NOTES ON CS403. PROGRAMMING PARADIGMS

MODULE-1

TOPICS:

Names, Scopes and Bindings:-

1. Names
2. Scopes
3. Binding Time
4. Scope Rules
5. Storage Management
6. Binding of Referencing Environments

Control Flow: -

7. Expression Evaluation
8. Structured and Unstructured Flow
9. Sequencing
10. Selection
11. Iteration
12. Recursion
13. Non-determinacy

NOTES:

Introduction to Programming paradigms:

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms. Common programming paradigms include:

- **Imperative** which allows side effects,
  - **Procedural** which groups code into functions(eg: **C, FORTRAN, Pascal, COBOL,BASIC, Ada etc**)
  - **Object-oriented** which groups code together with the state the code modifies.(eg:**C++,C#, Java, Simula,Smalltalk**)
- **Declarative** which does not state the order in which operations execute,
  - **Functional** which disallows side effects(eg: **Lisp, Scheme, Haskell, Erlang, ML,APL,Miranda,etc)**
  - **Logic** which has a particular style of execution model coupled to a particular style of syntax and grammar(eg:**Prolog**)
- **Symbolic** programming which has a particular style of syntax and grammar

1. Names

- A name is a mnemonic character string used to represent something else. Names in most languages are identifiers (alphanumeric tokens), though certain other symbols, such as + or :=, can also be names.
- Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers rather than low-level concepts like addresses.
- Names are also essential in the context of a second meaning of the word abstraction.
- In this second meaning, abstraction is a process by which the programmer associates a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of how that function is achieved.
- By hiding irrelevant details, abstraction reduces conceptual complexity, making it possible for the programmer to focus on a manageable subset of the program text at any particular time.
- Names are control abstractions and data abstractions for program
- Subroutines are control abstractions: they allow the programmer to hide arbitrarily complicated code behind a simple interface.
- Classes are data abstractions: they allow the programmer to hide data representation details behind a (comparatively) simple set of operations.

2. Scope

- A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within whicha declaration has effect.
- The scope of a binding is also known as the **visibility** of an entity.
- **Static scope**: Scoping follows the structure of the program. C is said to be *statically scoped.*
- **Dynamic scope**, where scoping follows the execution path.the languages, including APL, Snobol, and early versions of Lisp, are dynamically scoped: their bindings depend on the flow of execution at run time.

Example:

```
int i = 1;          //global variable

void printdata()
{
cout << i << endl;
}

int main ( )
{
int i = 2;
printdata();
return 0;
}
```

If static scoping used, the result will be 1

If dynamic scoping used, this would print out 2.

- The lexical scope of a variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on.
- Whereas with dynamic scope the calling function is searched, then the function which called that calling functions, and so on, progressing up the call stack. Of course, in both rules, we first look for a local definition of a variable.

- Most modern languages use lexical scoping for variables and functions, though dynamic scoping is used in some languages, notably some dialects of Lisp, some "scripting" languages, and some template languages.Perl 5 offers both lexical and dynamic scoping.

