Textbook: https://plus.pearson.com/
xrandr --output DP-2 --brightness 0.4
Class: Operating System
-OS
	-Before an OS, humans control when a program run with boomer gigantic computers.
	-So people wrote software to control the timing of how a computer runs programs.
	-Layer of OS onion be like
		-Computer Hardware is inside the OS, OS is inside the Application Software.
		-A VM would be on the Application Software that runs an OS. 6.1: Data Types
Class: Unix & C Crap
-Terminal commands
	-"pwd": Present Working Directory
	-"ls": List all files
		-Ubuntu: blue is a folder, white/extension is a file.
		-"ls -l": List all files in a list and display permissions and date
			-"ls -la": additionally shows hidden files.
	-"cd": Change Directory
		-"cd ~": cd to /home
		-"cd /": cd to root
		-"cd /[path]" starts at root, while "cd [path]" continues from pwd
		-"cd .." goes back out to the parent of pwd.
			-"cd ../.." goes back twice
			-"cd ../../[path]" goes back twice from the path's dir
		-"cd ." is move to the current directory. Useful when running a program.
	-"cp [copy from path] -copy to path]": Copy
	-"diff [1st file] [2nd file]" compares & show differences
	-"mv [move from path] [move to path]" Move/Rename files.
	-"rm [path]": Remove/Delete
	-"mkdir [new folder]": Make new folder
	-"rmdir [folder]": Delete folder
	-"cat [file/path]": Called Concatenate, it reads and display content of a file (prob text only)
	-"ssh [login]@[url or whatever it is]" ssh daveed@cssgate.insttech.washington.edu
	-Tips:
		-TAB is OP, autofill.
		-When there's no error in a command, it usually doesn't have an output message.
		-If you want to run a application, use "./[file/path]" when pwd
-C
	-Procedural Language

HelloWorld.c > main(int, char * [])

```c
#include <stdio.h>

int main(int argc, char* argv[]) {

    printf("HI\n");

}
```

		-"char*": The asterisk means a pointer to the char variable idk wat?
		-#include is kinda like import, idk yet.
	-With GCC app: "gcc [path]" Compiles the C source code.
		-The first compiler error will hide the others if there exist more.
Chapter 6: Data Type **Microsoft Word - CMPS401ClassNotesChap06.doc (southeastern.edu)**
-Type System
	-Data type: defines a collection of data values and a set of predefined operations on those values.
		-int, float, char, etc., those are the different types and different languages have different identifiers for them.
		-Helps error detection
		-Provides for program modularization
		-Provides documentation, easier to trace types/objects than memory addresses
	-Descriptor: The collection of variable attributes. Used for type checking and building the code for the allocation and deallocation operations.
		-Used during compiling if static, and used during run time if dynamic.
		-Ex: A descriptor for String needs 3 fields. Name of the type, length (if static), and address of the first char.
-Primitive type
	-Def: Data types that are NOT defined in terms of other types.
	-Integer
		-Eww 371 crap. 2s Complement Bitstring.
		-Some languages like Java have smaller/larger types of ints.
		-Most of the time, the hardware directly supports them.
	-Floating Point
		-Eww 371 crap. IEEE single/double precision.
			-Precision is the accuracy of the fractional part of a value, measured as the number of bits.
			-Range is a combination of the range of fractions and, more important, the range of exponents.
		-Languages use whatever is supported by hardware.
	-Complex
		-"a+bi", 2 floats for each part.
	-Decimal
		-Exact decimal numbers.
			-Ex: 0.1 can't be represented with 100% accuracy by floats.
		-Like chars of a string, "binary coded decimal" (BCD) takes a byte to store 1 (or 2) digit. (4 digits = 24 bits)
	-Boolean
		-1 Bit, T or F.
	-Character
		-ASCII (8 bit/char), Unicode (16 bit/char)
-String
	-Sequence of characters
	-Different languages debate if String is immutable and if they are primitives/object/array
		-C string manipulation is unsafe. They don't know the length of one and can write over wrong memory addresses.

-Java String is immutable. Its value are constant strings, while StringBuffer acts more like an array and is changeable.

-Descriptors (ie C uses compile-time & Java uses run-time)

Compile-time descriptor for static strings

Static string
Length
Address

Run-time descriptor for limited dynamic strings

Limited dynamic string
Maximum length
Current length
Address

-Special case: C & C++ knows when a String ends by a null value at the end of the sequence of memory address. (kinda like assembly)

-Most of the time, there's a Substring function

-Concatenation

-pattern matching (ie "\n")

-Enumertion

-Group of enumeration constants, used to define a group of constants

-Different Language debate how to implement

- Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
- Are enumeration values coerced to integer?
- Are any other types coerced to an enumeration type?

-Coercing them in to int is rarely used and can be lead to mistakes.

-Array

-An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensioned arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?

-Two distinct types are involved in an array type: the element type and the type of the subscripts.

-"array_name(subscript_value_list) → element" is usually how arrays work

-Back then, parenthesis were the only char usable, no []. It's poop readility because method calls are usually () too.

-Subscripts aka Indices

-Most older lang don't check array out of bounds.

-Some lang (Python) don't need contiguous indices

-Array aka Finite Mapping

-Subscript Binding

1. A **static array** is one in which the subscript ranges are statically bound and storage allocation is static.
   - Advantages: efficiency "No allocation & deallocation."
2. A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at elaboration time during execution.
   - Advantages: Space efficiency. A large array in one subprogram can use the same space as a large array in different subprograms.
3. A **stack-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic "during execution." Once bound they remain fixed during the lifetime of the variable.
   - Advantages: Flexibility. The size of the array is known at bound time.
4. A **fixed heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, but they are both fixed after storage is allocated.
   The bindings are done when the user program requests them, rather than at elaboration time, and the storage is allocated on the heap, rather than the stack.
5. A **heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, and can change any number of times during the array's lifetime.
   - Advantages: Flexibility. Arrays can grow and shrink during program execution as the need for space changes.

-Java non-generic arrays are heap-dynamic.

Arrays declared in C & C++ function that includes the `static` modifier are **static**.
Arrays declared in C & C++ function without the `static` modifier are **fixed stack-dynamic arrays**.
Ada arrays can be **stack dynamic**:

C & C++ also provide **fixed heap-dynamic arrays**. The function `malloc` and `free` are used in C. The operations `new` and `delete` are used in C++.
In Java all arrays are **fixed heap dynamic arrays**. Once created, they keep the same subscript ranges and storage.
C# provides **heap-dynamic arrays** using an array class `ArrayList`.

-Rectangular and Jagged arrays

-Rectangle array is like normal m x n elements, while Jagged is kinda like histogram type beat.

-C, C++, Java doesn't support rectangular (but u can still make one).

-Not supported: myArray[3][7]

-Supported: myArray[3, 7]

-Slices

-Some substructure of an array

-Ex: the first row in a 2D array

-Implementation

-Access function for single-dimensional arrays: address (list[k]) = address (list[0]) + k * element_size

-Access function for single-dimensional arrays (statically bounded): address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * element_size)

-Descriptor

A compile-time descriptor for a multidimensional array

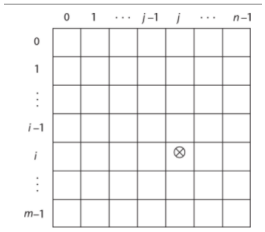
Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

Compile-time descriptor for single-dimensioned arrays

Array
Element type
Index type
Index lower bound
Index upper bound
Address

-2D Array: Row Major Order (Most Languages)

The location of the `[i,j]` element in a matrix



3 4 7

6 2 5

1 3 8 ==> 3, 4, 7, 6, 2, 5, 1, 3, 8 ==> (in memory) 3, 6, 1, 4, 2, 3, 7, 5, 8

-Location (a[i,j]) = address of a [1,1] + ((i - 1) * n + (j - 1)) * element_size

-Associative Arrays

-BRUHHH Its just Map/Table/Dictionary

-Unordered collection where keys gets you values.

-Keys are hashed into index

-Record

-An aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

-Basically, a data type that stores other different data types as fields.

- What is the syntactic form of references to fields?
- Are elliptical references allowed?

-C, C++, and C#, records are supported with the struct.

-Java record class https://howtodoinjava.com/java14/java-14-record-type/

A compile-time descriptor for a record

	Record
Field 1	Name
	Type
	Offset
⋮	⋮
Field n	Name
	Type
	Offset
	Address

-Tuple

-A data type that is similar to a record, except that the elements are not named.

-Doesn't have to be of same type

-Index begins at 1

-Python's Tuple

-myTuple = (3, 5.8, 'apple')

-myTuple[1] returns 3

-F# Ex: Making 3 variables using 3 items in tuple

```
let tup = (3, 5, 7);;
let a, b, c  = tup;;
```

-List

-IDK what? It just describe different array like thing, some lang allows different types for elems.

-Union

-A type whose variables may store different type values at different times during program execution.

-Very scary as you can't type check without discriminated union (a tag or something that identify type).

-Opposite of discriminated union is free union, which is C & C++.

- **Fortran, C, and C++** provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminanted union.*
- Supported by **Ada**

-Java & C# doesn't support

-Pointer

-One in which the variables have a range of values that consists of memory addresses and a special value, nil.

-A pointer can be used to access a location in an area where storage is dynamically allocated called a heap.

-Different Types of pointers:

-Heap-dynamic variables: Variables that are dynamically allocated from the heap.

-Anonymous variables: Variables without names

stores a memory address	pointer variable
variable allocated from the heap	heap-dynamic variable
variable without a name	anonymous variable
stores a reference to some other variable	reference variable
stores data	value type

-Pointer is aka Reference types, while Scalar Variables are Value types.

-Design Issues:

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable (the value a pointer references)?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

-Pointer Operations

-Assignment & Dereferencing

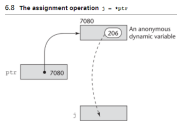
-Assignment sets a pointer var's value to some useful address.

-Dereferencing takes a reference through one level of indirection. (Accessing a memory location through a pointer variable)

If `ptr` is a pointer var with the value 7080, and the cell whose address is 7080 has the value 206, then the assignment

```
j = *ptr
```

sets j to 206.



In C and C++, the asterisk (*) denotes the **dereferencing** operation, and the ampersand (&) denotes the operator for producing the **address of** a variable. For example, in the code

```
int *ptr;
int count, init;
...
ptr = &init;
count = *ptr
```

-Dangling Pointers/Reference

-A pointer points to a heap-dynamic variable that has been deallocated.

- Dangling pointers are dangerous for the following reasons:
  - The location being pointed to may have been allocated to some new heap-dynamic var. If the new var is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
  - Even if the new one is the same type, its new value will bear no relationship to the old pointer's dereferenced value.
  - If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.
  - It is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.
- The following sequence of operations creates a dangling pointer in many languages:

  a. Pointer `p1` is set to point at a new heap-dynamic variable.
  b. Set a second pointer `p2` to the value of the first pointer p1.
  c. The heap-dynamic variable pointed to by p1 is explicitly deallocated (setting p1 to nil), but p2 is not changed by the operation. P2 is now a dangling pointer.

-Lost Heap-Dynamic Variables (Memory Leak)

-A heap-dynamic variable that is no longer referenced by any program pointer "no longer accessible by the user program."

- Such variables are often called **garbage** because they are not useful for their original purpose, and also they can't be reallocated for some new use by the program.
- Creating Lost Heap-Dynamic Variables:

  a. Pointer p1 is set to point to a newly created heap-dynamic variable.
  b. p1 is later set to point to another newly created heap-dynamic variable.
  c. The first heap-dynamic variable is now inaccessible, or lost.

- The process of losing heap-dynamic variables is called **memory leakage**.

-Reference Type

-A pointer refers to an address in memory, while a reference refers to an object or a value in memory.

-Pointers in C & C++

- **Extremely flexible** but must be used with care.
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible in C and C++ makes their pointers **more interesting** than those of the other programming languages.
- Unlike the pointers of Ada, which can only point into the heap, C and C++ pointers can point at virtually any variable **anywhere** in memory.
- Explicit dereferencing and address-of operators
- In C and C++, the asterisk (*) denotes the **dereferencing** operation, and the ampersand (&) denotes the operator for producing the **address of** a variable. For example, in the code

```
int *ptr;
int count, init;
…
ptr = &init;
count = *ptr
```

- the two assignment statement are equivalent to the single assignement

```
count = init;
```

- Example: Pointer Arithmetic in C and C++

```
int list[10];
int *ptr;
ptr = list;

*(ptr+5)   is equivalent to list[5]   and  ptr[5]
*(ptr+i)   is equivalent to list[i]   and  ptr[i]
```

- Domain type need not be fixed (**void ***)
- void * can point to **any** type and can be type checked (cannot be de-referenced)

-Solutions to Dangling Pointers
-Tombstones & Lock and Keys
-Heap Management / Garbage Collection
-Reference Counters (Eager Approach): reclamation is incremental and is done when inaccessible cells are created
-Keeps count of pointers to a cell, when 0, garbage ready.
-Mark-Sweep (Lazy Approach): reclamation occurs only when the list of available space becomes empty
-Sweep and check garbage after all cells are used up.

-Optional Types
-Ie reference that points to null and special types wow

-Type Checking
-The activity of ensuring that the operands of an operator are of Compatible Types.
-If not compatible, it is a Type Error
-Automatic conversion is called coercion.
-If variable types are not static you need Dynamic Type Checking during runtime.

-Strong Typing
-A programming language is strongly typed if type errors are always detected.
-C & C++ have union types that are not checked, so they are not a strongly typed language.
-Java and C# are nearly strongly typed.
-Types can be explicitly cast, which could result in a type error. However, there are no implicit ways type errors can go undetected.

-Type Equivalence
-Two types are equivalent if an operand of one type in an expression can be substituted for one of the other type, without coercion.
-Name type equivalence: 2 variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name.

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

The types of the variables `count` and `index` would not be equivalent;

`count` could not be assigned to `index` or vice versa.

