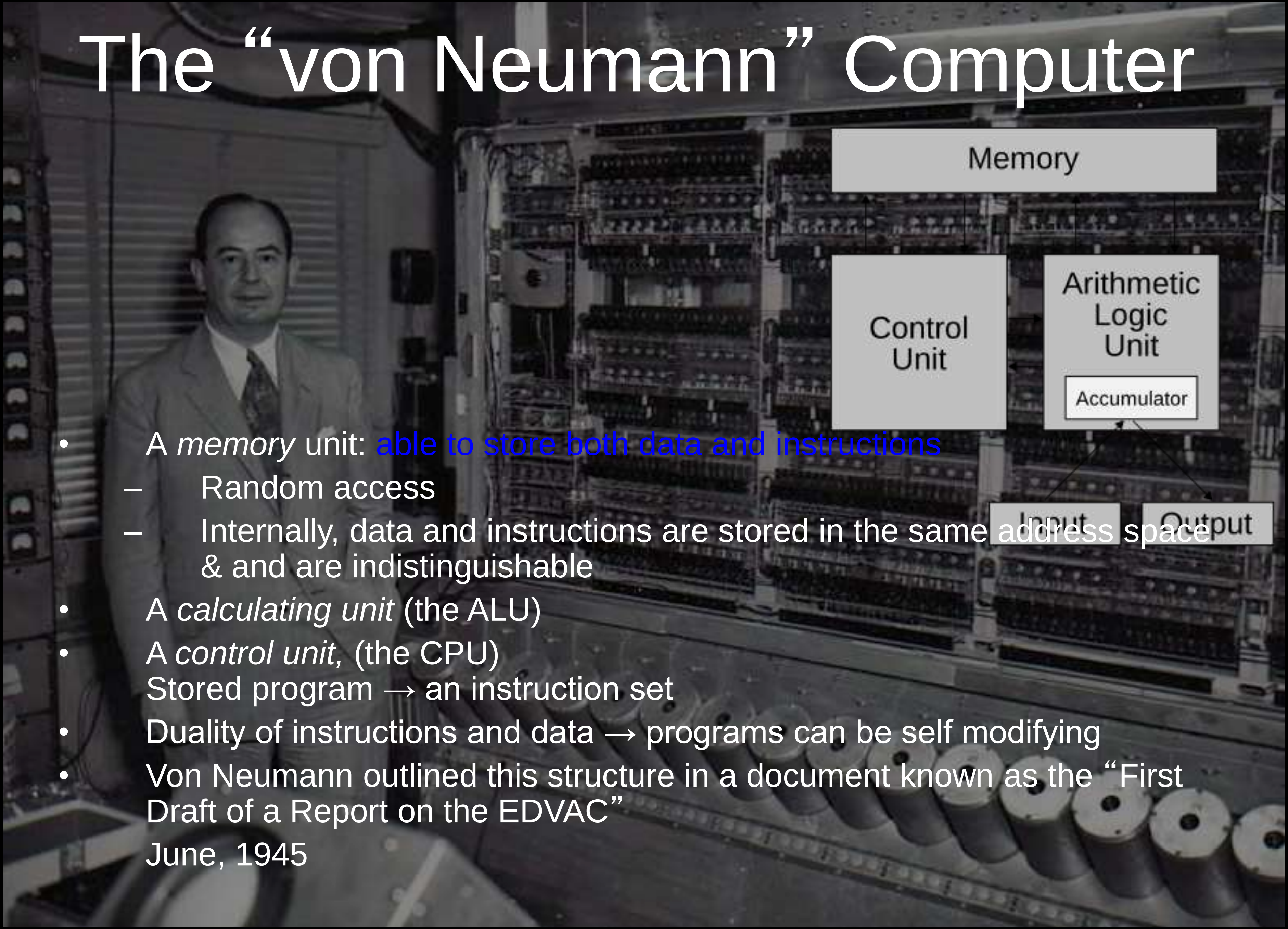
Von Neuman and Turing

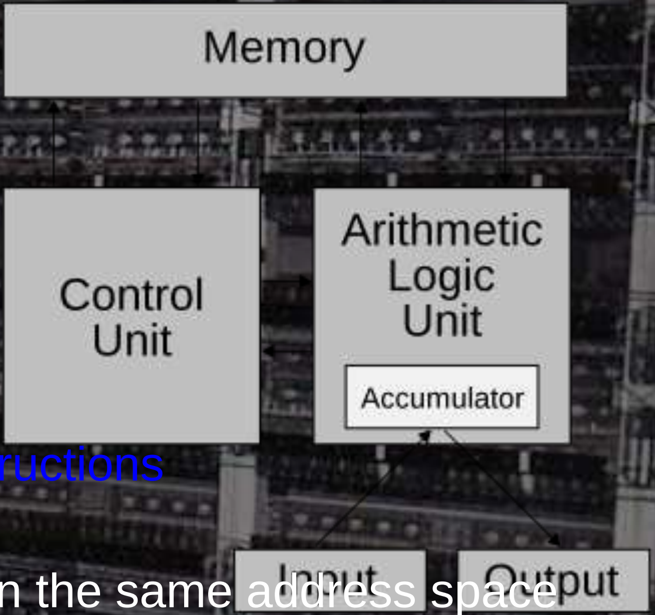
HISTORICAL REMARKS

Von Neumann Machines and Imperative Programming

- Commands in an imperative language are similar to the native machine instructions of traditional computer hardware – the von Neumann-Eckley model.
- John von Neumann: first person to document the basic concepts of stored program computers.
- Von Neumann was a famous Hungarian mathematician; came to US in 1930s & became interested in computers while participating in the development of the hydrogen bomb.