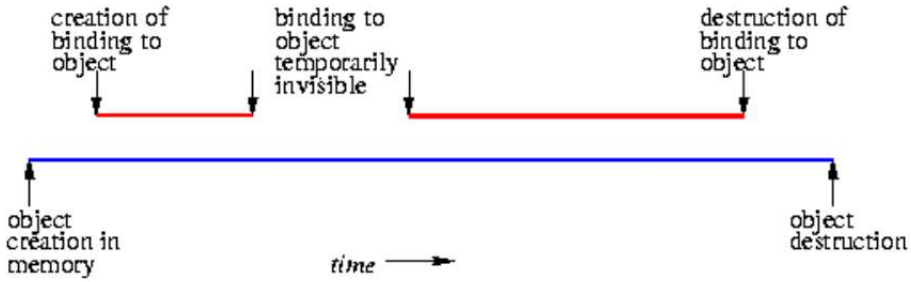
3. Binding Time

- A **binding** is an association between two things, such as a name and the thing it names.
- **Binding time** is the time at which a binding is created or, more generally, the time at which any implementation decision is made to create a name ❻❼ entity binding.
- There are many **different times at which decisions may be bound:**
  - ✓ **Language design time**: the design of specific program constructs (syntax), primitive types, and meaning (semantics).
  - ✓ **Language implementation time**: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc.
  - ✓ **Program writing time**: Programmers, of course, choose algorithms, data structures,and names.
  - ✓ **Compile time**: Compilers choose the mapping of high-level constructs to machinecode, including the layout of statically defined data in memory.
  - ✓ **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable
  - ✓ **Load time**: when the operating system loads the executable in memory
  - ✓ **Run time**: when a program executes

Language feature	*Binding time*
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada	Language design
Keywords, e.g. `class` in C++ and Java	Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. `float` and `struct` in C	Language design
Internal representation of literals, e.g. `3.1` and `"foo bar"`	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. `printf` in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

The Effect of Binding Time

- The compiler performs a process called binding when an object is assigned to an object variable. The early binding (static binding) refers to compile time binding and late binding (dynamic binding) refers to runtime binding. In general, early binding times are associated with greater efficiency, while later binding times are associated with greater flexibility.
- Binding lifetime: time between creation and destruction of binding to object.
    - Example: A pointer variable is set to the address of an object
    - Example: A formal argument is bound to an actual argument
- Object lifetime: time between creation and destruction ofan object.



- Key events in object lifetime
    - Object creation
    - Creation of bindings
    - References to variables, subroutines, types are made using bindings
    - Deactivation and reactivation of temporarily unusable bindings
    - Destruction of bindings
    - Destruction of objects

- Bindings are temporarily invisible when code is executed where thebinding (name ●● object) is out of scope
- Memory leak: object never destroyed (binding to object may havebeen destroyed, rendering access impossible)
- Dangling reference: object destroyed before binding is destroyed
- Garbage collection prevents these allocation/deallocation problems

4. Scope Rules

The textual region of the program in which a binding is active is its scope. In most modern languages, the scope of a binding is determined statically, that is at compile time. A scope is the body of a module, class, subroutine, or structured control flow statement, sometimes called a block. In C family languages it would be delimited with {...} braces.

Statically scoped language: the scope of bindings is determined at compile time.
Dynamically scoped language: the scope of bindings is determined at run time.

➔ *Static Scoping*

The bindings between names and objects can be determined by examination of the program text. Scope rules of a program language define the scope of variables and subroutines, which is the region of program text in which a name-to-object binding is usable.

- **EarlyBasic**:

- all variables are **global** and visible everywhere.

- **Fortran77**:
  - the scope of a local variable is limited to a subroutine;
  - the scope of a **global**variable is the whole program text unless it is hidden by a **local** variable declaration with thesame variable name.
- **Algol60**, **Pascal**, and **Ada**:
  - these languages allow nested subroutines definitions and adopt the **closest nested scope rule** with slight variations in implementation.
- **C**: one declares the variable **static**; A**lgol:** one declares it **own**.
  - A save-ed (static, own) variable has a lifetime that encompasses the entire execution of the program. Instead of a logically separate object for every invocation of the subroutine, the compiler creates a single object that retains its value from one invocation of the subroutine to the next.

➔ ***Nested Subroutines***

✓ The ability to nest subroutines inside each other, introduced in Algol 60, is a feature of many modern languages, including Pascal, Ada, ML, Python, Scheme, Common Lisp, and (to a limited extent) Fortran 90.
✓ Other languages, including C and its descendants, allow classes or other scopes to nest.
✓ But C based languages has no nested subroutines
✓ In Algol-family languages. Algol-style nesting gives rise to the closest nested scope rule for bindings from names to objects:
  ✓ A name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is hidden by another declaration of the same name in one or more nested scopes.
  ✓ To find the object corresponding to a given use of aname, we look for a declaration with that name in the current, innermost scope.
  ✓ If there is one, it defines the active binding for the name.
  ✓ Otherwise, we look for a declaration in the immediately surrounding scope.
  ✓ We continue outward, examining successively surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared.
  ✓ If no declaration is found at any level, then the program is in error.
✓ In Pascal, a procedure is defined using the **procedure** keyword. The general form of a procedure

In this below example, procedure Nested scopes P2 is called only by P1, and need not be visible outside. It is therefore declared inside P1, limiting its scope (its region of visibility) to the portion of the programshown here.