-Structure type equivalence: 2 variables have equivalent types if their types have identical structures.

```
type Celsius = Float;
     Fahrenheit = Float;
```

(Equivalent, but don't want it to be)

-Derived type: new type that is based on some previously defined type with which it is not equivalent, although it may have identical structure.

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

(Both are floats, but parameters can specify which of the derived type it'll take)

-Subtype: a type equivalent with its parent type.

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

Variables of both types, `Derived_Small_Int` and `Subrange_Small_Int`, have the same range of legal values and both inherit the operations of `Integer`. However, variables of type `Derived_Small_Int` are not compatible with any `Integer` type. On the other hand, variables of type `Subrange_Small_Int` are compatible with variables and constants of `Integer` type and any subtype of `Integer`.

-Theory & Data Type
-2 branches of type theory: Practical VS Abstract
-the practical branch is concerned with data types in commercial programming languages
-the abstract branch primarily focuses on typed lambda calculus
-Type System
-Rules that govern the set of types used in a program

Chapter 9: Subprograms [Chapter 9 (southeastern.edu)]

-Intro
-Process Abstraction as Subprograms save resources in memory as they can be reused, making them extremely important in computer programming.

-Fundamentals of Subprograms (aka functions & methods)

- Each subprogram has a single entry point.
- The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
- Control always returns to the caller when the subprogram execution terminates.

-Defs:
-Subprogram definition: describes the interface to and the actions of the subprogram abstraction.
-Subprogram call: is an explicit request that the called subprogram be executed
-Subprogram is said to be Active: if, after having been called, it has begun execution but has not yet completed that execution
-Subprogram header: The first line of definition

- It specifies that the following syntactic unit is a subprogram definition of some particular kind
- If the subprogram is not anonymous, the header provides a name for the subprogram
- It may optionally specify a list of parameters.

-Parameter Profile: (sometimes called the signature) of a subprogram is the number, order, and types of its formal parameters (args)

-Protocol: A subprogram's parameter profile plus, if it is a function, its return type (C & C++)
-Declaration: The protocol, without the body
-Java and C# do not need declarations

-Python methods can be made at runtime (i guess kinda)

```
if . . .
   def fun(. . . ):
   . . .
else
   def fun(. . . ):
   . . .
```

-Depending on if, fun() can be defined differently

-Parameters

-2 ways that a non-local method program can gain access to the data that it is to process: (direct access, parameter passing)

- Through **direct access** to non-local variables
  - Declared elsewhere but **visible** in the subprogram
- Through **parameter passing** "more flexible"
  - Data passed through parameters are accessed through names that are **local** to the subprogram
  - A subprogram with parameter access to the data it is to process is a parameterized computation
  - It can perform its computation on whatever data it receives through its parameters

-Actual/formal parameter correspondence: (positional (ie Java), keyword (any order, name based))

- **Positional**: The first actual parameter is bound to the first formal parameter and so forth
- **Keyword**: The name of the formal parameter is to be bound with the actual parameter.
  - They can appear in any order in the actual parameter list. Python functions can be called using this technique, as in

```
sumer(length = my_length, list = my_array, sum = my_sum)
```

  - Where the definition of `sumer` has the formal parameters `length`, `list`, and `sum`
  - Advantage: parameter order is **irrelevant**
  - Disadvantage: user of the subprogram must know the **names** of formal parameters

-Python, Ruby, C++, and PHP, formal parameters can have default values (when you don't specify arg, it defaults to that)

-Procedures and Functions

-2 distinct categories of subprograms: Procedures and Functions

-Functions return values and procedures do not (side effects)

- **Procedures** can produce results in the calling program unit by two methods:
  - If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure **can** change them
  - If the subprogram has formal parameters that allow the transfer of data to the caller, those parameters **can** be changed
- **Functions:** Functions structurally resemble procedures but are semantically modeled on mathematical functions
  - If a function is a faithful model, it produces **no** side effects
  - It modifies neither its parameters **nor** any variables defined outside the function
  - The **returned** value is its only effect

-Design Issues of a Subprogram

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

An **overloaded subprogram** is one that has the same name as another subprogram in the same referencing environment. A **generic subprogram** is one whose computation can be done on data of different types in different calls. A **closure** is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

-Local Referencing Environments

-Local Variable

-Def: Variables defined inside subprograms

-Static VS Stack Dynamic (bound to storage when the program begins execution and are unbound when execution terminates)?

- Local variables can be **static**
  - Advantages
    - Static local variables can be accessed faster because there is no indirection
    - No run-time overhead for allocation and deallocation
    - Allow subprograms to be history sensitive
  - Disadvantages
    - Inability to support recursion
    - Their storage can't be shared with the local variables of other inactive subprograms

- Local variables can be **stack dynamic**
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages:
    - Allocation/deallocation time
    - Indirect addressing "only determined during execution"
    - Subprograms cannot be history sensitive "can't retain data values between calls"

-C++, Java, and C# have only stack-dynamic local variables.

-Parameter-Passing Methods

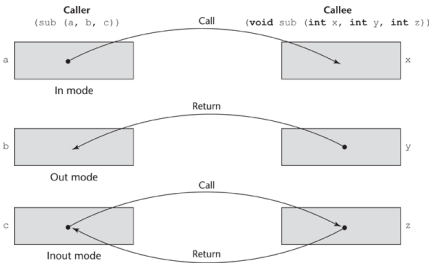
-The ways in which parameters are transmitted to and/or from called subprograms

-Semantic Models of Parameter Passing

-Formal Parameters

- **in** mode: They can receive data from corresponding actual parameters
- **out** mode: They can transmit data to the actual parameter
- **inout** mode: They can do both

For example, consider a subprogram that takes two arrays of `int` values as parameters— `list1` and `list2`. The subprogram must add `list1` to `list2` and return the result as a revised version of `list2`. Furthermore, the subprogram must create a new array from the two given arrays and return it. For this subprogram, `list1` should be in mode, because it is not to be changed by the subprogram. `list2` must be inout mode, because the subprogram needs the given value of the array and must return its new value. The third array should be out mode, because there is no initial value for this array and its computed value must be returned to the caller.



-2 Conceptual Models of how data transfers take places in parameter transmission

- Either an actual **value** is copied (to the caller, to the callee, or both ways), or
- An access **path** is transmitted

-Most commonly, the access path is a simple pointer or reference

-Implementation Models of Parameter Passing

-Pass-by-Value

- When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local var in the subprogram, thus implementing **in-mode** semantics
- Disadvantages:
  - Additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram
  - The actual parameter must be **copied** to the storage area for the corresponding formal parameter. The storage and the copy operations can be costly if the parameter is large, such as an **array** with many elements

-Pass-by-Result

- Pass-by-Result is an implementation model for **out-mode** parameters
- When a parameter is passed by result, **no value** is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the **caller's actual parameter**, which must be a variable
- One problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call

```
sub(p1, p1)
```

  - In `sub`, assuming the two formal parameters have different names, the two can obviously be assigned different values
  - Then whichever of the two is copied to their corresponding actual parameter **last** becomes the value of `p1`

-Pass-by-Value-Result (aka Pass-by-Copy)

- It is an implementation model for **inout-mode** parameters in which actual values are **copied**
- It is a combination of pass-by-value and pass-by-result
- The value of the actual parameter is used to **initialize** the corresponding formal parameter, which then acts as a local variable
- At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
- It is sometimes called **pass-by-copy** because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

-Pass-by-Reference

- Pass-by-reference is a second implementation model for **inout-mode** parameters
- Rather than copying data values back and forth. This method transmits an **access path**, usually just **an address**, to the called subprogram. This provides the access path to the cell storing the actual parameter
- The actual parameter is **shared** with the called subprogram
- Advantages
  - The passing process is **efficient** in terms of time and space. Duplicate space is not required, nor is any copying
- Disadvantages
  - Access to the formal parameters will be **slower** than pass-by-value, because of additional level of **indirect addressing** that is required
  - Inadvertent and erroneous **changes** may be made to the actual parameter
  - **Aliases** can be created as in C++

```
void fun(int &first, int &second)
```

    - If the call to fun happens to pass the same variable twice, as in

```
fun(total, total)
```

    - Then first and second in fun will be aliases

-Pass-by-Name

- The method is an **inout-mode** parameter transmission that does not correspond to a single implementation model
- When parameters are passed by name, the actual parameter is, in effect, **textually** substituted for the corresponding formal parameter in all its occurrences in the subprogram
- A formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.
- Because pass-by-name is **not** part of any widely used language, it is not discussed further here

-Implementing Parameter-Passing Methods

-In most contemporary languages, parameter communication takes place through the runtime stack

-The run-time stack is initialized and maintained by the run-time system, which is a system program that manages the execution of programs

-The run-time stack is used extensively for subprogram control linkage and parameter passing

- **Pass-by-value** parameters have their values copied into stack locations
  - The stack location then serves as storage for the corresponding formal parameters.
- **Pass-by-result** parameters are implemented as the opposite of pass-by-value copied from
  - The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram
- **Pass-by-value-result** parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-by-result
  - The stack location for the parameters is initialized by the call and it then used like a local variable in the called subprogram
- **Pass-by-reference** parameters are the simplest to implement.
  - Only its address must be placed in the stack
  - Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address

-Ex:

- The subprogram `sub` is called from `main` with the call **`sub(w, x ,y, z)`**, where `w` is **passed-by-value**, `x` is **passed-by-result**, `y` is **passed-by-value-result**, and `z` is **passed-by-reference**

Function call in `main`: `sub( w, x, y, z)`
Function header: **`void`** `sub(`**`int`** `a, `**`int`** `b, `**`int`** `c, `**`int`** `d)`
(pass `w` by **`value`**, `x` by **`result`**, `y` by **`value-result`**, `z` by **`reference`**)

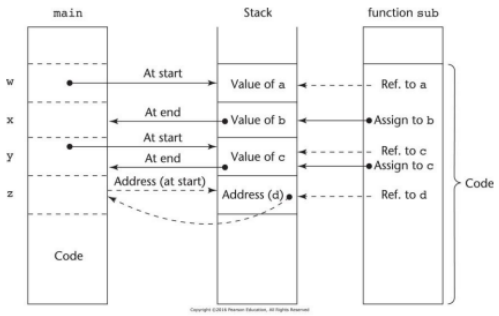


Figure 9.2 One possible stack implementation of the common parameter-passing methods

-Parameter-Passing Methods of Some Common Languages (blah blah, read for Java, C and whatnot)

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using **pointers** as parameters
- C++
  - A special pointer type called **reference** type. Reference parameters are **implicitly dereferenced** in the function or method, and their semantics is pass-by-reference
  - C++ also allows reference parameters to be defined to be **constants**. For example, we could have

```
void fun(const int &p1, int p2, int &p3)  { . . . }
```

    - `p1` is pass-by-reference: `p1` **cannot** be changed in the function `fun`
    - `p2` is pass-by-value
    - `p3` is pass-by-reference
    - Neither `p1` nor `p3` need be explicitly dereference in `fun`
- Java
  - All parameters are passed are **passed by value**
  - However, because objects can be accessed only through reference variables, object parameters are in effect **passed by reference**
    - Although an object reference passed as a parameter cannot itself be changed in the called subprogram, the referenced object can be changed if a method is available to cause the change
- Ada
  - Three semantics modes of parameter transmission: **in, out, inout**; in is the default mode
  - Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; inout parameters can be referenced and assigned
- C#
  - Default method: pass-by-value
  - Pass-by-reference can be specified by preceding both a formal parameter and its actual parameter with **ref**

```
void sumer(ref int oldSum, int newOne)  { . . . }
. . .
sumer(ref sum, newValue);
```

    - The first parameter to `sumer` is passed-by-reference; the second is passed-by-value
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects)
  - The process of changing the value of a variable with an assignment statement, as in

```
x = x + 1
```

  - does **not** change the object referenced by `x`. Rather, it takes the object referenced by `x`, increments it by `1`, thereby creating a **new** object (with the value `x + 1`), and then `x` to reference the new object.

-Type-Checking Parameters (widely accepted for reliability)

- It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.
- Ex:

```
result = sub1(1)
```

  - The actual parameter is an integer constant. If the formal parameter of `sub1` is a floating-point type, no error will be detected without parameter type checking.
- Early languages, such as Fortran 77 and the original version of C, did **not** require parameter type checking
- Pascal, **Java**, and Ada: it is always **required**
- Relatively new languages Perl, JavaScript, and PHP do **not** require type checking

-Design Issues for Functions

- Are side effects allowed?
- What types of values can be returned?
- How may values can be returned?

-Functional side effects

- Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be **in-mode** parameters
- For example, Ada functions can have only in-mode formal parameters
- This effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals
- In most languages, however, functions can have either **pass-by-value** or **pass-by-reference** parameters, thus allowing functions that cause side effects and aliasing

(Aka in-mode doesn't cause side effects. However, it can be limiting, so we have the other modes.)