The “von Neumann” Computer





- A *memory* unit: able to store both data and instructions
  - Random access
  - Internally, data and instructions are stored in the same address space & and are indistinguishable
- A *calculating unit* (the ALU)
- A *control unit,* (the CPU)
  Stored program → an instruction set
- Duality of instructions and data → programs can be self modifying
- Von Neumann outlined this structure in a document known as the “First Draft of a Report on the EDVAC”
  June, 1945

The von Neumann Computer – Historical Background

- Earlier computers had fixed programs: they were hardwired to do one thing.
- Sometimes external programs were implemented with paper tape or by setting switches.
- Eckert and Mauchly considered stored program computers as early as 1944
- During WW II they designed & built the ENIAC (although for simplicity the stored program concept was not included at first)

The von Neumann Computer – Historical Background

ABC - Atanasof-Berry Computer at Iowa State University

- Later (with von Neumann), they worked on the EDVAC
- First stored program electronic computer: the Manchester ESSM (Baby)
  - Victoria University of Manchester
  - Executed its first program June 21, 1948
- A number of other stored program machines were under development around this time

History of Imperative Languages

- First imperative languages: assembly languages
- 1954-1955: Fortran (FORmula TRANslator) John Backus developed for IBM 704
- Late 1950's: Algol (ALGOrithmic Language)
- 1958: Cobol (COmmon Business Oriented Language) Developed by a government committee; Grace Hopper very influential.

Turing Completeness

- A language is Turing complete if it can be used to implement any algorithm.
- Central to the study of computability
- Alan Turing: A British mathematician, logician, and eventually computer scientist.

The Imitation Game

Imperative Programming

- Imperative languages are Turing complete if they support integers, basic arithmetic operators, assignment, sequencing, looping and branching.
- Modern imperative languages generally also include features such as
  - Expressions and assignment
  - Control structures (loops, decisions)
  - I/O commands
  - Procedures and functions
  - Error and exception handling
  - Library support for data structures

Origins of modern languages

IMPERATIVE LANGUAGES

FORTRAN

- `The IBM Mathematical FORmula TRANslating system'.
- Designed circa 1955 (by, amongst others, Backus).
- First successful attempt to improve on "assembly languages".
- Still widely used for numerical applications.
- Has been revised to take account of ideas on structured programming etc., however is decreasing in popularity

ALGOL 60

- `ALGOrithmic Language 1960'.
- Developed in 1950s through a joint European-American committee.
- First block-structured language.
- First language whose syntax was defined using BNF.
- Direct ancestor of most modern imperative languages.

BNF: Backus–Naur Form

a formal mathematical way to describe a language,

was developed by John Backus
(and possibly Peter Naur as well)
to describe the syntax of the Algol 60 programming language.