- In a similar fashion, P4 is visible only within P1
- P3 is visible only within P2
- and F1 is visible only within P4.
- Under the standard rules for nested scopes, F1 could call P2 .,
- and P4 could call F1.,
- but P2 could not call F1.
- Though they are hidden from the rest of the program, nested subroutines are able to access the parameters and local variables (and other local objects) of the surrounding scope(s). In our example, P3 can name (and modify) A1, X, and A2, in addition to A3.

- Because P1 and F1 both declare local variables named X, the inner declaration hides the outer one within a portion of its scope.
- Uses of X in F1 refer to the inner X; uses of X in other regions of the code refer to the outer X.

```
procedure P1(A1:T1)
var X:real;
...
  procedure P2(A2:T2);
  ...
    procedure P3(A3:T3);
    ...
    begin
    (* body of P3: P3,A3,P2,A2,X of P1,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P2: P3,P2,A2,X of P1,P1,A1 are visible *)
  end;
  procedure P4(A4:T4);
  ...
    function F1(A5:T5):T6;
    var X:integer;
    ...
    begin
    (* body of F1: X of F1,F1,A5,P4,A4,P2,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P4: F1,P4,A4,P2,X of P1,P1,A1 are visible *)
  end;
...
begin
(* body of P1: X of P1,P1,A1,P2,P4 are visible *)
end
```

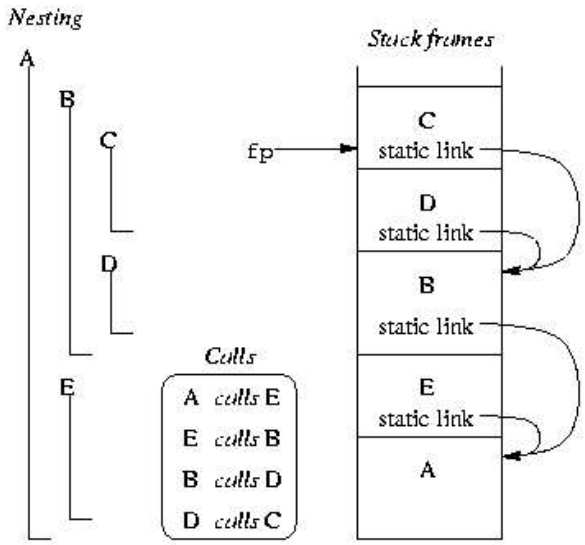
- To find the object referenced by a given name:
  - Look for a declaration in the current innermost scope
  - If there is none, look for a declaration in the immediately surrounding scope, etc.

Static Scope Implementation with static links

- Scope rules are designed so that we can only refer to variables that are alive: the variable must have been stored in the frame of a subroutine

- If a variable is not in the local scope, we are sure there is a frame for the surrounding scope somewhere below on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active
- The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the "parent frame".Each frame on the stack contains a static link pointing tothe frame of the static parent.

- Example static links.



- Subroutines C and D are declared nested in B.
- B is static parent of C and D.
- B and E are nested in A.
- A is static parent of B and E.
- The fp(frame pointer) points to the frame at the top of the stack.
- The static link in the frame points to the frame of the static parent.
- If a subroutine is declared at the outermost nesting level of the

Static Chains

- How do we access non-local (global) objects?
- The static links form a static chain, which is a linked listof static parent frames
- When a subroutine at nesting level j has a reference to an object declared in a static parent at the surrounding scope nested at level k, then j-k static links forms a static chain that is traversed to get to the frame containing theobject.
- The compiler generates code to make these traversals over frames to reach non-local objects