-Types of returned values

- Most imperative programming languages restrict the types that can be returned by their functions
- C allows **any** type to be returned by its functions **except** arrays and functions. Both of these can be handled by pointer type return values
- C++ is like C but also allows user-defined types, or classes, to be returned from its functions
- Java and C# methods can return any type (but because methods are not types, methods cannot be returned)
- Python, Ruby, and Lua treat methods as first-class objects, so they can be returned, as well as any other class
- JavaScript functions can be passed as parameters and returned from functions

-Number of return values

- In most of languages, only a **single** value can be returned from a function
- Ruby allows the return of more than one value from a method
- Lua also allows functions to return **multiple** values
  - Such values follow the return statement as a comma-separated list, as in the following:

```
return 3, sum, index
```

  - If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

```
a, b, c = fun()
```

- In F#, multiple values can be returned by placing them in a tuple and having the tuple be the last expression in the function

-Overloading Subprograms

-Overloaded Operator

-One with multiple means

-The meaning of a particular instance is determined by its types of its operands

-Ex: Java *

For example, if the * operator
- It has two floating-point operands in a Java program, it specifies floating-point multiplication
- But if the same operator has two integer operands, it specifies integer multiplication

-Overloaded Subprogram

-Has the same name as another subprogram in the same referencing environment

-Must have a unique protocol

-The meaning of a call to an overloaded subprogram is determined by the actual parameter list

-Ex:

Ada, Java, C++, and C# include predefined overloaded subprograms
- For examples, overloaded constructors
- Users are also allowed to write multiple versions of subprograms with the same

-Having default parameters can lead to ambiguous subprogram calls

```
void fun(float b = 0.0);
void fun( );
. . .

fun( ); // The call is ambiguous and will cause a compilation error
```

-Generic Subprograms

-Generic or Polymorphic Subprogram: takes parameters of different types on different activations

-Ad Hoc Polymorphism: Overloaded Subprograms

-Subtype Polymorphism: A variable of type T can access any object of type T or any type derived from T (OOP languages)

-Parametric (aka Generic) Polymorphism: provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram

Generic Functions in C++

- Generic functions in C++ have the descriptive name of **template** functions
  - Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names

```
template <class Type>
Type max(Type first, Type second) {
  return first > second ? first : second;
}
```

    - where `Type` is the parameter that specifies the type of data on which the function will operate
  - For example, if it were instantiated with `int` as the parameter, it would be:

```
int max(int first, int second) {
  return first > second ? first : second;
}
```

  - The following is the C++ version of the generic sort subprogram

```
template <class Type>
void generic_sort (Type list [ ], int len) {
  int top, bottom;
  Type temp;
  for (top = 0, top < len - 2; top ++)
      for (bottom = top + 1; bottom < len - 1; bottom++)
  if (list [top] > list [bottom]) {
      temp = list [top];
      list[top] = list[bottom];
  } // end for bottom
} // end for generic
```

  - The instantiation of this template function is:

```
float flt_list [100];
. . .
generic_sort (flt_list, 100);
```

Generic Methods in Java 5.0

- Java's generic methods differ rom the generic subprogram of C++ in serval important ways:
  - Generic parameters in Java 5.0 must be **classes** – they cannot be primitive type
  - Java 5.0 generic methods are instantiated just once as truly generic methods
  - Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called bounds
- As an example of a generic Java 5.0 method:

```
public static <T> T doIt(T[] list) { . . . }
```

    - This defines a method named `doIt` that takes an array of elements of a generic type
    - The name the generic type is `T` and it must be an array
  - An example call to `doIt`:

```
doIt<String>(myList);
```

  - Generic parameters can have bounds:

```
public static <T extends Comparable> T doIt(T[] list) { . . . }
```

    - The generic type must be of a class that implements the `Comparable` interface

Generic Methods in C# 2005

- The generic method of C# 2005 are similar in capability to those of Java 5.0
- One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
- For example, consider the following skeletal class definition:

```
class MyClass {
  public static T DoIt<T>(T p1) { . . . }
  . . .
}
```

  - For example, both following class are legal:

```
int myInt = MyClass.DoIt(17);          // Calls DoIt<int>
string myStr = MyClass.DoIt('apples'); // Calls DoIt<string>
```

-User-Defined Overloaded Operators

-just Overload Operators 4head

- Operators can be overloaded in Ada, C++, Python, and Ruby (**not** carried over into Java)
- A Python example:

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

    - The method is named __add__
  - For example, the expression `x + y` is implemented as

```
x.__add__(y)
```

-Closures (for nested subprograms)

-Def: A subprogram and the referencing environment where it was defined

- The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
- A static-scoped language that does not permit **nested** subprograms does not need closures
- Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
- To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

-Ex:

- Following is an example of a closure written in JavaScript:

```
function makeAdder(x) {
   return function(y) {return x + y;}
}
. . .
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```

  - The closure is the anonymous function returned by `makeAdder`
  - The output of this code, assuming it was embedded in an HTML document and displed with a browser, is as follows:

```
Add 10 to 20: 30
Add 5 to 20: 25
```

  - In this example, the closuer is the **anonymous** function defined inside the `makeAdder` function, which `makceAdder` return
  - The variable `x` referenced in the closure function is bound to the parameter what was sent to `makeAdder`
  - The `makeAdder` function is called twice, once with a parameter of `10` and once with `5`
  - Each of these call returns a different version of the closure because they are bound to different values of `x`
  - The first call to `makeAdder` creates a function that adds `10` to it parameter; the second creates a function that adds `5` to its parameter

- In C#, this same closure function can be written in C# using nested anonymous delegate
  - The type of the nesting method is specified to be a function that takes an `int` as a parameter and returns an anonymous **delegate**
  - The return type is specified with the special nation for such delegates, `Func<int, int>`
    - The first type in the angle brackets is the parameter type
    - The second type is the return type of the method encapsulated by the delegate

```
static Func<int, int> makeAdder(int x) {
   return delegate(int y) {return x + y;};
}
. . .
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

    - The output of this code is exactly the same as for the previous JavaScript closure example

```
Add 10 to 20: 30
Add 5 to 20: 25
```

-Ex: In Class power function passing (week4)

-Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
  - Functions return values and procedures do **not**
- Local variables in subprograms can be stack-dynamic or static
- JavaScript, Python, Ruby, and Lua allow subprogram definitions to be **nested**
- Three models of parameter passing: **in**-mode, **out**-mode, and **inout**-mode
- Five implementation models of parameter passing:
  - **Pass-by-value**: in-mode
  - **Pass-by-result**: out-mode
  - **Pass-by-value-result**: inout-mode
  - **Pass-by-reference**: inout-mode
  - **Pass-by-name**: inout-mode
- C and C++ support **pointers to functions**. C# has **delegates**, which are object that can store references to methods
- Ada, C++, C#, Ruby, and Python allow both subprogram and **operator overloading**
- Subprograms in C++, Java 5.0, and C# 2005 can be **generic**, using parametric polymorphism, so the desired types of their data objects can be passed to the **compiler**, which then can construct unit for the required types
- A closure is a subprogram and its reference environment
  - Closures are useful in languages that allow **nested** subprograms, are static-scoped, and allow subprograms to be returned from functions and assigned to variables
- A coroutine is a special subprogram with **multiple** entries

Chapter 10: Implementing Subprograms [COMP 356 Programming Language Structures Notes (docplayer.net)]

-The General Semantics of Calls and Returns

-Subprogram Linkage: The subprogram call and return operations.

-Calling Process

-Include the implementation of whatever parameter-passing method is used.

-Allocate storage and bind nonstatic local variables

-Save everything needed to resume the calling program unit (registers, CPU status bits, Environment Pointers (EP))

-Deal with variable scope for nested subprograms

-Return Process

-Out and Inout, implemented by Copy: Move local variables to associate args

-Deallocate storage used for local vars

-Restore calling programing unit's control

-Implementing "Simple" Subprograms

-Simple meaning not nested, all static local variables

-Call Process

1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to the called.
4. Transfer control to the called.

-Caller have to do 1,2,3

-Return Process

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred back to the caller.

-Caller have to do 1,3,4

-Requires storing

- Status information about the caller
- Parameters
- Return address
- Return value for functions
- Temporaries used by the code of the subprograms

-In general, the linkage actions of the called can occur at two different times, either at the beginning of its execution or at the end.

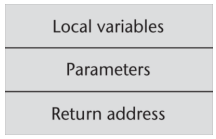
These are sometimes called the **prologue** and **epilogue** of the subprogram linkage.

-Consists of 2 parts

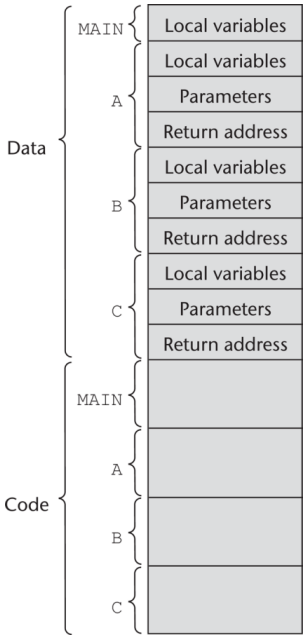
-The actual code: always constants, fixed size, containing static vars that can change during execution.

-The **Activation Record**: The noncode part, only relevant during activation/execution, static.

-Activation Record model



-Memory model of code and it's activation record



(languages can pair the record with the code instead)

-A **linker** puts this executable program together, part of the OS. (aka loaders, link editors)

In the previous example, the linker was called for `MAIN`. The linker had to find the machine code programs for `A`, `B`, and `C`, along with their activation record instances, and load them into memory with the code for `MAIN`. Then, it had to patch in the target addresses for all calls to `A`, `B`, `C`, and any library subprograms called in `A`, `B`, `C`, and `MAIN`.

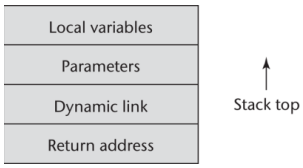
-Implementing Subprograms with Stack-Dynamic Local Variables

-With stack-dynamic local vars, you can do recursion

-More Complex Activation Records

- The compiler must generate code to cause the implicit allocation and deallocation of local variables.
- Recursion adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance (incomplete execution) of a subprogram at a given time, with at least one call from outside the subprogram and one or more recursive calls. The number of activations is limited only by the memory size of the machine. Each activation requires its own activation record instance.

-Activation Record



-**Dynamic Link**: A pointer to the base of the activation record instance of the caller.

-In dynamic-scoped languages, the dynamic link is used to access nonlocal variables

-In static-scoped languages, this link is used to provide traceback information when a run-time error occurs.

-Ex: C

```
void sub(float total, int part) {
  int list[5];
  float sum;
  . . .
}
```

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

-The **run-time stack** stores the calls of the subprograms bin FIFO type beat.
-The Environment Pointers
-EP points at the base of the first activation record of the program
-It should always point to the base of the current running subprogram unit
-When the subprogram is called, EP is saved in the new activation record instance as the dynamic link.
-EP is only saved, not stored in stack
-Upon return from the subprogram, the stack top is set to the value of the current EP minus one and the EP is set to the dynamic link from the activation record instance of the subprogram that has completed its execution.

-So that means the Linkage process is different
-Call

1. Create an activation record instance.
2. Save the execution status of the current program unit.
3. Compute and pass the parameters.
4. Pass the return address to the called.
5. Transfer control to the called.

-Prologue

1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

-Epilogue

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. Restore the stack pointer by setting it to the value of the current EP minus one and set the EP to the old dynamic link.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

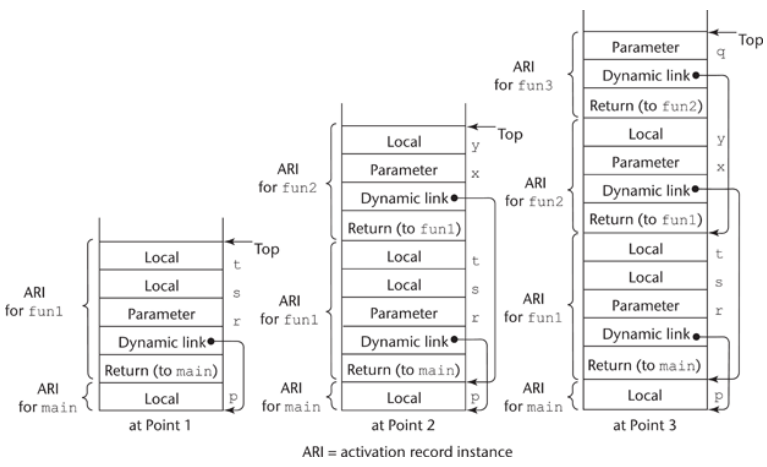
-Some languages pass parameters via registers

-Activation Record Examples
-No Recurssion C

```
void fun1(float r) {
  int s, t;
  . . .    <---------- 1
  fun2(s);
  . . .
}

void fun2(int x) {
  int y;
  . . .    <---------- 2
  fun3(y);
  . .
}
void fun3(int q) {
  . . .    <---------- 3
}
void main() {
  float p;
  . . .
  fun1(p);
  . . .
}
```

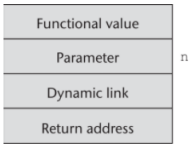
```
main calls fun1
fun1 calls fun2
fun2 calls fun3
```



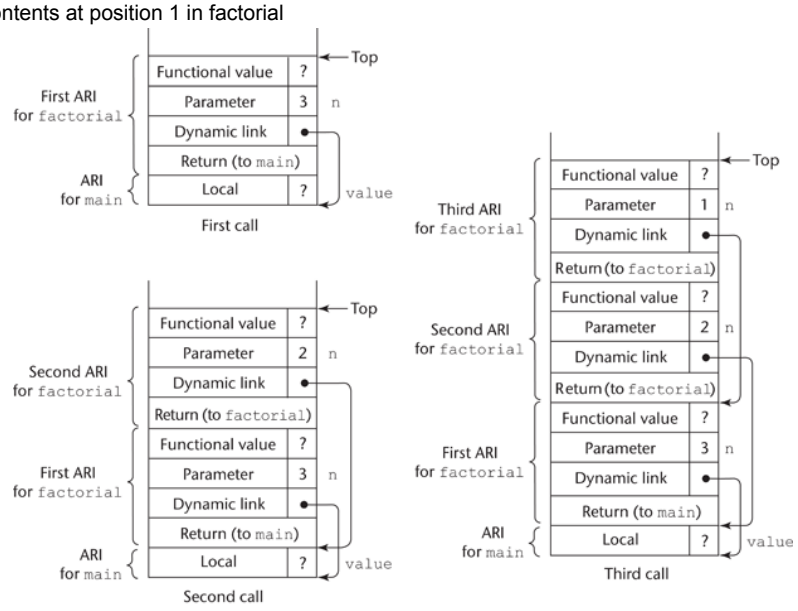
-Recurssion C

```
int factorial(int n) {
    <---------- 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <---------- 2
 }
void main() {
  int value;
  value = factorial(3);
    <---------- 3
 }
```