BNF: Backus–Naur Form

An integer:

```
<constant> ::= <digit>
<constant> ::= <constant> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Simple arithmetic expression:

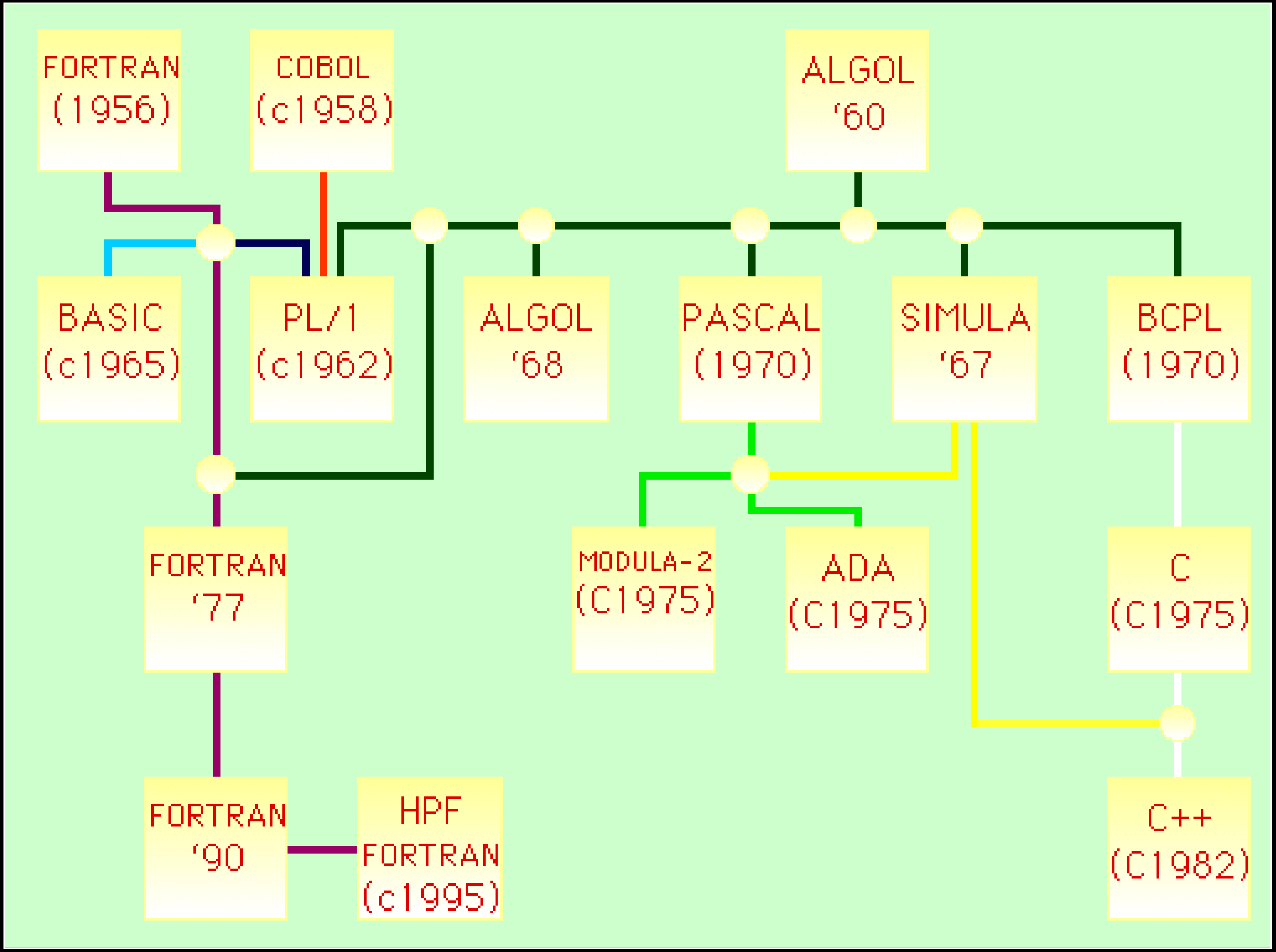
```
<expression> ::= <expression> + <expression>
 <expression> ::= <expression> - <expression>
<expression> ::= <expression> × <expression>
<expression> ::= ( <expression> )
<expression> ::= <constant>
```

COBOL

- `COmmon Business Oriented Language'
- Developed in 1950s through a committee consisting mainly of US computer manufacturers.
- Designed to process large data files and is therefore the most extensively used language for data processing.
- Has been revised several times, but revisions have been unable to take into account programming concepts such as structured programming and modularity.

SKIP

IMPERATIVE LANGUAGES "FAMILY TREE"

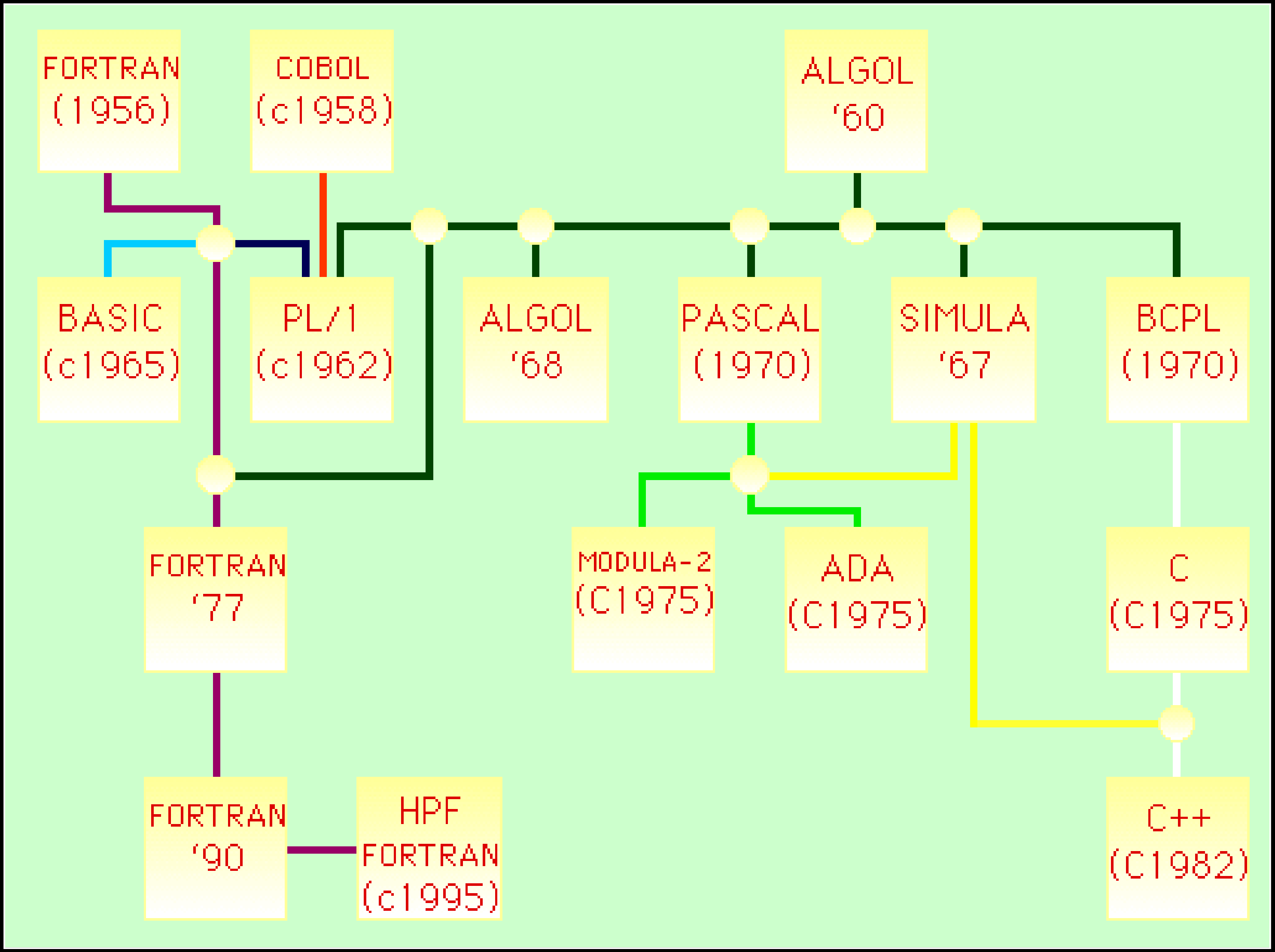


- **BASIC** - Beginner's All-purpose Symbolic Instruction Code. Teaching language made popular by the advent of the microcomputer in the mid 70s.
- **PL/1** - Intended to be an all purpose programming language to support both scientific and data processing applications. Did not live up to its aspirations.
- **Algol'68** - Successor to ALGOL'60. Proved less popular than its predecessor.
- **Pascal** - A popular block structured teaching language devised by Niklaus Wirth.
- **Simula'67** - Simulation language, significance of which is that it introduced the "class" concept.

Niklaus Wirth

(Pascal, Oberon)

IMPERATIVE LANGUAGES “FAMILY TREE”



- **Modula-2** - Pascal extended with modules.
- **Ada** - Pascal based language sponsored by the US Department of Defence.
- **BCPL** - Systems programming languages, significance of which is that it is a precursor to C.
- **C** - Systems programming language used in the development of the UNIX system. Is at a lower level than most imperative languages. Standardised in 1988 to ANSI C.
- **C++** - Object-oriented extension to C.
- **Java**

Variables: Locations and Values

- When a variable is declared, it is bound to some memory location and becomes its identifier
  - Location could be in global, heap, or stack storage
- l-value: memory location (address)
- r-value: value stored at the memory location identified by l-value
- Assignment: A (target) = B (expression)
  - Destructive update: overwrites the memory location identified by A with a value of expression B
    - What if a variable appears on both sides of assignment?

Variables and Assignment

- On the RHS of an assignment, use the variable's r-value; on the LHS, use its l-value
  - Example: x = x+1 == := <-
  - Meaning: "get r-value of x, add 1, store the result into the l-value of x"
- An expression that does not have an l-value cannot appear on the LHS of an assignment
  - What expressions don't have l-values?
    - Example: 1=x+1

l-Values and r-Values (1)

- Any expression or assignment statement in an imperative language can be understood in terms of l-values and r-values of variables involved
  - In C, also helps with complex pointer dereferencing and pointer arithmetic
- Literal constants
  - Have r-values, but not l-values
- Variables
  - Have both r-values and l-values
  - Example: x=x*y means "compute rval(x)*rval(y) and store it in lval(x)"

l-Values and r-Values (2)

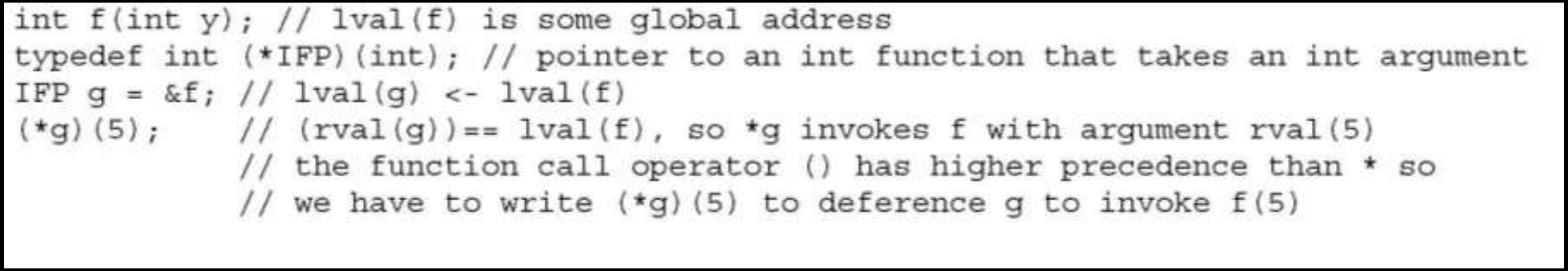
- Pointer variables
  - Their r-values are l-values of another variable
    - Intuition: the value of a pointer is an address
- Overriding r-value and l-value computation in C
  - &x always returns l-value of x
  - *p always return r-value of p
    - If p is a pointer, this is an l-value of another variable