(3) Subroutine A is at nesting level1 and C at nesting level 3
(4) When C accesses an object of A, 2 static links are traversed to get to A's frame thatcontains that object

Out of Scope

- Non-local objects can be hidden by local name-to-object bindings and the scope is said to have a "***hole***" in which the non-local binding is temporarily inactive but notdestroyed.
- Some languages, notably Ada and C++ use qualifiers or scope resolution operators to access non-local objectsthat are hidden

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

➔ *Dynamic Scoping*

- Scope rule: the "current" binding for a given name is the one encountered most recently during execution.
- Typically adopted in (early) functional languages that are interpreted

- Languages with Dynamic scoping:
  - APL, Snobol, TEX, early versions of Lisp, Perl
- Perl v5 allows you to choose scope method for each variableseparately
- With dynamic scope:
  - Name-to-object bindings cannot be determined by a compiler in general
  - Easy for interpreter to look up name-to-object binding in a stack ofdeclarations
- Sometimes useful:
  - Unix environment variables have dynamic scope
  - It makes it very easy for an interpreter to look up the meaning of a name
  - All that is required is a stack of declarations.
- Generally considered to be "a bad programming language feature"
  - Hard to keep track of active bindings when reading a program text
  - Most languages are now compiled, or a compiler/interpreter mix
  - Unfortunately, this simple implementation has a very high run-time cost, and experience indicates that dynamic scoping makes programs harder to understand.
  - The modern consensus seems to be that dynamic scoping is usually a bad idea.

```
1.  n : integer              -- global declaration

2.  procedure first
3.      n := 1

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

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

Static versus dynamic scoping.

Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.

- If **static scoping** is in effect, the above program Static vs. dynamic scoping
  - Prints a 1.
- If **dynamic scoping** is in effect, the output depends on the value read at line 8 at run time:
  - if **the input is positive**, the program **prints a 2**;
  - if **the input is negative or 0.,,** the program **prints a 1.**
- Why the difference? At issue is whether the assignment to the variablen at line 3 refers to the global variable declared at line 1 or to the local variabledeclared at line 5.
- **Static scope rulesrequire** that the reference resolve to the **closestlexically enclosing declaration**, namely the global n. Procedure first changes n to1, and line 12 prints this value.
- **Dynamic scope rules**, on the other hand, **require**that we choose the **most recent, active binding for n at run time.**

Problems with Dynamic Scoping

- Run-time errors with dynamic scoping:With dynamic scoping, errors associated with the referencing environment maynot be detected until run time.
- For example, the declaration of local variable max_score in procedure foo accidentally redefines a global variable used by function scaled_score, which is then called from foo. Since the global max_score is an integer, while the local max_score is a floating-point number, dynamic semantic checks in at least some languages will result in a type clash message at run time.
- If the local max_score had been an integer, no error would have been detected, but the program would almost certainly have produced incorrect results.
- This sort of error can be very hard to find.
- In this example, function scaled_score probably does not do what the programmer intended: with dynamic scoping, max_score in scaled_score is bound to foo's local variable max_score after foo calls scaled_score, which was the most recent bindingduring execution.