The activation record for `factorial`

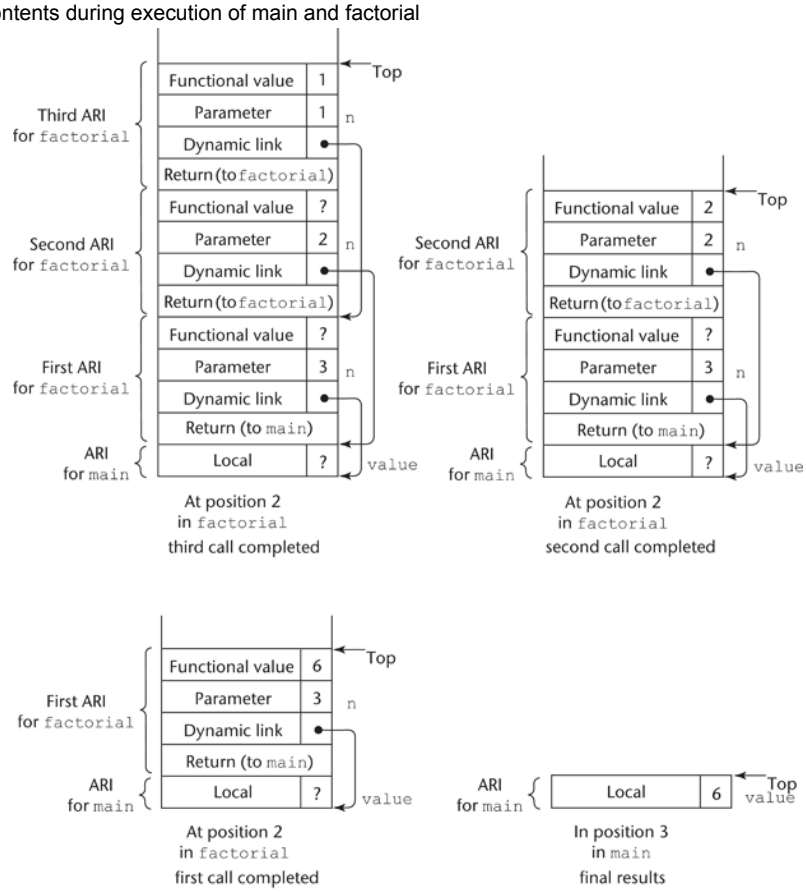


-Stack contents at position 1 in factorial



ARI = activation record instance

-Stack contents during execution of main and factorial



ARI = activation record instance

-Nested Subprograms

-Some of the non–C-based static-scoped programming languages use stack-dynamic local variables and allow subprograms to be nested.

-Fortran, Ada, Python, JavaScript, Ruby, and Swift

-The Basics

-References to a nonlocal variable in a static-scoped language with nested subprograms requires a two-step access process.

-**Static Chain**: a chain of static links that connect certain activation record instances in the stack.

-The most common way to implement static scoping in languages that allow nested subprograms is static chaining, which means that a new pointer,

called a static link, is added to the activation record which points to activation record of the caller

-Blocks

-Provide for user-specified local scopes for variables called **blocks**

```
{ int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

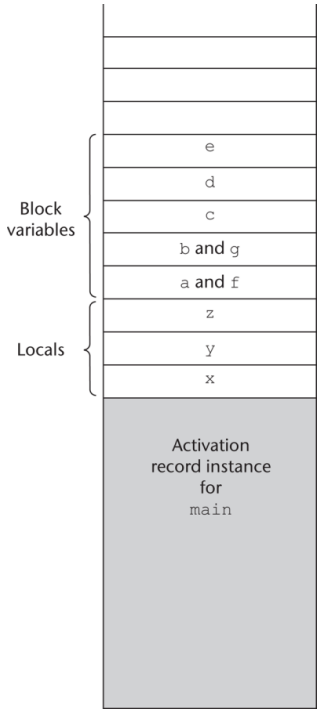
-Treated as parameterless subprograms

-So each has an activation record

-Easier and more efficient to implement, since size is static and you can compute the offset from one to another

-Ex:

```
void main() {
  int x, y, z;
  while ( . . ) {
    int a, b, c;
    .
    while ( . . . ) {
      int d, e;
      .
    }
  }
  while ( . . . ) {
    int f, g;
    . . .
  }
  . .
}
```



-Summary

Subprogram linkage semantics requires many actions by the implementation. In the case of "simple" subprograms, these actions are relatively uncomplicated. At the call, the status of execution must be saved, parameters and the return address must be passed to the called subprogram, and control must be transferred. At the return, the values of pass-by-result and pass-by-value-result parameters must be transferred back, as well as the return value if it is a function, execution status must be restored, and control transferred back to the caller. In languages with stack-dynamic local variables and nested subprograms, subprogram linkage is more complex. There may be more than one activation record instance, those instances must be stored on the run-time stack, and static and dynamic links must be maintained in the activation record instances. The static link is to allow references to nonlocal variables in static-scoped languages.

Subprograms in languages with stack-dynamic local variables and nested subprograms have two components: the actual code, which is static, and the activation record, which is stack dynamic. Activation record instances contain the formal parameters and local variables, among other things. Access to nonlocal variables is implemented with a chain of static parent pointers.

Access to nonlocal variables in a dynamic-scoped language can be implemented by use of the dynamic chain or through some central variable table method. Dynamic chains provide slow accesses but fast calls and returns. The central table methods provide fast accesses but slow calls and returns.

Chapter 11: Abstract Data Types and Encapsulation Constructs [chapter11.pptx (live.com)]

-Concept of Abstraction

An *abstraction* is a view or representation of an entity that includes only the most significant attributes

- The concept of abstraction is fundamental in programming (and computer science)
- Nearly all programming languages support process abstraction with subprograms
- Nearly all programming languages designed since 1980 support *data abstraction*

-Enclosure for data representation and subprograms

-Intro to Data Abstraction

-COBOL and C with records were the beginnings.

-An instance of an abstract data type is called an **object**.

-User Defined Abstract Data Types

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition
  - The declarations of the type and the protocols of the operations on objects of the type are contained in a single syntactic unit. Other program units are allowed to create variables of the defined type.

- Advantages the first condition
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting client code
  - Reduces the range of code and variables of which the programmer must be aware
  - Name conflicts are less likely
- Advantages of the second condition
  - Provides a method of program organization
  - Aids modifiability (everything associated with a data structure is together)
  - Separate compilation

-The program unit that uses a ADT is a **client** of that ADT.

-Language Examples

Uses a destructor to deallocate heap-dynamic members of the object	C++
Only allows implicit garbage collection to deallocate objects and references	Java
Uses structs as lightweight classes allocated on the stack and without inheritance	C#
Methods can be added or removed by parts of the program other than the original class definition	Ruby

(most likely right)

-Design Issues for ADTs

-Requirements

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

-4 key issues

- What is the form of the container for the interface to the type?
- Can abstract types be parameterized?
- What access controls are provided?
- Is the specification of the type physically separate from its implementation?

-Parameterized?

- Parameterized ADTs allow designing an ADT that can store any type elements – only an issue for static typed languages
- Also known as generic classes
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs

-Encapsulation Constructs

-Not data types anymore, but program abstraction.

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
- Such collections are called *encapsulation*

-ie libraries

-C

-#include, copying the headers into current program. You can copy it manually too., can create syncing issues if done like this.

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a *header file*
- Problem: the linker does not check types between a header and associated implementation
- `#include` preprocessor specification – used to include header files in applications

-C++

- Can define header and code files, similar to those of C
- Or, classes can be used for encapsulation
  - The class is used as the interface (prototypes)
  - The member definitions are defined in a separate file
- *Friend*s provide a way to grant access to private members of a class

-C# Assemblies

- A collection of files that appears to application programs to be a single dynamic link library or executable
- Each file contains a module that can be separately compiled
- A DLL is a collection of classes and methods that are individually linked to an executing program
- C# has an access modifier called `internal`; an `internal` member of a class is visible to all classes in the assembly in which it appears

-Naming Encapsulation

-Like how do you know assertEquals() is from JUnit instead of some user made Class?

- Large programs define many global names; need a way to divide into logical groupings
- A *naming encapsulation* is used to create a new scope for names
- C++ Namespaces
  - Can place each library in its own namespace and qualify names used outside with the namespace
  - C# also includes namespaces
- Java Packages
  - Packages can contain more than one class definition; classes in a package are *partial* friends
  - Clients of a package can use fully qualified name or use the ***`import`*** declaration
- Ada Packages
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the `with` clause

-Ruby Modules

-Modules define collections of methods and constants

-Summary

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs
- C++, C#, Java, Ada, and Ruby provide naming encapsulations

Chapter 12: Support for OOP [chapter12.pptx (live.com)]

-Intro

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., Ada 95+ and C++)
  - Some support functional program (e.g., CLOS)
  - Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
  - Some are pure OOP language (e.g., Smalltalk & Ruby)
  - Some functional languages support OOP, but they are not discussed in this chapter

-OOP

-3 major features of OOP

- Abstract data types (Chapter 11)
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- Polymorphism (Polymorphism: dynamic binding of method calls to methods)

-Inheriotance

- Productivity increases can come from reuse
  - ADTs are difficult to reuse—always need changes
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy (reuse via hierarchy cheese)

-Concepts/Vocab

- ADTs are usually called *classes*
- Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*
- Inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses
  - A class can hide entities from its clients
  - A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one *overrides* the inherited one
  - The method in the parent is *overriden*
- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent

-Subclass

-3 ways for subclass to be unique

- Three ways a class can differ from its parent:
  1. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass
  2. The subclass can add variables and/or methods to those inherited from the parent
  3. The subclass can modify the behavior of one or more of its inherited methods.

-Class VS Instance

- There are two kinds of variables in a class:
  - *Class variables* - one/class
  - *Instance variables* - one/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- Single vs. Multiple Inheritance
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance (Multiple Inheritance is just having at least a "grandparent")

-Dynamic Bindings/Dispatch

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
- Allows software systems to be more easily extended during both development and maintenance

-ie Java having super class variable storing subclass object. (Principle of Substitution)

-Abstract methods

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated

-Design Issues for OOP

- The Exclusivity of Objects
- Are Subclasses Subtypes?
- Single and Multiple Inheritance
- Object Allocation and Deallocation
- Dynamic and Static Binding
- Nested Classes
- Initialization of Objects

-Issue: The Exclusivity of Objects (aka Only Having Objects? Then what about Primitives?)

- Everything is an object
  - Advantage - elegance and purity
  - Disadvantage - slow operations on simple objects
- Add objects to a complete typing system
  - Advantage - fast operations on simple objects
  - Disadvantage - results in a confusing type system (two kinds of entities)
- Include an imperative-style typing system for primitives but make everything else objects
  - Advantage - fast operations on simple objects and a relatively small typing system
  - Disadvantage - still some confusion because of the two type systems

(confusing type system: There is no distinction between predefined and user-defined classes)

-Issue: Are Subclasses Subtypes?

-Class B is Subtype if B is derived from A; B has everything A has; the behavior of objects of type B when used in place of an object of A is identical

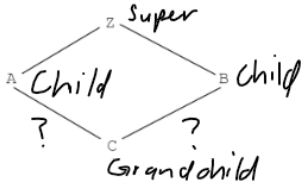
- Does an "is-a" relationship hold between a parent class object and an object of the subclass?
  - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
- A derived class is a subtype if it has an is-a relationship with its parent class
  - Subclass can only add variables and methods and override inherited methods in "compatible" ways

-A subtype inherits interfaces and behavior, while a subclass inherits implementation, primarily to promote code reuse.

-Issue: Single & Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is quite convenient and valuable

-Diamond Inheritance



(Should C inherit from A and/or B for a behavior?)

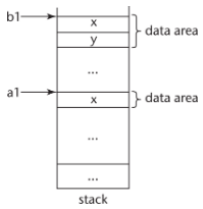
-Issue: Allocation & DeAllocation of Objects

- From where are objects allocated?
  - If they behave line the ADTs, they can be allocated from anywhere
    - Allocated from the run-time stack
    - Explicitly create on the heap (via `new`)
  - If they are all heap-dynamic, references can be uniform thru a pointer or reference variable
    - Simplifies assignment - dereferencing can be implicit
  - If objects are stack dynamic, there is a problem with regard to subtypes – *object slicing*
- Is deallocation explicit or implicit?

-Object Slicing

-You have B, a subtype of A. B has a unique instance variable.

-If you assign a B object to a polymorphic A variable, it can't hold both data.



-Issue: Nested Class

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

-Issue: Object Initialization

- Are objects initialized to values when they are created?
  - Implicit or explicit initialization
- How are parent class members initialized when a subclass object is created?

-Implementing OO Constructs

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

-Instance Data Storage (Class Instance Record (CIR))

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient

-Dynamic Binding of Method Calls

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

-Reflection

Chapter 15: Functional Programming Languages [pl11ch15.pdf (uidaho.edu)]

-Intro

-Imperative for efficient & for specific architecture, while functional uses math functions to broaden compatible architecture

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

-Java is imperative, Erlang is functional.

-Imperative has states via variables, while functional has none.

-Imperative depends on global variables in subprograms, allowing side effects.

A mathematical function maps its parameter(s) to a value (or values), rather than specifying a sequence of operations on values in memory to produce a value.

-Math Functions

-math be like function

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*

-Lambda Expressions

- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  λ(x) x * x * x

  for the function `cube(x) = x * x * x`

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

-Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

-Functional form

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

form

$h \equiv f \circ g$

$f(x) \equiv x + 2$

$g(x) \equiv 3 * x$

$h(x) \equiv f(g(x))$, or $h(x) \equiv (3 * x) + 2$

-Apply to All form

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For `h(x) ≡ x * x`

  α(h, (2, 3, 4)) yields `(4, 9, 16)`

-Fundamental of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics
- *Referential Transparency* – In an FPL, the evaluation of a function always produces the same result given the same parameters

-Functional Lang has Primitive Functions

A functional language provides a set of primitive functions, a set of functional forms to construct complex functions from those primitive functions, a function application operation, and some structure or structures for representing data. These structures are used to represent the parameters and values computed by functions. If a functional language is well designed, it requires only a relatively small number of primitive functions.

-

-Lisp Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.

  e.g., If the list `(A B C)` is interpreted as data it is a simple list of three atoms, `A`, `B`, and `C`

  If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`
- The first Lisp interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation

-Scheme

-Origin

- A mid-1970s dialect of Lisp, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of Lisp
- Uses only static scoping
- Functions are first-class entities
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables, passed as parameters, and returned from functions

-Scheme's Interpreter (REPL)

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
  - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

-Tail Recursion

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

-If it has tail recursion, it gets transformed into iterative by compiler

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
  (IF (<= n 0)
     1
     (* n (factorial (- n 1)))
))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 0)
     factpartial
     facthelper((- n 1) (* n factpartial)))
))
(DEFINE (factorial n)
  (facthelper n 1))
```