```
int x = 5;  // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)
```

What are the values of p and x at this point?

l-Values and r-Values (3)

- Declared functions and procedures
  - Have l-values, but no r-values



```
int f(int y); // lval(f) is some global address
typedef int (*IFP)(int); // pointer to an int function that takes an int argument
IFP g = &f; // lval(g) <- lval(f)
(*g)(5);    // (rval(g))== lval(f), so *g invokes f with argument rval(5)
            // the function call operator () has higher precedence than * so
            // we have to write (*g)(5) to deference g to invoke f(5)
```

Modifiable Data Structures

variable bound to a value
keeps this value to the end of its lifetime.
You can only modify this binding with a redefinition
in which case we are not really talking about the "same" variable;
rather, a new variable of the same name
now masks the old one,
which is no longer directly accessible,
but which remains unchanged.
With modifiable values, you can change the value associated with a variable without having to redeclare the latter.
You have access to the value of a variable
for writing as well as for reading.

http://caml.inria.fr/pub/docs/oreilly-book/pdf/chap3.pdf

Order of Evaluation of Arguments

Order of evaluation of the operands of almost all C++ operators
(including the order of evaluation of function arguments in a function-call expression and the order of evaluation of the subexpressions within any expression)

is unspecified.

The compiler can evaluate operands in any order,
and may choose another order
when the same expression is evaluated again.

side effect

http://en.cppreference.com/w/cpp/language/eval_order

Order of Evaluation of Arguments

…there is no concept of left-to-right or right-to-left evaluation in C++.

This is not to be confused
with left-to-right and right-to-left associativity of operators:
the expression

f1() + f2() + f3()

is parsed as

(f1() + f2()) + f3()

due to left-to-right associativity of operator+,

but the function call to f3 may be evaluated
first, last, or between f1() or f2() at run time.

http://en.cppreference.com/w/cpp/language/eval_order

Order of Evaluation of Arguments

Undefined behavior

1) If a side effect on a scalar object is unsequenced relative to another side effect on the same scalar object, the behavior is undefined.

```
i = ++i + i++; // undefined behavior
i = i++ + 1; // undefined behavior
   (but i = ++i + 1; is well-defined)
f(++i, ++i); // undefined behavior
f(i = -1, i = -1); // undefined behavior
```

2) If a side effect on a scalar object is unsequenced relative to a value computation using the value of the same scalar object, the behavior is undefined.