```
max_score : integer        -- maximum possible score

function scaled_score(raw_score : integer) : real
    return raw_score / max_score * 100
...
procedure foo
    max_score : real := 0        -- highest percentage seen so far
    ...
    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent
```

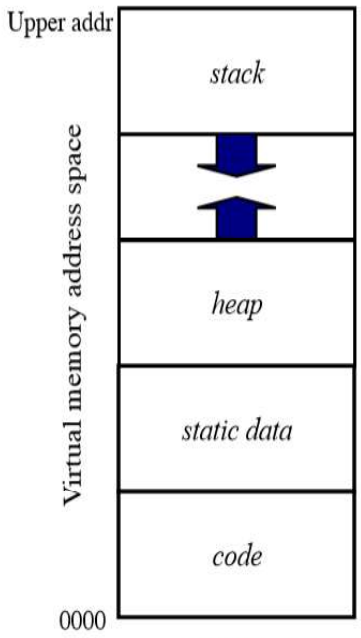
The problem with dynamic scoping.
Procedure scaled_score probably does not do what the programmer intended when dynamic scope rules allow procedure foo to change the meaning of max_score.

➔ *Implementing Scope*

- To keep track of the names in a **statically scoped program**, a compiler relies on a data abstraction called a **symbol table**. In essence, the symbol table is a dictionary: it maps names to the information the compiler knows about them.
- In a language with **dynamic scoping**, an **interpreter** (or the output of a compiler) must perform operations analogous to symbol table insert and lookup at runtime. In principle, any organization used for a symbol table in a compiler could be used to track name-to-object bindings in an interpreter, and vice versa. In practice, implementations of dynamic scoping tend to adopt one of two specific organizations: **an association list or a central reference table.**

5. Storage Management

Objects (program data and code) have to be stored in memory during their lifetime. Object lifetimes generally correspond to one of three principal storage allocation mechanisms, used to manage the object's space. They are static allocation, stack allocation, and heap allocation.



- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

FIGURE: TYPICAL PROGRAM AND DATA LAYOUT IN MEMORY

(1) STATIC ALLOCATION

Static Objects have an absolute storage addressthat is retained throughout the execution of the program. (**Absolute addresses are also called real addresses and machine addresses. A fixed address in memory).**They are often allocated in protected, read-only memory. So that any attempt to write to them will cause a processor interrupt, allowing the operating system to announce a run-time error.**Advantage** of statically allocated object is the **fast access** due to absolute addressing of the object.

Examples of static objects are:

- ✓ Global variables are static objects
- ✓ Program code is statically allocated in most implementations of imperative languages.
- ✓ Variables that are local to single subroutine, but retain their values from one invocation to the next; their space is statically allocated.
- ✓ Numeric and string valued constants can be allocated statically.
- ✓ Run time tables produced by compilers (these tables are used for support of debugging, type checking, garbage collection, exception handling and other purposes) are statically allocated.
- ✓ static local variables in C
- ✓ In earlier versions of FORTRAN, recursion was not supported.As a result, there can never be more than one invocation of subroutine active. Thus the compiler may choose to use static allocation for local variables in that language. In languages with recursion, we can't use static allocation of local variables.

(2) STACK ALLOCATION

Static allocation does not work for local variables in potentially recursive subroutines. Every (recursive) subroutine call must haveseparate instantiations of local variables. So we use stack allocation.**Stack Objects**are allocated in last-in first-out(LIFO) order, usually in conjunction with subroutine calls and returns**.**

- ✓ Each instance of a subroutine at run time has a frame on the run-time stack (also called activation record).
- ✓ Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards.