-Primitive Value Interpreter

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

-w/ Lambda expression

- Primitive Arithmetic Functions: `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`
  e.g., `(+ 5 2)` yields `7`

- Lambda Expressions
  - Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x)`

  `x` is called a bound variable

- Lambda expressions can be applied to parameters

  e.g., `((LAMBDA (x) (* x x)) 7)`

- `LAMBDA` expressions can have any number of parameters

  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

-ML

- A static-scoped functional language with syntax that is closer to Pascal than to Lisp
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Does not have imperative-style variables
- Its identifiers are untyped names for values
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

-Currying

Currying

- ML functions actually take just one parameter—if more are given, it considers the parameters a tuple (commas required)
- Process of *currying* replaces a function with more than one parameter with a function with one parameter that returns a function that takes the other parameters of the original function
- An ML function that takes more than one parameter can be defined in curried form by leaving out the commas in the parameters

```
fun add a b = a + b;
```

A function with one parameter, `a`. Returns a function that takes `b` as a parameter. Call: `add 3 5;`

-Partial Evaluation

Partial Evaluation

- Curried functions can be used to create new functions by partial evaluation
- Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost actual parameters

```
fun add5 x add 5 x;
```

Takes the actual parameter `5` and evaluates the `add` function with `5` as the value of its first formal parameter. Returns a function that adds `5` to its single parameter

```
val num = add5 10;  (* sets num to 15 *)
```

-Haskell

-Lazy Evaluation

- A language is *strict* if it requires all actual parameters to be fully evaluated
- A language is *nonstrict* if it does not have the strict requirement
- Nonstrict languages are more efficient and allow some interesting capabilities – *infinite lists*
- Lazy evaluation – Only compute those values that are necessary
- Positive numbers

```
positives = [0..]
```

- Determining if `16` is a square number

```
member [] b = False
member(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```

-Imperative VS Functional

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Less efficient execution
  - Programs can automatically be made concurrent

-Imperative also have to worry about side effects and state.

-Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution
- Lisp began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of Lisp that uses static scoping exclusively
- Common Lisp is a large Lisp-based language
- ML is a static-scoped and strongly typed functional language that uses type inference
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- F# is a .NET functional language that also supports imperative and object-oriented programming
- Some primarily imperative languages now incorporate some support for functional programming
- Purely functional languages have advantages over imperative alternatives, but still are not very widely used

Class: Lambda Expressions

Top-Level Division

- -Imperative - how a computer is to do something
  - -Procedural (Pascal, Basic, C)
  - -OO (Smalltake, C++, Java)
  - -Scripting (Per, Python, Javascript)
- -Declarative - hwat a computer is to do
  - -Function (Lisp, Scheme, Erlang, Haskell)
  - -Logic, Constraint Based (Prolog, Datalog, RPG)

-Ex Swap Comparison

- -C

```
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

- -Erlang

```
({a,b}) => {b,a}
```

-Functional Programming

- -1st class function & higher order functions
  - -Functions can be stored in variables including parameters.
  - -Functions can be passed into other functions.
  - -Functions can be returned from other functions.
- -Pure Functions
  - -No side effects (Not 100% true. Printing into console kinda need mutatable variables to get new input)
- -No State
  - -Variables are just constant
  - -Easier to do concurrency without threads overwriting each other's state
- -Recursion instead of iteration
- -Weakly typed
  - -Variables, Function Params, Returns, etc. do not need type definitions
- -Vocab review
  - -1st class function: Functions able to be stored in variables, returned and created by functions
  - -Side effect: When you have a global variable, and a mutating method changes that variable, changing the program's state.
- -In practiced
  - -if you got a lot of input at the "same" time, use functional lang to process it concurrenty doing REPL cheese.
  - -Elegant list processing

Chapter 5: Names, Bindings, and Scopes Chapter 5 (southeastern.edu) pl11ch5.pdf (uidaho.edu)

-5.1 Intro

- Imperative languages are abstractions of von Neumann architecture
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of memory
- Variables are characterized by a collection of properties or attributes
  - The most important of which is **type**, a fundamental concept in programming languages
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

(Type, Memory, Processor)

-5.2 Names

- -A name is a string of characters used to identify some entity in a program.
- -Design Issues
  - -Length
    - If too short, they cannot be connotative
    - Language examples:
      - FORTRAN I: maximum 6
      - COBOL: maximum 30
      - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
      - C# and Java: **no limit**, and all characters are significant
      - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
  - -Special Characters
    - PHP: all variable names must begin with dollar signs
    - Perl: all variable names begin with special characters, which specify the variable's type
    - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables
  - -Case Sensitivity
    - Disadvantage: readability (names that look alike are different)
      - Names in the C-based languages are case sensitive
      - Names in others are not
      - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
  - -Special Words
    - An aid to readability; used to delimit or separate statement clauses
    - A *keyword* is a word that is special only in certain contexts
    - A *reserved word* is a special word that cannot be used as a user-defined name
    - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

(reserved, keyword)

-5.3 Variables

-A variable is an abstraction of a memory cell or a collection of memory cells.

-Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Lifetime
- Scope

- Name
  - Not all variables have names: Anonymous, heap-dynamic variables
- Address
  - The memory address with which it is associated
  - A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time stack when the subprogram is called, different calls may result in that var having different addresses.
  - The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement.
- Aliases
  - If two variable names can be used to access **the same** memory location, they are called aliases
  - Aliases are created via **pointers**, **reference variables**, C and C++ **unions.**
  - Aliases are harmful to readability (program readers must remember all of them)
- Type
  - Determines the **range** of values of variables and the set of **operations** that are defined for values of that type; in the case of floating point, type also determines the precision.
  - For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.
- Value
  - The value of a variable is the **contents** of the memory cell or cells associated with the variable.
  - Abstract memory cell - the physical cell or collection of cells associated with a variable.
  - A variable's value is sometimes called its ***r*-value** because that is what is required when a variable appears in the **right** side of an assignment statement.
    - The ***l*-value** of a variable is its address.
    - The ***r*-value** of a variable is its value.

(type, l-value (address), r-value (value))

-5.4 The Concept of Binding

-A binding is an association, such as between an attribute and an entity (ie variable and its type), or between an operation and a symbol.

-Binding time is the time at which a binding takes place.

- Possible binding times:
  - Language design time: bind operator symbols to operations.
    - For example, the asterisk symbol (*) is bound to the multiplication operation.
  - Language implementation time:
    - A data type such as **int** in C is bound to a **range** of possible values.
  - Compile time: bind a variable to a **particular data type** at compile time.
  - Load time: bind a variable to a **memory cell** (ex. C **static** variables)
  - Runtime: bind a **nonstatic** local variable to a memory cell.
- Link Time:

(language design time, language implementation time, Compile time, load time, runtime)

-Static Vs Dynamic

- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

-Type Bindings

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an implicit declaration

-Static Type Binding

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.
- An **explicit** declaration is a program statement used for declaring the types of variables.
- An **implicit** declaration is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- Fortran, PL/I, Basic, and Perl provide implicit declarations.

(explicit, implicit)

-Ex: (also adavantages/disadvantages)

- In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
  I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
- Advantage: writability.
- Disadvantage: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
- In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
- Less trouble with **Perl**: Names that begin with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
- In this scenario, the names @apple and %apple are unrelated.

variable declaration that lists its type, e.g. in C: int myVar = 5;	explicit declaration
variable declaration that uses some default convention to determine the type, e.g. in Perl: $myVar = 5;	implicit declaration
variable declaration that uses context to determine its type, e.g. in Erlang: MyVar = 5.	type inference

-Type Inference

-Some languages use type inferencing to determine types of variables (var or no type is given, the compiler "guesses" the type)

- **C#** - a variable can be declared with **var** and an initial value. The initial value sets the type

```
var sum = 0;            // sum is int
var total = 0.0;        // total is float
var name = "Fred";      // name is string
```

- **Visual Basic, ML, Haskell, and F#** also use type inferencing. The context of the appearance of a variable determines its type

-Dynamic Type Binding

-With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name.
-Instead, the variable is bound to a type when it is assigned a value in an assignment statement.

- Dynamic Type Binding: In **Python, Ruby, JavaScript, and PHP**, type binding is dynamic
- Specified through an assignment statement
- Ex, JavaScript

```
list = [2, 4.33, 6, 8]; → single-dimensioned array
list = 47;              → scalar variable
```

  - Advantage: **flexibility** (generic program units)
  - Disadvantages:
    - **High cost** (dynamic type checking and interpretation)
      - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
      - Pure interpretation typically takes at least **10** times as long as to execute equivalent machine code.
    - Type error detection by the **compiler** is difficult because any variable can be assigned a value of any type.
      - Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
      - Ex, JavaScript

```
i, x        → Integer
y           → floating-point array

i = x;      → what the user meant to type

but because of a keying error, it has the assignment statement

i = y;      → what the user typed instead
```

      - **No error** is detected by the compiler or run-time system. `i` is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

-Storage Bindings and Lifetime

-Allocation: getting a cell from some pool of available cells
-Deallocation: putting a cell back into the pool.
-Lifetime:

- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes:
  - **static**,
  - **stack-dynamic**,
  - **explicit heap-dynamic**, and
  - **implicit heap-dynamic**

-Static Variables (Lifetime)

- Static variables are bound to memory cells **before** execution begins and remains bound to the same memory cell throughout execution
- e.g. all FORTRAN 77 variables, C **static variables** in functions
- Advantages:
  - **Efficiency** (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocation and deallocation of static variables.
  - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- Disadvantage:
  - Storage **cannot** be shared among variables.
  - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

-Stack-dynamic Variables (Lifetime)

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.
  - **Local variables** in C subprograms and Java methods.

- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.
  - Advantages:
    - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
    - In the absence of recursion, it conserves storage b/c all subprograms share the same memory space for their locals.
  - Disadvantages:
    - Overhead of allocation and deallocation.
    - Subprograms **cannot** be history sensitive.
    - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.

-Explicit Heap-dynamic Variables (Lifetime)

- Nameless memory cells that are allocated and deallocated by explicit directives "run-time instructions", specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. Dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;       // create a pointer
. . .
intnode = new int; // allocates the heap-dynamic variable
. . .
delete intnode;     // deallocates the heap-dynamic variable
                    // to which intnode points
```

  - An explicit heap-dynamic variable of int type is created by the new operator.
  - This operator can be referenced through the pointer, intnode.
  - The var is deallocated by the **delete** operator.
- In Java, all data except the primitive scalars are **objects**.
  - Java objects are explicitly heap-dynamic and are accessed through **reference variables**.
  - Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
  - Advantage:
    - Provides for dynamic storage management.
  - Disadvantage:
    - Inefficient "Cost of allocation and deallocation" and unreliable "difficulty of using pointer and reference variables correctly"

-Implicit Heap-dynamic Variables (Lifetime)

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment** statements.
- All their attributes are bound every time they are assigned.
- e.g. all variables in APL; all strings and arrays in Perl and JavaScript, and PHP.
- Ex, JavaScript

```
highs = [74, 84, 86, 90, 71]; ➔ an array of 5 numeric values
```

  - Advantage:
    - Flexibility allowing generic code to be written.
  - Disadvantages:
    - Inefficient, because all attributes are dynamic "run-time."
    - Loss of error detection by the compiler.

-5.5 Scope

- The scope of a variable is the range of statements in which the variable is visible.
- A variable is **visible** in a statement if it can be referenced in that statement.
- **Local variable** is local in a program unit or block if it is declared there.
- **Non-local variable** of a program unit or block are those that are visible within the program unit or block but are not declared there.

(local, non-local)

-Static Scope

- ALGOL 60 introduced the method of binding names to non-local vars is called **static scoping**.
- Static scoping is named because the scope of a variable can be statically determined – that is **prior** to execution.
- This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
- There are two categories of static scoped languages:
  - Nested Subprograms.
  - Subprograms that cannot be nested.
- Ada, and JavaScript, Common Lisp, Scheme, F#, and Python allow **nested** subprograms, but the C-based languages do **not**.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.

- For example:
  - Suppose a reference is made to a var `x` in subprogram `sub1`. The correct declaration is found by first searching the declarations of subprogram `sub1`.
  - If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is called its **static parent**.
    - If a declaration of `x` is not found there, the search continues to the next larger enclosing unit (the unit that declared `sub1`'s parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. ➔ an undeclared var error has been detected.
  - The static parent of subprogram `sub1`, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of `sub1`.
- Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`.
  - This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there.
  - The search thus continues in the static parent of `sub2`, `big`, where the declaration of `x` is found.
  - The `x` declared in `sub1` is ignored, because it is **not** in the static ancestry of `sub2`.
- The variable `x` is declared in both `big` and `sub1`, which is nested inside `big`.
  - Within `sub1`, every simple reference to `x` is to the local `x`.
  - The outer `x` is **hidden** from `sub1`

- Ex: JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function big() {
   function sub1() {
      var x = 7;
      sub2();
   }
   function sub2() {
      var y = x;
   }
   var x = 3;
   sub1();
}
```

-Evaluation

- Works well in many situations
- Problems:
  - In most cases, it allows more access to both variables and subprograms that is necessary
  - As a program evolves, the initial structure is destroyed and local variables often become **global**; subprograms also gravitate toward become global, rather than nested
- An alternative to the use of static scoping to control access to variables and subprograms is an **encapsulation** construct.