```
cout << i << i++; // undefined behavior
a[i] = i++; // undefined behavior
```

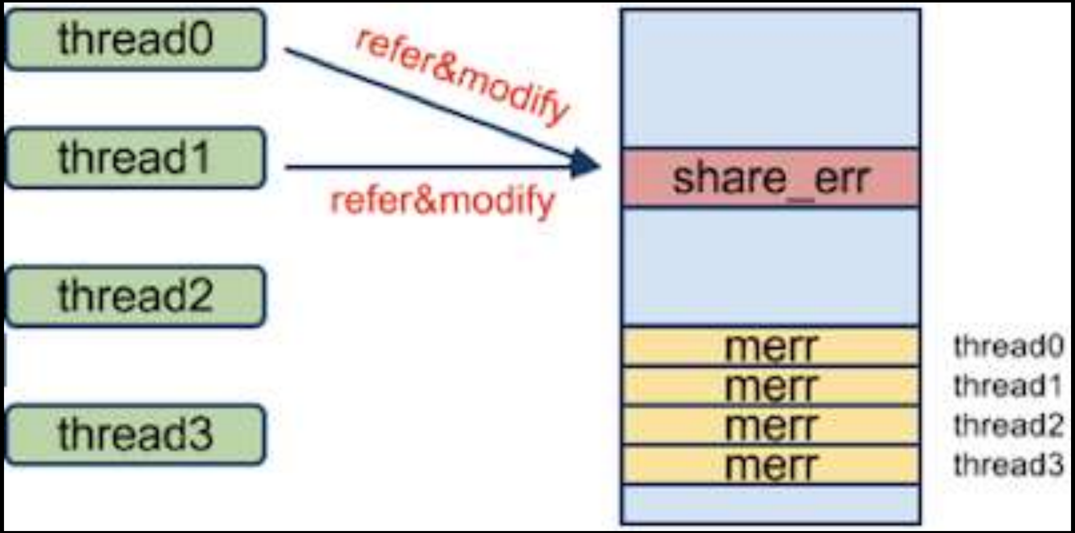
http://en.cppreference.com/w/cpp/language/eval_order

The Downfall of Imperative Programming

mutable

There is no doubt in my mind, and most experts agree, that concurrency and parallelism are the future of programming.

They are all failing because of one problem -- data races.



Imperative programs will always be vulnerable to data races because they contain mutable variables.

http://wiki.ccs.tulane.edu/index.php5/Data-Race_Condition

https://www.fpcomplete.com/blog/2012/04/the-downfall-of-imperative-programming time dependent error

X = Y + Z equation

Data Race

X = X + 1 assignment

A data race occurs when:

1. two or more threads in a single process access the same memory location concurrently, and
2. at least one of the accesses is for writing, and the threads are not using any exclusive locks
3. to control their accesses to that memory.

When these three conditions hold,
the order of accesses is non-deterministic,
and the computation may give different results
from run to run depending on that order.

http://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

Data Race Failure: Life Critical Systems

The accidents occurred when the high-power electron beam
was activated instead of the intended low power beam,

…Previous models had hardware interlocks in place to prevent this,
but Therac-25 had removed them,
depending instead on software interlocks for safety.

The software interlock could fail due to a race condition.

The defect was as follows:
a one-byte counter in a testing routine frequently overflowed;
if an operator provided manual input to the machine at the precise moment that this counter overflowed, the interlock would fail.

The value of a variable

SCOPE

Variable Scope

- The *scope* of a variable is the range of statements in a program over which it’s visible
- Typical cases:
  - Explicitly declared => local variables
  - Explicitly passed to a subprogram => parameters
  - The *nonlocal* variables of a program unit are those that are visible but not declared.
  - Global variables => visible everywhere.
- The scope rules of a language determine how references to names are associated with variables.
- The two major schemes are static scoping and dynamic scoping

Static Scope

- Also known as “lexical scope”
- Based on program text and can be determined prior to execution (e.g., at compile time)
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process:* search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*

Blocks

- A block is a section of code in which local variables are allocated/deallocated at the start/end of the block.
- Provides a method of creating static scopes inside program units
- Introduced by ALGOL 60 and found in most PLs.
- Variables can be hidden from a unit by having a "closer" variable with same name
  - C++ and Ada allow access to these "hidden" variables

Examples of Blocks

C and C++:

```
for (...) {
   int index;
   ...
   }
```

Ada:

```
declare LCL :
  FLOAT;
   begin
   ...
   end
```

Common Lisp:

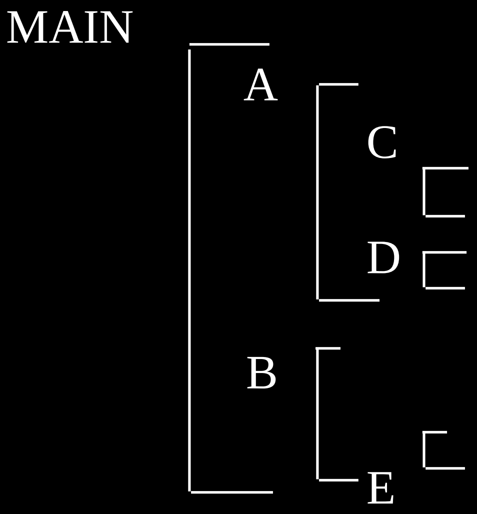
```
(let ((a 1)
      (b foo)
      (c))
   (setq a (* a a))
   (bar a b c))
```

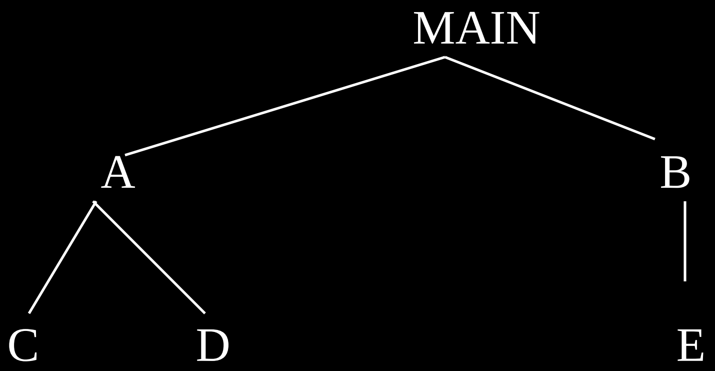
Static scoping example

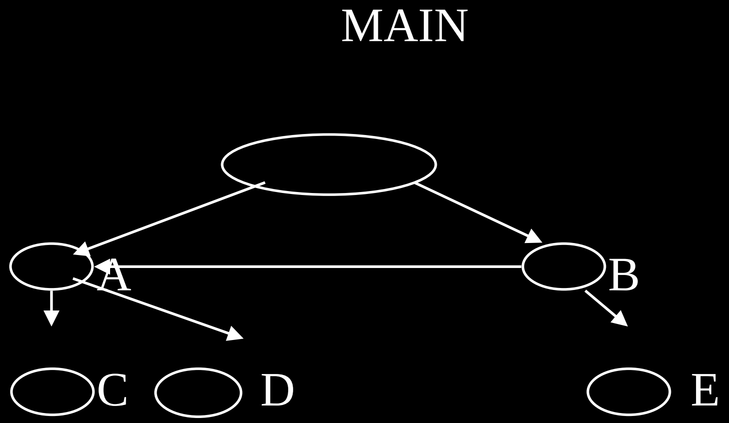
MAIN calls A and B

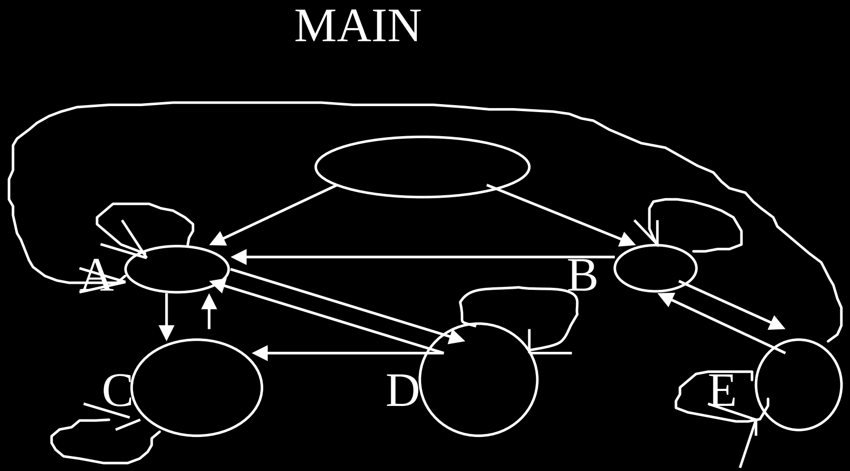
A calls C and D

B calls A and E









Evaluation of Static Scoping

Suppose the spec is changed so that D must now access some data in B

Solutions:

1. Put D in B (but then C can no longer call it and D cannot access A's variables)

2. Move the data from B that D needs to MAIN (but then all procedures can access them)

Same problem for procedure access!

Overall: static scoping often encourages many globals

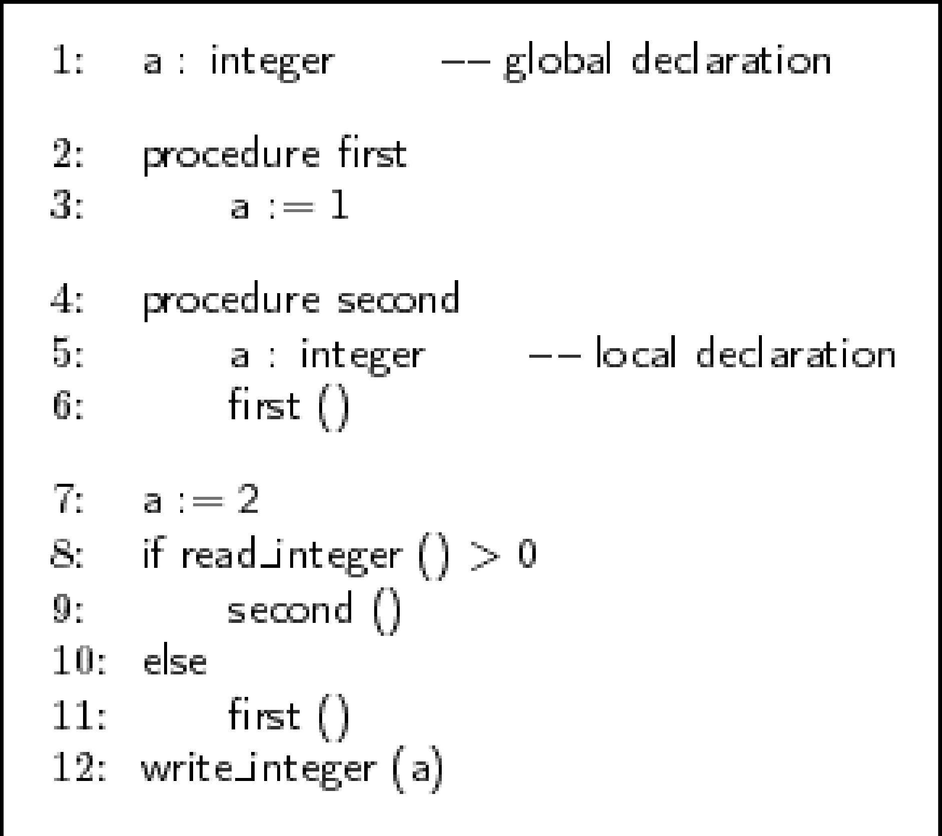
Dynamic Scope

- **Based on calling sequences of program units, not their textual layout (temporal versus spatial)**
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
- Used in APL, Snobol and LISP
  - implemented as interpreters rather than compilers.
- Consensus is that PLs with dynamic scoping leads to programs which **are difficult to read and maintain**.
  - Lisp switch to using static scoping as it's default circa 1980, though dynamic scoping is still possible as an option.

Dynamic Scope

- Bindings between names and objects depend on **the flow of control at run time**
  - The *current* binding is the one found most recently *during execution*

- Example
  - If the scoping is static, the output of the program is 1
  - If the scoping is dynamic, output is 1 or 2 depending on the value read at line 8 (>0 or <=0 respectively)