- ✓ Frame layouts vary between languages and implementations
- ✓ A frame pointer (fp) points to the frame of the currently active subroutine at run time (always topmost frame on stack)
- ✓ Subroutine arguments, local variables, and return values are accessed by constant address offsets from fp
- ✓ The stack pointer (sp) points to free space on the stack.
- ✓ Even in a language without recursion, it can be advantageous to use a stack for local variables, rather than allocate them statically.

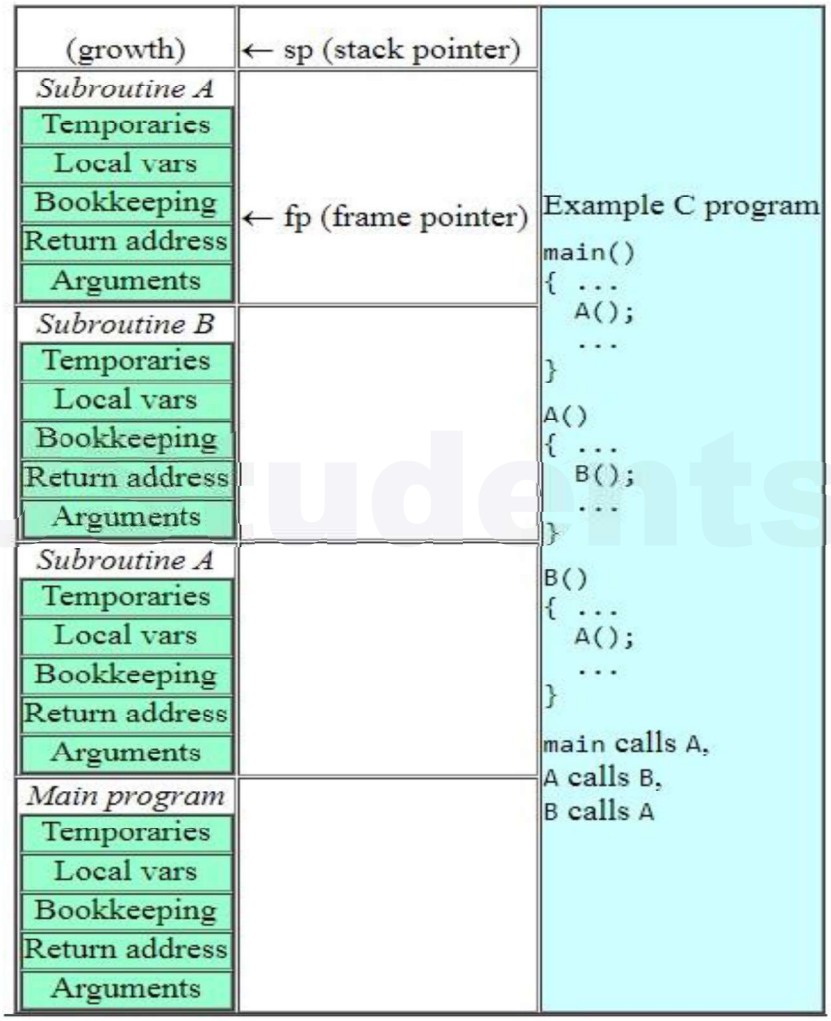


FIGURE: STACK BASED ALLOCATION OF SPACE FOR SUBROUTINES

(3) HEAP ALLOCATION

A heap is a region of storage in which sub-blocks can be allocated and de-allocated at arbitrary times. Heaps are required for the dynamically allocated pieces of linked data structures, and for objects like fully general character strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation.

Strategies to manage space in a heap

- The principal concerns are **speed and space**, and as usual there are tradeoffs between them.
- Space concerns can be further subdivided into issues of **internal and external fragmentation**

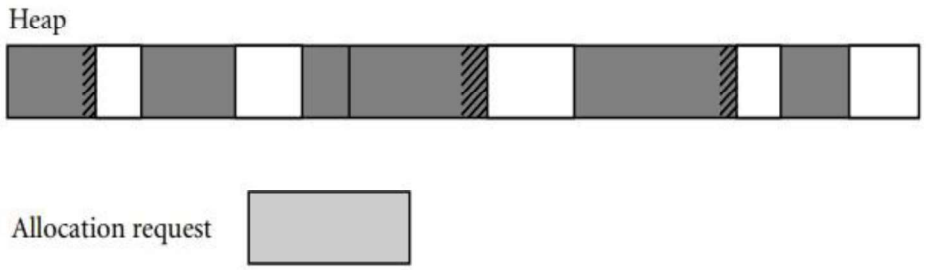


Figure : **Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represents internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

ternalfragmentation:

ɔccurs when a storage-management algorithm allocates a block that is larger than required to hold a given object. The extra space is then unused.

ternalfragmentation:

ɔccurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some future request.

Many **storage-management algorithms** maintain a **single linked list—the free list**—of heap blocks not currently in use.

- Initially the list consists of a single block comprising the entire heap.
- At each allocation request the algorithm searches the list for a block of appropriate size.
- With a ***first fit algorithm*** we select the first block on the list that is large enough to satisfy the request.
- With a ***best fit algorithm*** we search the entire list to find the smallest block that is large enough to satisfy the request.
- In either case, if the chosen block is significantly larger than required, then we divide it in two and return the unneeded portion to the free list as a smaller block. (If the unneeded portion is below some minimum threshold in size, we may leave it in the allocated block as internal fragmentation.)
- When a block is de-allocated and returned to the free list, we check to see whether either or both of the physically adjacent blocks are free; if so, we coalesce them.
- Intuitively, one would expect a ***best fit algorithm to do a better job*** of reserving large blocks for large requests. At the same time, ***it has higher allocation cost than a first fit algorithm***, because it must always search the entire list, and it tends to result in a larger number of very small "left-over" blocks.
- Which approach—first fit or best fit—results in lower external fragmentation depends on the distribution of size requests.

In any algorithm that maintains a single free list, the cost of allocation is linear in the number of free blocks. **To reduce this cost to a constant**, some storage management algorithms ***maintain separate free lists for blocks of different sizes***.

- ✓ Each request is rounded up to the next standard size (at the cost of internal fragmentation) and allocated from the appropriate list.
- ✓ In effect, the ***heap is divided into*** "***pools***," ***one for each standard size***.
- ✓ The division may be static or dynamic.