-Blocks

- From ALGOL 60, allows a section of code to have its own local variables whose scope is minimized.
- Such variables are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- The **C-based** languages allow any compound statement (a statement sequence surrounded by matched braces) to have declarations and thereby defined a new scope.
- Ex: Skeletal C function:

```
void sub() {
  int count;
  . . .
  while (. . .) {
      int count;
      count ++;
      . . .
  }
  . . .
}
```

- The reference to `count` in the while loop is to that loop's local `count`. The `count` of `sub` is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a variable with the same name in a larger enclosing scope.
- Note that this code is **legal** in C and C++ but **illegal** in Java and C#

- Most functional languages (Scheme, ML, and F#) include some form of **let** construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part

(Let construct)

- Ex. Scheme:

```
(LET (
  (name1 expression1)
  . . .
  (namen expressionn)
)
```

  - Consider the following call to LET:

```
(LET (
  (top (+ a b))
  (bottom (- c d))
  (/ top bottom)
)
```

  - This call computes and returns the value of the expression (a + b) / (c – d)

-Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear **anywhere** a statement can appear
- In C99, C++, and Java, the scope of all local variables is **from** the declaration to the end of the block
- In C#, the scope of any variable declared in a block is the **whole** block, regardless of the position of the declaration in the block
- However, a variable still must be declared before it can be used

- For example, consider the following C# code:

```
{int x;
  . . .
  {int x;   // Illegal
      . . .
  }
  . . .
}
```

  - Because the scope of a declaration is the whole block, the following nested declaration of x is also illegal:

```
{
  . . .
  {int x;   // Illegal
      . . .
  }
 int x;
}
```

  - Note that C# stall requires that all be declared before they are used
- In C++, Java, and C#, variables can be declared in for statements
  - The scope of such variables is restricted to the for construct

```
void fun() {
  . . .
  for (int count = 0; count < 10; count++) {
      int count;
      . . .
  }
  . . .
}
```

  - The scope of count is from the for statement to the end of for its body (the right brace)

-Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- For example, C and C++ have both declarations and definitions of global data
  - A declaration outside a function definition specifies that it is defined in another file
  - A global variable in C is implicitly visible in all subsequent functions in the file.
  - A global variable that is defined after a function can be made visible in the function by declaring it to be external, as the in the following:

```
extern int sum;
```

-Dynamic Scoping

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic. **Perl** and Common Lisp also allow variables to be declared to have dynamic scope, although the default scoping mechanism is these languages is static.
- Dynamic Scoping is based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined only at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

(calling sequence)

-5.6 Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- For example, In a Java method
  - The scope of such a variable is from its **declaration** to the end of the method
  - The lifetime of that variable is the period of **time** beginning when the method is entered and ending when execution of the method terminates
- Consider a **static** variable in a C or C++ function
  - Statically bound to the scope of that function and is also statically bound to storage
  - Its scope is static and local to the function but its lifetime extends over the **entire** execution of the program of which it is a part
- Ex: C++ functions

```
void printheader() {
   . . .
}     /* end of printheader */
void compute() {
   int sum;
   . . .
   printheader();
}     /* end of compute */
```

  - The **scope** of `sum` in contained within `compute` function
  - The **lifetime** of `sum` extends over the time during which `printheader` executes.
  - Whatever storage location `sum` is bound to before the call to `printheader`, that binding will continue during and after the execution of `printheader`.

-5.7 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, it is the local variables plus all of the visible variables in all of the enclosing scopes
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.
- 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.
- Ex, Python skeletal, **static-scoped language**

```
g = 3;                  # A global

def sub1():
   a = 5;               # Crates a local
   b = 6;               # Crates another local
   . . .                ← 1
   def sub2():
      global g;         # Global g is now assignable here
      c = 9;            # Creates a new local
      . . .             ← 2
      def sub3():
         nonlocal c;    # Makes nonlocal c visible here
         g = 11;        # Creates a new local
         . . .          ← 3
```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	local `a` and `b` (of `sub1`), global `g` for reference, but not for assignment
2	local `c` (of `sub2`), global `g` for both reference and for assignment **Note**: a and b (of sub1) for reference, but not for assignment
3	nonlocal `c` (of `sub2`), local `g` (of `sub3`) **Note**: a and b (of sub1) for reference, but not for assignment

- Ex, **Dynamic-scoped language**
- Consider the following program; assume that the only function calls are the following: `main` calls `sub2`, which calls `sub1`

```
void sub1( ) {
   int a, b;
   . . .                ← 1
}     /* end of sub1 */
void sub2( ) {
   int b, c;
   . . .                ← 2
   sub1();
}     /* end of sub2 */
void main( ) {
   int c, d;
   . . .                ← 3
   sub2( );
}     /* end of main */
```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main` (`c` of `main`, `b` of `sub2` hidden)
2	`b` and `c` of `sub2`, `d` of `main` (`c` of `main` is hidden)
3	`c` and `d` of `main`

-5.8 Named Constants

- It is a variable that is bound to a value only at the time it is bound to storage; its value **cannot** be change by assignment or by an input statement.
- Ex, Java

```
final int LEN = 100;
```

- Advantages: readability and modifiability

-Varable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, C++

```
int   sum = 0;
int*  ptrSum = &sum;
char  name[] = "George Washington Carver";
```

-Summary

- Variables are characterized by the 6 of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

- Binding is the association of attributes with program entities. Binding can be static or dynamic type binding.
  - **Static type binding**:
    - A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
    - Declaration either explicit or implicit, provide a means of specifying the static binding of variables to types
  - **Dynamic type binding**:
    - A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.
    - It allows greater flexibility but at the expense of readability, efficiency, and reliability
    -
- Scalar variables can be separated into 4 categories:
  - **Static Variables**
  - **Stack Dynamic Variables**
  - **Explicit Heap Dynamic Variables**
  - **Implicit Heap Dynamic Variables**

- The scope of a variable is the range of statements in which the variable is visible.
  - **Static scope**:
    - Static scoping is named because the scope of a variable can be **statically** determined – that is **prior** to execution
    - This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
    - It provides a simple, reliable, and efficient method of allowing visibility of nonlocal variables in subprograms
  - **Dynamic scope**:
    - It is based on **calling sequences** of program units, not their textual layout and thus the scope is determined only at **run time**.
    - It provides more flexibility than static scoping but, again, at expense of readability, reliability, and efficiency

Chapter 7: Expressions and Assignment Statements Chapter 7 (southeastern.edu) pl11ch7.pdf (uidaho.edu)

-7.1 Intro

- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence of imperative languages is dominant role of assignment statements.

-7.2 Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in mathematics.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
  - C-based languages include a **ternary** operator, which has three operands (conditional expression).
- The purpose of an arithmetic expression is to specify an arithmetic computation.

(unary, binary, ternary)

-Design Issues

- What are the operator **precedence** rules?
- What are the **operator associativity** rules?
- What is the **order of operand evaluation**?
- Are there restrictions on operand evaluation **side effects**?
- Does the language allow user-defined **operator overloading**?
- What **mode mixing** is allowed in expressions?

-Operand Evaluation Order

1. Variables: fetch the value from memory
2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
3. Parenthesized expressions: evaluate all operands and operators first
4. The most interesting case is when an operand is a function call

-Potentials for Side Effects

- *Functional side effects:* when a function changes a two-way parameter or a non-local variable
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

-Functional Side Effects

- Two possible solutions to the problem
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions
     - No non-local references in functions
     - **Advantage:** it works!
     - **Disadvantage:** inflexibility of one-way parameters and lack of non-local references
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations
     - Java requires that operands appear to be evaluated in left-to-right order

-Referential Transparency

- A program has the property of *referential transparency* if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If `fun` has no side effects, `result1 = result2`

Otherwise, not, and referential transparency is violated

- Advantage of referential transparency
  - Semantics of a program is much easier to understand if it has referential transparency
- Because they do not have variables, programs in pure functional languages are referentially transparent
  - Functions cannot have state, which would be stored in local variables
  - If a function uses an outside value, it must be a constant (there are no variables). So, the value of a function depends only on its parameters

-Operator Evaluation Order

-Precedence

- The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evaluated.
- Many languages also include unary versions of addition and subtraction.
- Unary addition **(+)** is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
- In Java and C#, unary minus also causes the implicit conversion of `short` and `byte` operands to `int` type.
- In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is **parenthesized** to prevent it from being next to another operator. For example, unary minus operator (-):

```
A + (- B) * C          // is legal
A + - B * C            // is illegal
```

- Exponentiation has higher precedence that unary minus, so

```
-A ** B
```

Is equivalent to

```
-(A ** B)
```

- The precedences of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	*C-Base Languages*
Highest	`**`	`postfix ++, --`
	`unary +, -`	`prefix ++, --, unary +, -`
	`*, /, %`	`*, /, %`
Lowest	`binary +, -`	`binary +, -`

-Associativity

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the **same precedence** level are evaluated. An operator can be either left or right associative.
- Typical associativity rules:
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., Fortran)
- Ex: Java

```
a - b + c              // left to right
```

- Ex: Fortran

```
A ** B ** C            // right to left

(A ** B) ** C          // in Ada it must be parenthesized
```

- The associativity rules for a few common languages are given here:

Language	Associativity Rule
Ruby, Fortran	`Left: *, /, +, -`
	`Right: **`
C-based languages	`Left: *, /, %, binary +, binary -`
	`Right: ++, --, unary -, unary +`

- **APL** is different; **all** operators have equal precedence and all operators associate **right to left**.
- Ex: APL

```
A x B + C              // A = 3, B = 4, C = 5   ➔   27
```

- Precedence and associativity rules can be overridden with **parentheses**.

-Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
- A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.
- Ex:

```
(A + B) * C            // addition will be evaluated first
```

-Expressions in Lisp

- All arithmetic and logic operations are by explicitly called subprograms
- Ex: to specify the c expression `a + b * c` in Lisp, one must write the following expression:

```
(+ a (* b c))          // + and * are the names of functions
```

-Conditional Expressions

- Sometimes **if-then-else** statements are used to perform a conditional expression assignment.
- Ex: C-based languages

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

- In the C-based languages, this can be specified more conveniently in an assignment statement using a conditional expression. Note that ? is used in conditional expression as a ternary operator (3 operands).

```
expression_1 ? expression_2 : expression_3
```

- Ex:

```
average = (count == 0) ? 0 : sum / count;
```

-7.3 Overloaded Operators

- The use of an operator for **more than one purpose** is operator overloading.
- Some are common (e.g., + for `int` and `float`).
- Java uses + for addition and for **string catenation**.
- Some are potential trouble (e.g., `&` in C and C++)

```
x = &y     // & as unary operator is the address of y
           // &  as binary operator bitwise logical AND
```

  - Causes the address of `y` to be placed in `x`.
  - Some loss of readability to use the same symbol for two completely unrelated operations.
  - The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler "difficult to diagnose".

- C++, C#, and F# allow **user-defined** overloaded operators
  - When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense

-7.4 Type Conversions

-Narrowing & Widening

- A **narrowing** conversion is one that converts an object to a type that cannot include all of the values of the original type e.g., **`double`** to **`float`**.
- A **widening** conversion is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., **`int`** to **`float`**.

-Coercion in Expressions

- A **mixed-mode expression** is one that has operands of different types.
- A coercion is an **implicit** type conversion.
- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In ML and F#, there are **no** coercions in expressions
- Language designers are not in agreement on the issue of coercions in arithmetic expressions
  - Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they eliminate the benefits of type checking
  - Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions.
  - The issue is whether programmers should be concerned with this category of errors or whether the compiler should detect them.
- Ex: Java

```
int a;
float b, c, d;
. . .
d = b * a;
```

  - Assume that the second operand of the multiplication operator was supposed to be `c`, but because of a keying error it wat typed as `a`
  - Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, `a` will be coerced to **`float`**.

-Explicit Type Conversions

- In the C-based languages, explicit type conversions are called **casts**
- Ex: In Java, to specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

```
(int)angle
```

-Errors in Expressions

- Caused by:
  - Inherent limitations of arithmetic — e.g. division by zero
  - Limitations of computer arithmetic — e.g. overflow or underflow
- Floating-point overflow and underflow, and division by zero are examples of **run-time errors**, which are sometimes called exceptions.

-7.5 Relational and Boolean Expressions

-Relational Expressions

- A relational operator: an operator that compares the values of its two operands
- Relational Expressions: two operands and one relational operator
- The value of a relational expression is Boolean, unless it is not a type included in the language
  - Use relational operators and operands of various types
  - Operator symbols used vary somewhat among languages (`!=, /=, .NE., <>, #`)
- The syntax of the relational operators available in some common languages is as follows:

Operation	***Ada***	***C-Based Languages***	***Fortran 95***
Equal	`=`	`==`	`.EQ. or ==`
Not Equal	`/=`	`!=`	`.NE. or <>`
Greater than	`>`	`>`	`.GT. or >`
Less than	`<`	`<`	`.LT. or <`
Greater than or equal	`>=`	`>=`	`.GE. or >=`
Less than or equal	`<=`	`<=`	`.LE. or >=`

- JavaScript and PHP have two additional relational operator, `===` and `!==`
  - Similar to their cousins, `==` and `!=`, except that they do not coerce their operands

```
"7" == 7     // true in JavaScript
"7" === 7    // false in JavaScript, because no coercion is done on the operand
                of this operator
```

-Boolean Expressions

- Operands are Boolean and the result is **Boolean**

FORTRAN 77	*FORTRAN 90*	*C*	*Ada*
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not

- Versions of C prior to C99 have **no** Boolean type; it uses `int` type with 0 for false and **nonzero** for true.
- One odd characteristic of C's expressions: `a > b > c` is a legal expression, but the result is not what you might expect

```
a > b > c
```

  - The left most operator is evaluated first because the relational operators of C are **left** associative, producing either 0 **or** 1
  - Then, this result is compared with var `c`. There is **never** a comparison between `b` and `c`.