```
1:   a : integer        -- global declaration

2:   procedure first
3:        a := 1

4:   procedure second
5:        a : integer        -- local declaration
6:        first ()

7:   a := 2
8:   if read_integer () > 0
9:        second ()
10:  else
11:       first ()
12:  write_integer (a)
```

Accessing Variables with Dynamic Scope

- Two approaches:
  - Keep a stack (*association list*) of all active variables.
    - When you need to find a variable,
      - hunt down from top of stack.
    - This is equivalent to searching the activation records on the dynamic chain.
  - Keep a central table with one slot for every variable name.
    - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time. Otherwise, you'll need a hash function or something to do lookup.
    - Every subroutine changes the table entries for its locals at entry and exit.

Referencing Environments

- **The *referencing environment* of a statement is the collection of all names that are visible in the statement**
- In a **static scoped** language, that is the local variables plus all of the visible variables in all of the enclosing scopes.
- A subprogram is *active* if its execution has begun but has not yet terminated
- In a **dynamic-scoped** language, the referencing environment is the local variables plus all visible variables in **all active subprograms**. See book example (p. 185)

Variables in function calls

CALL BY VALUE OR REFERENCE

Call-by-Value (CBV)

- In *call-by-value* (*eager evaluation*), arguments to functions are fully **evaluated before** the function is invoked
  - This is the standard evaluation order that we're used to from C, C++, and Java

Call-by-Value in Imperative Languages

– What does this program print?

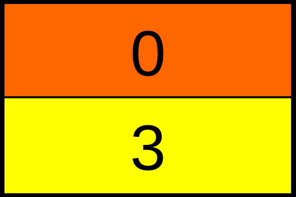
```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

– Prints 0

Call-by-Value in Imperative Languages, con't.

- Actual parameter is copied to stack location of formal parameter

```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

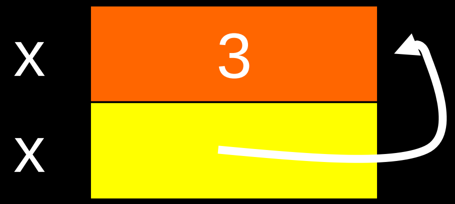


- Modification of formal parameter not reflected in actual parameter!

Call-by-Reference (CBR)

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
  - If the function writes to it the actual parameter is

```
void f(int x) {
 x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Reference (cont’d)

- Advantages
  - The entire argument doesn't have to be copied to the called function
    - It's more efficient if you’re passing a large (multi-word) argument
    - Can do this without explicit pointer manipulation
  - Allows easy multiple return values
- Disadvantages
  - Can you pass a non-variable (e.g., constant, function result) by reference?
  - It may be hard to tell if a function modifies an argument
  - What if you have *aliasing*?

Aliasing

- We say that two names are *aliased* if they refer to the same object in memory
  - C examples (this is what makes optimizing C hard)

```
    int x;
    int *p, *q; /*Note that C uses pointers to
        simulate call by reference */
    p = &x;  /* *p and x are aliased */
    q = p;   /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }
struct list *p, *q;
...
q = p;   /* *q and *p are aliased */
         /* so are p->x and q->x */
         /* and p->next->x and q->next->x... */
```

Call-by-Reference (cont’d)

- Call-by-reference is still around (e.g., C++/Java)
  - Older languages (e.g., Fortran, Ada, C with pointers) still use it
  - Possible efficiency gains not worth the confusion
  - “The hardware” is basically call-by-value

Evaluation Order Discussion

- Call-by-value is the standard for languages with side effects
  - When we have side effects, we need to know the order in which things are evaluated, otherwise programs have unpredictable behavior
  - Call-by-reference can sometimes give different results
  - Call-by-value specifies the order at function calls
- But there are alternatives to call by value and call by reference ...

SKIP

Call-by-Name (CBN)

- *Call-by-name* (lazy evaluation*)*
  - First described in description of Algol (1960)
  - Generalization of Lambda expressions (to be discussed later)
  - Idea simple: In a function:

```
Let add x y = x+y
add (a*b) (c*d)
```

Example:

```
add (a*b) (c*d) =
     (a*b) + (c*d) 🡐 executed function
```

Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively

  - But implementation: Highly complex, inefficient, and provides little improvement over other mechanisms, as later slides demonstrate

SKIP

Call-by-Name (cont'd)

- In *call-by-name*, arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y

let z = add (add 3 1) (add 4 1)
```

OCaml; cbv; arguments evaluated here

Haskell; cbn; arguments evaluated here

```
add x y = x + y

z = add (add 3 1) (add 4 1)
```

SKIP

Call-by-Name (cont'd)

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; eager; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; lazy; never evaluated because parameter is never used

SKIP

Cool Application of Lazy Evaluation

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build “infinite” data structures

```
let rec integers n = n::(integers (n+1))

let rec take n i = match i with
  h::t -> if n > 0 then h::(take (n-1) t)
                   else []

take 10 (integers 0)  (* infinite loop in cbv *)
```

How a program Is executed

CONTROL FLOW

Control Flow

- Basic paradigms for control flow:
  - **Sequencing**
  - **Selection**
  - **Iteration**
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling and Speculation
  - Nondeterminacy

Sequencing

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neuman
- In assembly, the only way to "jump" around is to use branch statements.
- Early programming languages mimicked this, such as Fortran (and even Basic and C).

The end of goto

- In 1968, Edsger Dijkstra wrote an article condemning the goto statement.
- While hotly debated after this, gotos have essentially disappeared from modern programming language.
- This is the advent of “structured programming”, a model which took off in the 1970’s. Emphasizes:
  - Top down design
  - Modularization of code
  - Structured types
  - Descriptive variables
  - Iteration

Dijkstra ("Go-to statement considered harmful")

Alternatives to goto

- Getting rid of goto was actually fairly easy, since it was usually used in certain ways.
  - Goto to jump to end of current subroutine: use return instead
  - Goto to escape from the middle of a loop: use exit or break
  - Goto to repeat sections of code: loops

Biggest need for goto

- Several settings are very useful for gotos, however.
  - Want to end a procedure/loop early (for example, if target value is found).
    - Solution: **break or continue**
  - Problem: What about "bookkeeping"? We' re breaking out of code which might end a scope - need to call desctructors, deallocate variables, etc.
  - Adds overhead to stack control - must be support for "unwinding the stack"

Breaks
Programming practice

```
// Simple loop with an early exit

for (;;)  {
   int   ch;

   ch = read();
   if (ch == EOF)
      break;   // With a loop escape

   parse(ch);
}
```

```
// Simple loop with no loop escape mechanism
bool  incomplete = true;
while (incomplete) {
   int   ch;

   ch = read();
   if (ch == EOF)
      incomplete = false; // Without a loop escape
   else
      parse(ch);
}
```

Biggest need for goto

- Another example: **exceptions**
- Goto was generally used as error handling, to exit a section of code without continuing
- Modern languages generally throw and catch exceptions, instead.
  - Adds overhead
  - But allows more graceful recovery if a section of code is unable to fulfill its **contract**.

Sequencing

- Blocks of code are executed in a sequence.
- Block are generally indicated by { … } or similar construct.
- Interesting note: without side effects (as in Agol 68), blocks are essentially useless - the value is just the last return
- In other languages, such as Euclid and Turing, functions which return a value are not allowed to have a side effect at all.
  - Main advantage: these are idempotent - any function call will have the same value, no matter when it occurs
- Clearly, that is not always desirable, of course. (Think of the rand function, which should definitely not return the same thing every time!)

Selection

- Selection: introduced in Algol 60
  - sequential if statements

```
 if ... then ... else
 if ... then ... elsif ... else
```

  - Lisp variant:

```
(cond
          (C1) (E1)
          (C2) (E2)
          ...
          (Cn) (En)
             (T)  (Et)
 )
```

Selection

- Jump is especially useful in the presence of **short-circuiting**

  (minimal evaluation of boolean expressions)

  the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression

```
if ((A > B) and (C > D)) or (E <> F) then
   then_clause
  else
   else_clause
```

Selection

- Code generated w/o short-circuiting (Pascal)

```
        r1 := A                  -- load
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 := r2 > r3
        r1 := r1 & r2
        r2 := E
        r3 := F
        r2 := r2 $<>$ r3
        r1 := r1 $|$ r2
        if r1 = 0 goto L2
L1:     then_clause      -- label not actually used
        goto L3
L2:     else_clause
L3:
```

Selection

- Code generated w/ short-circuiting (C)

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
L4:     r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:     then_clause
        goto L3
L2:     else_clause
L3:
```

Selection: Case/switch

- The case/switch statement was introduced in Algol W to simplify certain if-else situations.
- Useful when comparing the same integer to a large variety of possibilities:
  - i := (complex expression)

    if i == 1: …

    elsif i in 2,7: …
  - Case (complex expression)

    1: …

    2-7: …

Selection: Case/switch

- While it looks nicer, principle reason is code optimization.
  - Instead of complex branching, just loads possible destinations into simple array.
- Additional implementations:
  - If set of labels is large and sparse (e.g. 1, 2-7, 8-100, 101, 102-105, …) then can make it more space efficient using hash tables or some other data structure.

Iteration

- Ability to perform some set of operations repeatedly.
  - Loops
  - Recursion
- Can think of iteration as the only way a function won’t run in linear time.
- In a real sense, this is the most powerful component of programming.
- In general, loops are more common in imperative languages, while recursion is more common in functional languages.

Iteration

- Enumeration-controlled: originated in Fortran
  - Pascal or Fortran-style for loops

```
do i = 1, 10, 2
    …
 enddo
```

  - Changed to standard for loops later, eg Modula-2

```
FOR i := first TO last BY step DO

        …

END
```

Iteration: Some issues

- Can control enter or leave the loop other than through enumeration mechanism?
  - Usually not a big deal - break, continue, etc. (NOT goto.)
- What happens if the loop body alters variables used to compute end-of-loop condition?
  - Some languages only compute this once. (Not C.)
- What happens if the loop modifies the index variable itself?
  - Most languages prohibit this entirely, although some leave it up to the programmer.
- Can the program read the index after the loop has been completed, and if so, what is its value?
  - Ties into issue of scope, and is very language dependent.

KISS - Keep It Simple, Stupid!

Iteration: Loops in C

- The for loop in C is called a combination loop - it allows one to use more complex structures in the for loop.
- Essentially, for loops are almost another variant of while loops, with more complex updates and true/false evaluations each time.
- Operator overloading (such as operator++) combined with iterators actually allow highly non-enumerative for loops.
- Example:

```
for (list<int>::iterator it = mylist.begin(); it !=
  mylist.end(); it++) {
  …
}
```

Iteration: iterator based loops

- Other languages (Ruby, Python, C# etc.) require any container to provide an iterator that enumerates items in that class.
- This is extremely high level, and relatively new.
- Example:

```
for item in mylist:
   #code to look at items
```

Iteration: logically controlled loops

- While loops are different than the standard, Fortran-style for loops, since no set number of enumerations is predefined.
- These are inherently strong - closer to if statements, in some ways, but with repetition built in also.
- Down side: Much more difficult to code properly, and more difficult to debug.
- Code optimization is also (in some sense) harder - none of the for loop tricks will work.