Two common mechanisms for dynamic pooladjustment are known as the **buddy system** and the **Fibonacci heap**.

- ✓ In **the buddy system**, the standard block sizes are powers of two. If a block of size 2^k is needed, but none is available, a block of size 2^{k+1} is split in two. One of the halves is used to satisfy the request; the other is placed on the k^{th} free list. When a block is deallocated, it is coalesced with its "buddy"—the other half of the split that created it—if that buddy is free.
- ✓ **Fibonacci heaps** are similar, but use Fibonacci numbers for the standard sizes, instead of powers of two. The algorithm is slightly more complex, but leads to slightly lower internal fragmentation, because the Fibonacci sequence grows more slowly than2^k**.**
- ✓ The problem with external fragmentation is that the ability of the heap to satisfy requests may degrade over time. To eliminate external fragmentation, we must be prepared to ***compactthe heap***, by moving already-allocated blocks. This task is complicated by the need to find and update all outstanding references to a block that is being moved.

What happens when the user no longer needs the heap-allocated space?

- ✓ Manual de-allocation: The user issues a command like free or delete to return the space (C, Pascal). When a block (including any internal wasted space) is returned, it is coalesced, if possible, with any adjacent free blocks.
- ✓ Automatic de-allocation via garbage collection (Java, C#, Scheme, ML, Perl).
- ✓ Semi-automatic de-allocation using destructors (C++, Ada). The destructor is called automatically by the system, but the programmer writes the destructor code.

Poorly done manual de-allocation is a common programming error.

- If an object is de-allocated and subsequently used, we get a dangling reference.
- If an object is never de-allocated, we get a memory leak.

We can run out of heap space for at least three different reasons.

- What if there is not enough free space in total for the new request and all the currently allocated space is needed?
  Solution: Abort.
- What if we have several, non-contiguous free blocks, none of which are big enough for the new request, but in total they are big enough?
  This is called external fragmentation since the wasted space is outside (external to) all allocated blocks.
  Solution: Compactify.
- What if some of the allocated blocks are no longer accessible by the user, but haven't been returned?
  Solution: Garbage Collection

GARBAGE COLLECTION

- A *garbage collection* algorithm is one that automatically de-allocates heap storage when it is no longer needed.
- There are two aspects to garbage collection: first, determining automatically what portions of heap allocated storage will (definitely) not be used in the future, and second making this unneeded storage available for reuse.

Disadvantages of Garbage Collection:

- Extra burden for the language implementer.
- When the garbage collector is running, machine resources are being consumed.
- For some programs the garbage collection overhead can be a significant portion of the total execution time.
- The programmer can easily tell when the storage is no longer needed, it is normally much quicker for the programmer to free it manually than to have a garbage collector do it.

6. Binding of Referencing Environments

What is Referencing Environment?

- It is the collection of all names that are visible in the statement.
- In a static-scoped language, referencing Environments 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 non-local variables in both static and dynamic scoped languages.
- 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.