-7.6 Short-Circuit Evaluation

- A short-circuit evaluation of an expression is one in which the result is determined **without** evaluating all of the operands and/or operators.
- Ex:

```
(13 * a) * (b/13 - 1)        // is independent of the value of (b/13 - 1)
                             if a = 0, because 0 * x = 0 for any x
```

  - So when `a = 0`, there is no need to evaluate `(b/13 - 1)` or perform the second multiplication.
  - However, this shortcut is not easily detected during execution, so it is never taken.
- The value of the Boolean expression:

```
(a >= 0) && (b < 10)         // is independent of the second expression
                             if a < 0, because expression (FALSE && (b < 10))
                             is FALSE for all values of b
```

  - So when a < 0, there is **no** need to evaluate b, the constant 10, the second relational expression, or the && operation.
  - Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.
- Short-circuit evaluation exposes the potential problem of side effects in expressions

```
(a > b) || (b++ / 3)         // b is changed only when a <= b
```

- If the programmer assumed `b` would change every time this expression is evaluated during execution, the program will fail.
- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**&&** and ||), but also provide **bitwise** Boolean operators that are **not** short circuit (**&** and |)

-7.7 Assignment Statements

-Simple Assignments (ie =)

- The C-based languages use == as the equality relational operator to avoid confusion with their assignment operator
- The operator symbol for assignment:
  1. = Fortran, Basic, PL/I, C, C++, Java
  2. := ALGOL, Pascal, Ada

-Conditional Targets (ie tenary)

-Ex: Perl

```
($flag ? $count1 : $count2) = 0; ⇔    if ($flag) {
                                          $count1 = 0;
                                      } else {
                                          $count2 = 0;
                                      }
```

-Compound Assignment Operators (ie +=)

- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment
- The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

- The syntax of assignment operators that is the catenation of the desired binary operator to the = operator

```
sum += value;        ⇔    sum = sum + value;
```

-Unary Assignment Operators (ie ++)

- C-based languages include two special unary operators that are actually abbreviated assignments
- They combine increment and decrement operations with assignments
- The operators ++ and -- can be used either in expression or to form stand-alone single-operator assignment statements. They can appear as **prefix** operators:

```
sum = ++ count;      ⇔    count = count + 1; sum = count;
```

- If the same operator is used as a **postfix** operator:

```
sum = count ++;      ⇔    sum = count; count = count + 1;
```

-Assignment as an Expression (having assignments in an expression, very cancerous to read)

- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- Ex:

```
while ((ch = getchar())!= EOF) {. . .}
                                  // why ( ) around assignment?
```

  - The assignment statement must be parenthesized because the precedence of the assignment operator is **lower** than that of the relational operators.
- Disadvantage: another kind of expression side effect which leads to expressions that are **difficult** to read and understand. For example

```
a = b + (c = d / b) - 1
```

denotes the instructions

```
Assign d / b to c
Assign b + c to temp
Assign temp - 1 to a
```

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors.

```
if (x = y) . . .
```

instead of

```
if (x == y) . . .
```

-Multiple Assignments (ie assigning each value of a list in 1 line)

- Perl, Ruby, and Lua provide **multiple-target** multiple-source assignments
- Ex: Perl

```
($first, $second, $third) = (20, 30, 40);
```

  - The semantics is that `20` is assigned to `$first`, `40` is assigned to `$second`, and `60` is assigned to `$third`.

    Also, the following is legal and performs an **interchange**:

```
($first, $second) = ($second, $first);
```

  - The correctly interchanges the values of `$first` and `$second`, `60` **without** the use of a temporary variable

-Erlang ex: {A, B, C} = {1, 2, 3}.

-Assignment in Functional Programming Languages

- Identifiers in functional languages are only names of values
- Ex: in ML, names are bound to values with the **`val`** declaration, whose form is exemplified in the following:

```
val cost = quantity * price;
```

  - If `cost` appears on the left side of a subsequent **`val`** declaration, that declaration creates a **new** version of the name `cost`, which has **no** relationship with the previous version, which is then hidden
- F#'s **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

Chapter 8: Statement-Level Control Structures southeastern.edu http://aturing.umcs.maine.edu/~meadow/courses/cos301/cos301-8.pdf

-8.1 Intro

- A control structure is a control statement and the statements whose execution it controls
  - Selection Statements
  - Iterative Statements
- There is only one design issue that is relevant to all of the selection and iteration control statements:
  - Should a control structure have multiple entries?

-8.2 Selection Statements

- A selection statement provides the means of choosing between two or more paths of execution.
- Selection statement fall into two general categories:
  - Two-way selection
  - Multiple-way selection

-Two-way Selection Statements

- The general form of a two-way selector is as follows:

```
if control_expression
   then clause
   else clause
```

- Design issues
  - What is the form and type of the control expression?
  - How are the **`then`** and **`else`** clauses specified?
  - How should the meaning of nested selectors be specified?
- The control expression
  - Control expressions are specified in parenthesis if the `then` reserved word is not used to introduce the `then` clause, as in the C-based languages
  - In C89, which did not have a Boolean data type, **arithmetic** expressions were used as control expressions
  - In contemporary languages, such as Java and C#, **only Boolean** expressions can be used for control expressions
- Clause Form
  - In most contemporary languages, the `then` and `else` clauses either appear as single statements or compound statements.
  - C-based languages use **braces** to form compound statements.
  - One exception is **Perl**, in which all then and else clauses must be **compound** statements, even if they contain single statements
  - In Python and Ruby, clauses are statement sequences
  - Python uses **indentation** to define clauses

```
if x > y :
   x = y
   print " x was greater than y"
```

    - All statements equally indented are included in the compound statement. Notice that rather than `then`, a colon is used to introduce the `then` clause in the Python

- Nesting Selectors
  - In Java and contemporary languages, the static semantics of the language specify that the `else` clause is always paired with the **nearest** unpaired `then` clause

```
if (sum == 0)
    if (count == 0)
        result = 0;
else
   result = 1;
```

    - A rule, rather than a syntactic entity, is used to provide the disambiguation
    - So, in the example, the `else` clause would be the alternative to the second `then` clause
    - To force the alternative semantics in Java, a different syntactic form is required, in which the inner `if` is put in a compound, as in

```
if (sum == 0) {
    if (count == 0)
        result = 0;
}
else
   result = 1;
```

  - C, C++, and C# have the same problem as Java with selection statement nesting
  - Ruby, statement sequences as clauses:

```
if sum == 0 then
    if count == 0 then
        result = 0
    else
        result = 1
    end
end
```

  - Python, all statements uses indentation to define clauses

```
if sum == 0 :
    if count == 0 :
        result = 0
    else :
        result = 1
```

-Multiple-way Selection Constructs

- The multiple selection construct allows the selection of one of any number of statements or statement groups.

- Design Issues
  - What is the form and type of the control expression?
  - How are the selectable segments specified?
  - Is execution flow through the structure restricted to include just a single selectable segment?
  - How are case values specified?
  - What is done about unrepresented expression values?
- C, C++, Java, and JavaScript switch

```
switch (expression) {
   case constant_expression1 : statement1;
        ...
   case constant_expressionn : statementn;
   [default: statementn+1]
}
```

  - The control expression and the constant expressions some discrete type including integer types as well as character and enumeration types
  - The selectable statements can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  - `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)
  - Any number of segments can be executed in one execution of the construct (a trade-off between reliability and flexibility—convenience.)
  - To avoid it, the programmer must supply a break statement for each segment.
- C# switch
  - C# switch statement differs from C-based in that C# has static semantic rule disallows the implicit execution of more than one segment
  - The rule is that every selectable segment must end with an explicit unconditional branch statement either a `break`, which transfers control out of the switch construct, or a `goto`, which can transfer control to on of the selectable segments. C# switch statement example:

```
switch (value)   {
   case -1:    Negatives++;
               break;
   case  0:    Positives++;
               goto case 1;
   case  1:    Positives ++;
   default:    Console.WriteLine("Error in switch \n");
}
```

- Multiple Selection Using `if`
  - Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses
  - Ex, Python selector statement (note that else-if is spelled `elif` in Python):

```
if count < 10 :
   bag1 = True
elif count < 100 :
   bag2 = True
elif count < 1000 :
   bag3 = True
```

which is equivalent to the following:

```
if count < 10 :
   bag1 = True
else :
   if Count < 100 :
      bag2 = True
else :
     if Count < 1000 :
        bag3 = True
```

    - The `elsif` version is the more readable of the two.

  - The Python example can be written as a Ruby `case`

```
case
   when count < 10       then bag1 = true
   when count < 100      then bag2 = true
   when count < 1000     then bag3 = true
end
```

  - Notice that this example is not easily simulated with a **switch-case** statement, because each selectable statement is chosen on the basis of a Boolean expression
  - In fact, none of the multiple selectors in contemporary languages are as **general** as the if-then-else-if statement

-8.3 Iterative Statment

- An iterative statement is one that cause a statement or collection of statements to be executed zero, one, or more times
- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- An iterative statement is often called **loop**
- Iteration is the very essence of the power of computer
- The repeated execution of a statement is often accomplished in a functional language by recursion rather than by iteration
- General design issues for iteration control statements:
  - How is iteration controlled?
  - Where should the control mechanism appear in the loop statement?
- The primary possibilities for iteration control are logical, counting, or a combination of the two
- The main choices for the location of the control mechanism are the top of the loop or the bottom of the loop
- The **body** of a loop is the collection of statements whose execution is controlled by the iteration statement
- The term **pretest** means that the loop completion occurs before the loop body is executed
- The term **posttest** means that the loop completion occurs after the loop body is executed
- The iteration statement and the associated loop body together form an **iteration statement**

-Counter-Controlled Loops

- A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained
- It also includes means of specifying the **intial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, called the **stepsize**.
- The intial, terminal and stepsize are called the **loop parameters**.
- Design issues:
  - What are the type and scope of the loop variable?
  - Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  - Should the loop parameters be evaluated only once, or once for every iteration?
  -
- Fortran 90's **DO** syntax:

```
[name:]  DO label variable = initial, terminal [, stepsize]
            . . .
END DO [name]
```

  - The label is that of the last statement in the loop body, and the stepsize, when absent, defaults to **1**.
  - Loop variable **must** be an INTEGER and may be either negative or positive.
  - The loop parameters are allowed to be expressions and can have negative or positive values.
  - They are evaluated at the beginning of the execution of the DO statement, and the value is used to compute an iteration count, which then has the number of times the loop is to be executed.
  - The loop is controlled by the iteration count, not the loop param, so even if the params are changed in the loop, which is legal, those changes cannot affect loop control.
  - The iteration count is an internal var that is inaccessible to the user code.
  - The DO statement is a single-entry structure

- The for statement of the C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3])
   loop body
```

  - The loop body can be a single statement, a compound statement, or a null statement

```
for (count = 1; count <= 10; count++)
   . . .
```

  - All of the expressions of C's `for` are optional
  - If the second expression is absent, it is an **infinite** loop
  - If the first and third expressions are absent, no assumptions are made
  - The C-based languages for design choices are:
    - There are no explicit loop variable or loop parameters
    - All involved variables can be changed in the loop body
    - First expression is evaluated once, but the other two are evaluated with each iteration
    - It is legal to branch into the body of a for loop in C
  - C's for is more **flexible** than the counting loop statements of Fortran and Ada, because each of the expressions can comprise multiple statements, which in turn allow multiple loop variables that can be of any type
  - Consider the following for statement:

```
for (count1 = 0, count2 = 1.0;
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5)
      ;
```

The operational semantics description of this is:

```
count1 = 0
count2 = 1.0
loop:
   if count1 > 10 goto out
   if count2 > 100.0 goto out
   count1 = count1 + 1
   sum = count1 + count2
   count2 = count2 * 2.5
   goto loop
out:
   . . .
```

    - The loop above does not need and thus does **not** have a loop body
  - The for statement of C99 and C++ differs from earlier version of C in two ways:
    - It can use an arithmetic expression or a Boolean expression for loop control
    - The first expression can include variable definitions (scope is from the definition to the end of the loop body), for example

```
for (int count = 0; count <= 10; count++) { . . . }
```

  - The for statement of Java and C# is like that of C++, except that the loop control expression is restricted to Boolean

- The for statement of Python
  - The general form of Python's `for` is:

```
for loop_variable in object:
    - loop body
[else:
    - else clause]
```

  - The `object` is often range, which is either a list of values in brackets `([2, 4, 6])`, or a call to the range function `(range(5)`, which returns `0, 1, 2, 3, 4)`
  - The loop variable takes on the values specified in the given range, one for each iteration
  - The else clause, which is optional, is executed if the loop terminates normally
  - Consider the following example:

```
for count in [2, 4, 6] :
    print count
```

produces

```
2
4
6
```

-Logically Controlled Loops

- Repetition control is based on a Boolean expression rather than a counter
- Design Issues:
  - Should the control be pretest or posttest?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?
- The C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements
- The **pretest** and **posttest** logical loops have the following forms (`while` and `do-while`):

```
while (control_expression)
   loop body
```

and

```
do
   loop body
while (control_expression);
```

- These two statements forms are exemplified by the following C# code:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine( ));
while (indat >= 0) {
   sum += indat;
   indat = Int32.Parse(Console.ReadLine( ));
}

value = Int32.Parse(Console.ReadLine( ));
do {
   value /= 10;
   digits ++;
} while (value > 0);
```

- The only real difference between the do and the while is that the do always causes the loop body to be executed **at least once**
- Java's `while` and `do` statements are similar to those of C and C++, except the control expression must be Boolean type, and because Java does **not** have a goto, the loop bodies cannot be entered anywhere but at their beginning

-User-Located Loop Control Mechanisms

- It is sometimes convenient for a programmer to choose a location for loop control other than the top or bottom of the loop
- Design issues:
  - Should the conditional mechanism be an integral part of the exit?
  - Should only one control body be exited, or can enclosing loops also be exited?
- C and C++ have unconditional unlabeled exits (`break`)
- Java, Perl, and C# have unconditional labeled **exits** (`break` in Java and C#, `last` in Perl)
- The following is an example of nested loops in C#:

```
OuterLoop:
   for (row = 0; row < numRows; row++)
      for (col = 0; col < numCols; col++) {
         sum += mat[row][col];
         if (sum > 1000.0)
              break outerLoop;
   }
```

- C and C++ include an unlabeled control statement, `continue`, that transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to **skip** the rest of the loop statements on the current iteration without terminating the loop structure. Ex:

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) continue;
   sum += value;
}
```

- A negative value causes the assignment statement to be **skipped**, and control is transferred instead to the conditional at the top of the loop
- On the other hand, in

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) break;
   sum += value;
}
```

  - A negative value **terminates** the loop

- Java, Perl, and C# have statements similar to continue, except they can include labels that specify which loop is to be continued
- The motivation for user-located loop exits is simple: They fulfill a common need for goto statements through a highly restricted branch statement
- The target of a goto can be many places in the program, both above and below the goto itself
- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement

-Iteration Based on Data Structures

- A general data-based iteration statement uses a user-defined data structure and a user-defined function (the **iterator**) to go through the structure's elements
- The iterator is called at the beginning of each iteration, and each time it is called, the iterator return a n element from a particular data structure in some specific order
- C's `for` can be used to build a user-defined iterator:

```
for (ptr=root; ptr==NULL; ptr = traverse(ptr)) {
   . . .
}
```

- Java 5.0 uses `for`, although it is called `foreach`
  - The following statement would iterate though all of its elements, setting each to myElement:

```
for (String myElement : myList) { . . . }
```

  - The new statement is referred to as "`foreach`," although is reserved word is `for`

- C#'s `foreach` statement iterates on the elements of array and other collections
  - C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues)
  - For example, there are **generic** collection classes for lists, which are dynamic length array, stacks, queues, and dictionaries (has table)
  - All of these predefined generic collections have built-in iterator that are used implicitly with the `foreach` statement
  - Furthermore, users can define their own collections and write their own iterators, implement the `IEnumerator` interface, which enables the use of use `foreach` on these collections

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Ted");
foreach (Strings name in names)
      Console.WriteLine (name);
```

-8.4 Unconditional Branching

- An unconditional branch statement transfers execution control to a specified place in the program
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
- However, using the goto carelessly can lead to **serious problems**
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly **unreliable** and costly to **maintain**
- There problems follow directly from a goto's ability to force any program statement to follow any other in execution sequence, regardless of whether the statement proceeds or follows previously executed statement in textual order
- Java, Python, and Ruby do **not** have a goto. However, most currently popular languages include a goto statement
- **C#** uses goto in the **switch** statement

-8.5 Guarded Commands

- New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
- His primary motivation was to provide control statements that would support a **new** program design methodology that ensured correctness (verification) during development rather than when verifying or testing completed programs
- Basis for two linguistic mechanisms for concurrent programming in CSP (Hoare, 1978)
- Basic idea: if the order of evaluation is not important, the program should **not** specify one
- Dijkstra's selection guarded command has the form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[] <Boolean expr> -> <statement>
fi
```

- Semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one **non-deterministically**
  - If none are true, it is a runtime error
- Ex

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

  - If `i = 0` and `j > i`, this statement chooses non-deterministically between the first and third assignment statements
  - If `i` is equal to `j` and is not zero, a run-time error occurs because none of the condition are true

- Ex

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

  - This computes the desired result without over specifying the solution
  - In particular, if x and y are equal, it does **not** matter which we assign to max
  - This is a form of abstraction provide by the non-deterministic semantics of the statement

- The loop structure proposed by Dijkstra has the form

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
 ...
[] <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

- Ex Consider the following problem: Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that $q1 \le q2 \le q3 \le q$
  - **Without** guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables q1, q2, q3, and q4. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included.
  - Now, uses guarded commands to solve the same problem but in a more concise and **elegant** way

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

- Dijkstra's guarded command control statements are **interesting**, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa.
- Program verification is impossible when **goto** statements are used
- Verification is greatly simplified if
  - only selection and logical pretest loops
  - only guarded commands

-Summary

- Control statements occur in several categories:
  - Selection Statements
  - Iterative Statements
  - Unconditional branching
- The **switch** statement of the C-based languages is representative of multiple-selection statements
- C's **for** statement is the most flexible iteration statement although its flexibility lead to some reliability problem
- Data-based iterators are loop statements for processing data structures, such as linked lists, hashes, and trees.
  - The **for** statement of the C-based languages allows the user to create iterators for user-defined data
  - The **foreach** statement of Perl and C# is a predefined iterator for standard data structure
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
  - The unconditional branch, or goto, has been part of **most** imperative languages
  - Its problems have been widely discussed and debated.
  - The current consensus is that it should remain in most languages but that its **dangers** should be **minimized** through programming discipline
- Dijkstra's **guarded commands** are alternative control statement with positive theoretical characteristics.
  - Although they have not been adopted as the control statements of a language, part of the semantics appear in the concurrency mechanisms of CSP and the function definitions of Haskell

Chapter 13: Concurrency

-Intro

- Concurrency can occur at four levels:
  - Machine instruction level
  - High-level language statement level
  - Unit level
  - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here (4 levels, provided by OS)

-Multiprocessor Architecture

- Late 1950s – one general-purpose processor and one or more special-purpose processors for input and output operations
- Early 1960s – multiple complete processors, used for program-level concurrency
- Mid-1960s – multiple partial processors, used for instruction-level concurrency
- Single-Instruction Multiple-Data (SIMD) machines
- Multiple-Instruction Multiple-Data (MIMD) machines
  - Independent processors that can be synchronized (unit-level concurrency)

(SIMD, MIMD)

-Categories of Concurrency

- A *thread of control* in a program is the sequence of program points reached as control flows through the program
- Categories of Concurrency:
  - *Physical concurrency* – Multiple independent processors (multiple threads of control)
  - *Logical concurrency* – The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Coroutines (*quasi-concurrency) have a single thread of control*

(thread of control, physical, logical)

-Motivation for Concurrency

- Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
- Multiprocessor computers capable of physical concurrency are now widely used

-Sub-Program Concurrency

- A *task* or *process* is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
  - A task may be implicitly started
  - When a program unit starts the execution of a task, it is not necessarily suspended
  - When a task's execution is completed, control may not return to the caller
- Tasks usually work together

(task, process)

-2 Categories of Tasks

- *Heavyweight tasks* execute in their own address space
- *Lightweight tasks* all run in the same address space – more efficient
- A task is *disjoint* if it does not communicate with or affect the execution of any other task in the program in any way

-Task Synchronization

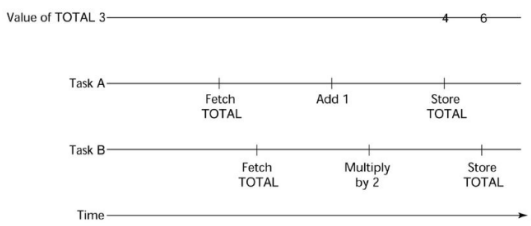
- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
  - *Cooperation* synchronization
  - *Competition* synchronization
- Task communication is necessary for synchronization, provided by:
  - Shared nonlocal variables
  - Parameters
  - Message passing

(Cooperation, Competition)

-Kinds of Synchronization

- *Cooperation*: Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem
- *Competition*: Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)

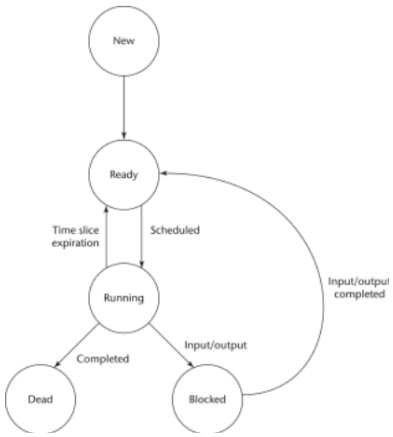
-Need for Competition Sync



-Scheduler

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the *scheduler*, which maps task execution onto available processors

-Task Execution States



- *New* – created but not yet started
- *R*ready – ready to run but not currently running (no available processor)
- *Running*
- *Blocked* – has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* – no longer active in any sense

-Liveness and Deadlock

- *Liveness* is a characteristic that a program unit may or may not have
  - In sequential code, it means the unit will eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

-Design Issues

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created

-Semaphors

- Dijkstra – 1965
- A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, *wait* and *release* (originally called *P* and *V* by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

-Cooperation Sync w/ Semaphores

- Example: A shared buffer
- The buffer is implemented as an ADT with the operations `DEPOSIT` and `FETCH` as the only ways to access the buffer
- Use two semaphores for cooperation: `emptyspots` and `fullspots`
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- `DEPOSIT` must first check `emptyspots` to see if there is room in the buffer
- If there is room, the counter of `emptyspots` is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of `emptyspots`
- When `DEPOSIT` is finished, it must increment the counter of `fullspots`

- FETCH must first check fullspots to see if there is a value
  - If there is a full spot, the counter of fullspots is decremented and the value is removed
  - If there are no values in the buffer, the caller must be placed in the queue of fullspots
  - When FETCH is finished, it increments the counter of emptyspots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named *wait* and *release*

-Wait & Release Operation and Producer & Consumer Code

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
   decrement aSemaphore's counter
else
   put the caller in aSemaphore's queue
   attempt to transfer control to a ready task
     -- if the task ready queue is empty,
     -- deadlock occurs
end
```

```
release(aSemaphore)
if aSemaphore's queue is empty then
   increment aSemaphore's counter
else
   put the calling task in the task ready queue
   transfer control to a task from aSemaphore's queue
end
```

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
   -- produce VALUE --
   wait (emptyspots); {wait for space}
   DEPOSIT(VALUE);
   release(fullspots); {increase filled}
   end loop;
end producer;
```

```
task consumer;
   loop
   wait (fullspots);{wait till not empty}}
   FETCH(VALUE);
   release(emptyspots); {increase empty}
   -- consume VALUE --
   end loop;
end consumer;
```

-Competition Sync w/ Semphores

- A third semaphore, named access, is used to control access (competition synchronization)
  - The counter of **access** will only have the values 0 and 1
  - Such a semaphore is called a *binary semaphore*
- Note that wait and release must be atomic!

-Producer & Consumer Code

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
   -- produce VALUE --
   wait(emptyspots); {wait for space}
   wait(access);     {wait for access}
   DEPOSIT(VALUE);
   release(access); {relinquish access}
   release(fullspots); {increase filled}
   end loop;
end producer;
```

```
task consumer;
   loop
   wait(fullspots);{wait till not empty}
   wait(access);   {wait for access}
   FETCH(VALUE);
   release(access); {relinquish access}
   release(emptyspots); {increase empty}
   -- consume VALUE --
   end loop;
end consumer;
```

-Evaluation

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of fullspots is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of access is left out

-Monitors

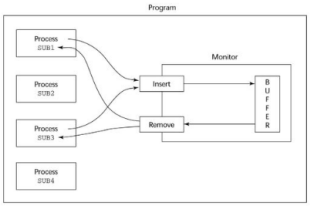
- Ada, Java, C#
- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

-Competition Sync

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
  - Monitor implementation guarantee synchronized access by allowing only one access at a time
  - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

-Cooperative Sync

- Cooperation between processes is still a programming task
  - Programmer must guarantee that a shared buffer does not experience underflow or overflow



-Evaluation

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

-Message Passing

- Message passing is a general model for concurrency
  - It can model both semaphores and monitors
  - It is not just for competition synchronization
- Central idea: task communication is like seeing a doctor--most of the time she waits for you or you wait for her, but when you are both ready, you get together, or *rendezvous*

-Rendezvous

- To support concurrent tasks with message passing, a language needs:
  - A mechanism to allow a task to indicate when it is willing to accept messages
  - A way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message
- When a sender task's message is accepted by a receiver task, the actual message transmission is called a *rendezvous*

-Java Threads

- The concurrent units in Java are methods named `run`
  - A `run` method code can be in concurrent execution with other such methods
  - The process in which the `run` methods execute is called a *thread*

```
Class myThread extends Thread
  public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

-Controlling Thread Execution

- The `Thread` class has several methods to control the execution of threads
  - The `yield` is a request from the running thread to voluntarily surrender the processor
  - The `sleep` method can be used by the caller of the method to block the thread
  - The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution

-Thread Priority

- A thread's default priority is the same as the thread that create it
  - If `main` creates a thread, its default priority is `NORM_PRIORITY`
- Threads defined two other priority constants, `MAX_PRIORITY` and `MIN_PRIORITY`
- The priority of a thread can be changed with the methods `setPriority`

-Competition Sync

- A method that includes the `synchronized` modifier disallows any other method from running on the object while it is in execution

```
...
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

- The above two methods are synchronized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized thru `synchronized` statement

```
synchronized (expression)
   statement
```

-Cooperative Sync

- Cooperation synchronization in Java is achieved via `wait`, `notify`, and `notifyAll` methods
  - All methods are defined in `Object`, which is the root class in Java, so all objects inherit them
- The `wait` method must be called in a loop
- The `notify` method is called to tell one waiting thread that the event it was waiting has happened
- The `notifyAll` method awakens all of the threads on the object's wait list

-Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks