

CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the Winter 2013/2014 term at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

©David B. Wortman, 2003,2004,2008,2009,2010,2012,2013,2014

©Marsha Chechik, 2005,2006,2007

Reading Assignment

Fischer, Cytron and LeBlanc

Chapter 12

Runtime Organization

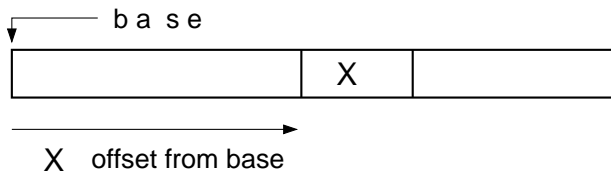
- Storage Mapping for variables
scalars, arrays, records, structures, unions
- Runtime storage organization and addressing
address, global storage, local storage (activation records)
- Display based addressing
display update algorithms
- Dynamically created variables, associative data structures
- Dynamic Storage Management

Storage Allocation for Variables

- Variables declared in a procedure or function are usually allocated storage in the local storage of the procedure or function in the order that they are declared.
- Scalar variables are simply allocated storage based on the underlying hardware container (byte, word, double word) that is used to implement the variables type.
- More complicated data structures (arrays, records, unions) require the compiler to provide a *mapping algorithm* to for access to information stored in the data structure (e.g. elements of an array, fields in a record).
- The mapping algorithm is related to the way memory is allocated for data objects.

Addressing for Structured Data

- The compiler must implement access to non-scalar data objects.
e.g. arrays, records, unions
- General case:**
Need to be able to access the contents of a data object given only:
 - The address of the beginning of the object in memory.
 - A *compatible* type declaration for the object.
 - Compatible* is a language-specific property.
- Must implement the most general case of object access allowed in the language.



255

Array Access Mapping

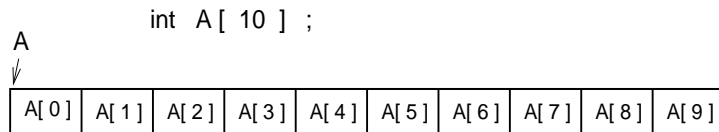
- In most modern languages (except Fortran), arrays are laid out in memory in *row major order*, i.e. the rightmost subscript of consecutive array elements varies most rapidly.
- Because arrays can be passed as parameters and/or allocated in dynamic storage, the compiler needs to be able to calculate the address of an arbitrary element of an array given only the base address of the array and information from the declaration of the array.
- If the basic unit of addressing (e.g. bytes) is different from the size of the array element (e.g. words, double words) then each subscript must be *scaled* by the size of the array element in address units.
- Generic array declaration:

$$\text{type}A[L_1 : U_1 , L_2 : U_2 , \dots L_n : U_n]$$
- Generic array reference:

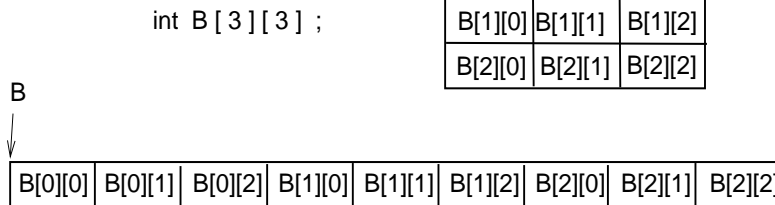
$$A[E_1 , E_2 , \dots E_n]$$

256

One Dimensional Array



Two Dimensional Array



257

Array Access Mapping

- Define: **base address** of an array

$$\text{base} = \text{addr}(A[L_1 , L_2 , \dots L_n])$$
- Define: **stride** in the i_{th} dimension

$$\text{stride}_i = U_i - L_i + 1$$
- Then the address of an arbitrary element $A[E_1 , E_2 , \dots E_n]$ is

$$\text{addr}(A[E_1 , E_2 , \dots E_n]) = \text{base} + \sum_{i=1}^n ((E_i - L_i) \cdot \prod_{j=i+1}^n \text{stride}_j)$$

- To simplify, let: mult_n be the size of an array element (in address units)
and $\text{mult}_i = \prod_{j=i+1}^n \text{stride}_j = \text{stride}_{i+1} \cdot \text{mult}_{i+1}$ then

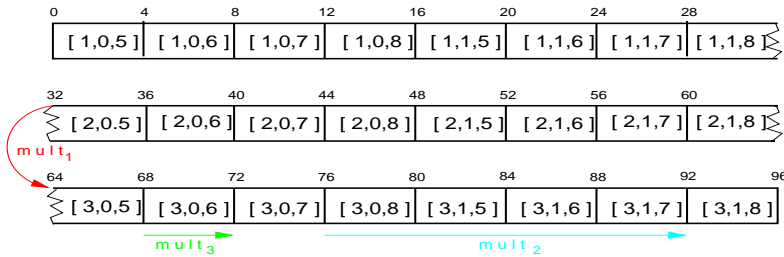
$$\text{addr}(A[E_1 , E_2 , \dots E_n]) = \text{base} + \sum_{i=1}^n ((E_i - L_i) \cdot \text{mult}_i)$$

and mult_0 is the total size of a compact array.

258

Array Subscripting Example

real B[1 : 3 , 0 : 1 , 5 : 8] ;				
i	L_i	U_i	$stride_i$	$mult_i$
0				96
1	1	3	3	32
2	0	1	2	16
3	5	8	4	4



259

Calculating Array Subscripts Efficiently

- The obvious array subscript calculation based on

$$addr(A[E_1, E_2, \dots, E_n]) = base + \sum_{i=1}^n ((E_i - L_i) \cdot mult_i)$$

Can be done using two registers as

$$R_a \leftarrow base$$

$$R_b \leftarrow (E_1 - L_1)$$

$$R_b \leftarrow R_b \cdot mult_1$$

$$R_a \leftarrow R_a + R_b$$

...

- This calculation can be simplified further if all of the L_i are zero (e.g. as in C) or are compile time constants (e.g. as in Turing)

- Horners Rule (to optimize array subscript calculation)

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots \text{ may be written as}$$

$$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots)))$$

260

- Given

$$addr(A[E_1, E_2, \dots, E_n]) = base + \sum_{i=1}^n ((E_i - L_i) \cdot \prod_{j=i+1}^n stride_j)$$

Apply Horner's rule for evaluating polynomials and factor

$$constPart = (((L_1 \cdot stride_2 + L_2) \cdot stride_3 + \dots L_{n-1}) \cdot stride_n + L_n)$$

$$varPart = (((E_1 \cdot stride_2 + E_2) \cdot stride_3 + \dots E_{n-1}) \cdot stride_n + E_n)$$

then

$$addr(A[E_1, E_2, \dots, E_n]) = base - constPart + varPart$$

- Using this formulation, an array subscript can be calculated using one register

$$R_a \leftarrow E_1$$

$$R_a \leftarrow R_a \cdot stride_2$$

$$R_a \leftarrow R_a + E_2$$

...

$$R_a \leftarrow R_a + (base - constPart)$$

261

Array Subscript Checking

- An arbitrary array subscript $A[E_1, E_2, \dots, E_n]$ is in range if $\forall_i (L_i \leq E_i \leq U_i)$
- An alternative formulation which is often easier to check as a part of the array subscript calculation is $\forall_i ((E_i - L_i) \geq 0 \wedge (E_i - L_i) < stride_i)$

- Example: $A[E_1]$

LOAD	R_a, E_1	Calculate E_1
SUBI	R_a, L_1	$E_1 - L_1$
BNEG ^a	SUBERR	Error $E_1 < L_1$
CMPI ^a	$R_a, stride_1$	$(E_1 - L_1) : stride_1$
BGE ^a	SUBERR	Error $E_1 > U_1$
...		

^aA really clever compiler would replace these instructions with an unsigned compare

UCMP	$R_a, stride_1$
BGE	SUBERR

262

Dynamic Arrays

- In many languages, the lower and/or upper bounds of an array in $typeA[L_1 : U_1, L_2 : U_2, \dots, L_n : U_n]$ can be run time expressions, the size of the array is determined when these expressions are evaluated at run time.
- At the point of declaration
 - Generate a runtime data structure to hold array subscript information and add it to the local storage of the nearest enclosing major scope.
 - Emit code to calculate L_i, U_i
 - Emit code (routine call ?) to validate L_i and U_i , and calculate $stride_i$.
 - Emit code to store L_i and $stride_i$ in the runtime data structure.
If all L_i are compile time constants then only $stride_i$ is required.
Save U_i if required for runtime subscript checking.
 - Emit code to allocate storage for the array at runtime using the information in the runtime data structure
- If the language allows subarrays (e.g. PL/I, APL) then saving $mult_i$ instead of $stride_i$ makes subarray computation much easier.

263

Sub Arrays

- If $typeA[L_1 : U_1, L_2 : U_2, \dots, L_3 : U_3]$ is a 3 dimensional array, then it has the subarrays:

$A[I, J, K]$ $A[I, *, *]$ $A[*, J, *]$ $A[*, *, K]$
 $A[*, J, K]$ $A[[I, *, K]$ $A[I, J, *]$ $A[*, *, *]$

- For an array reference $A[E_1, E_2, \dots, E_n]$
Assume each array subscript E_i is either an expression or *. Calculate

$$newBase = base + \sum_{i=1}^n (if E_i = * then 0 else (E_i - L_i) * mult_i)$$

For each E_i which is *, copy runtime data structure entry for that dimension to a new run time data structure.

- Dimension of the sub array is number of subscripts where $E_i = *$
- Unless the * occur only in the rightmost subscript positions, the subarray is **not compact**, i.e. logically consecutive subarray elements are not in consecutive memory locations.

265

- At runtime when the declaration is encountered
 - Execute the code to compute L_i, U_i and $stride_i$
 - Allocate storage for the array and save the *base* address of the array.
- At runtime for each array subscript
 - Access runtime data structure to get the *base* address of the array
 - Use L_i and $stride_i$ to calculate address of array element

264

Array and Subarray Example

- The array declaration **int** A[1:4, 0:3, -10:10, 1:12] has the runtime data structure

base	4	16128
1	4	4032
0	3	1008
-10	10	48
1	12	4

- The runtime data structure for the subarray A[*, 2, *, 6] is

base+2036	2	48384
1	4	4032
-10	10	48

- This subarray is **not compact**.

266

Array and Subarray Example

Integer A[1 : 2, -1 : 1, 0 : 1, 3 : 4] ;

base A	4	96
1	2	48
-1	1	16
0	1	8
3	4	4

@ A[2, 0, 1, 4] =

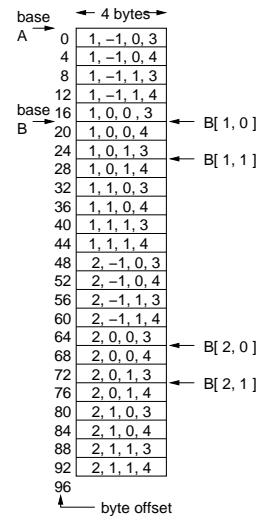
$$\text{baseA} + (2 - 1) * 48 + (0 - -1) * 16 + (1 - 0) * 8 + (4 - 3) * 4$$

$$\text{baseA} + 76$$

Let B = A[*, 0, *, 4]

base A+ 20	4	96
1	2	48
0	1	8

@ B[2, 1] = (baseA + 20) + (2 - 1) * 48 + (1 - 0) * 8

$$\text{baseA} + 76$$


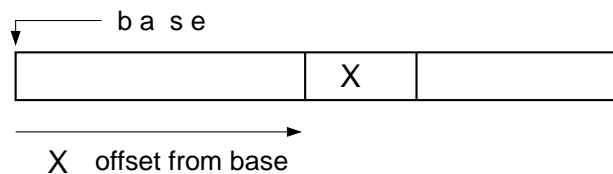
267

Record and Structures

- It must usually be possible to access any field of a structure given only the base address of the structure and a *compatible* declaration for the structure.
- There are several possible definitions for *compatible declaration*
 - Strict Equivalence** The declaration used to access the structure must be the one that was used to declare the structure. (Pascal, Turing).
 - Left to Right Equivalence** The declaration used to access the structure needs to match the structure declaration only up to the field being accessed. (C, P/I/I)
 - Implies:** Must map structure fields in left-to-right order.
 - Position of field can not depend on following fields.
 - Major Minor Equivalence** Access to any embedded substructure in a structure must be possible given only the address of the substructure and a declaration for the substructure. (most languages)
 - Implies:** Must use same algorithm for structures and contained substructures
 - Position of fields in a substructure can only depend on the substructure.
- If the same mapping algorithm is used consistently on all records/structures then Structural Type Equivalence (Slide 210) is supported.

268

- A record or struct is a collection of *fields*.
Storage must be allocated for the structure and the layout of the fields in the structure must be computed.
- Storage for records and structures can be *packed* or *unpacked*. Data items in a structure can be aligned on their natural hardware boundaries or they can be *unaligned*. Packing usually implies unaligned data. Access to unaligned data is usually much slower than access to aligned data.
- Once the record or structure has been mapped, the offset of each field relative to the start of the record or structure is stored in the compilers symbol table and used to generate the address of the field.



269

Storage Mapping for Records and Structures

- For most languages, the fields of a structure are laid out in memory in left to right order. *Internal fill* is introduced between scalar items as required so that scalar items are properly aligned on their natural hardware boundaries.
- Most structure storage mapping algorithms try to achieve
 - Proper storage alignment for all scalar data items.
 - Economical use of space, i.e. minimize internal fill.
 - Use *no* internal mapping information (embedded pointers)
 - Allow access to field given only base address and compatible declaration.
 - Support Major/Minor equivalence.
 - Support Left/Right equivalence if required by the programming language.
- If a structure is packed (data items are unaligned) then the fields in a record are laid out in order with no intervening fill

270

Storage Mapping and Alignment

- There are several possible algorithms for laying out fields in a record. They differ in the sophistication of their fill minimization strategies.

- Definitions for storage alignment

$address_i$ is the relative address (in bits) of $field_i$ in the structure

$align_i$ is the alignment constraint (in bits) for $field_i$

$align_i$ depends on the type of $field_i$.

$size_i$ is the size (in bits) of $field_i$

$fill_i$ is the amount of internal fill introduced in front of $field_i$

so that it is properly aligned.

- The normal alignment constraint for structure fields is

$$offset_i = address_i \bmod align_i$$

Where $offset_i$ is the offset from an address evenly divisible by $align_i$ to the start of $field_i$. $offset_i = 0$ for scalar data items.

271

Structure Mapping Algorithm

- A simple algorithm sometimes used for Cobol.
Structure is an ordered collection of fields. Substructure is ignored.
 $length_i$ and $align_i$ determined only by $field_i$.

- Algorithm:

$$align = \max_{i=1}^n align_i$$

$$offset = 0$$

$$length = length_1$$

for($i = 2$; $i \leq n$; $i++$) {

$$fill_i = rmod(-length, align_i)$$

$$length = length + fill_i + length_i$$

}

- This algorithm adds just enough fill in front of each field to make the field properly aligned in memory.
- Works for left-to-right equivalence, but not major-minor equivalence.

273

Structure Storage Mapping Algorithms

- A storage mapping algorithm for a structure with n fields computes

$align$ the alignment factor for the entire structure

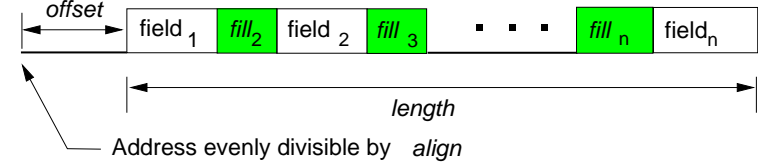
$offset$ the offset for the entire structure.

$fill_i$ the amount of fill in front of $field_i$, $2 \leq i \leq n$

$length$ the total size of the structure.

$$length = length_1 + \sum_{i=2}^n (length_i + fill_i)$$

- Storage mapping memory layout



- The reflexive mod function ($rmod(x, a)$) is defined as:

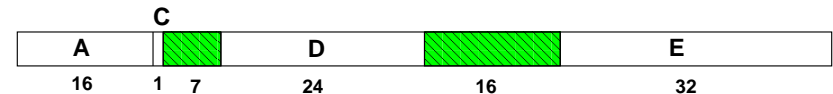
$$rmod(x, a) = \begin{cases} mod(x, a) & \text{if } a > 0, x > 0 \\ 0 & \text{if } mod(|x|, a) = 0 \\ |a| - mod(|x|, |a|) & \text{otherwise} \end{cases}$$

272

Structure Mapping Example - Linear

```
struct {
  char A[ 2 ];      /* item 1 align=8 length=16 */
  struct {
    unsigned C:1 ;  /* item 2 align=1, length=1 */
    char D[ 3 ];    /* item 3 align=8, length=24 */
    int E ;         /* item 4 align=32, length=32 */
  } B ;
} X ;
```

name	i	align	length	offset	align _i	length _i	fill _i
A	1				8	16	
		32	16	0			0
C	2				1	1	
		32	17	0			0
D	3				8	24	
		32	48	0			7
E	4				32	32	
		32	96	0			16



274

Better Structure Mapping Algorithms^a

- Most practical structure mapping algorithms try to satisfy the goals in Slide 270.
- To support Major/Minor equivalence, each embedded record or structure must be mapped separately and independently of its surroundings.
- Structures are mapped *inside/out*, i.e. the most deeply nested sub records or structures are mapped first.
The outermost (main) record or structure is mapped only after all embedded records or structures have been mapped.
- The process also supports Left/Right equivalence.
- The algorithm on slide 273 can be used for this if it is applied separately to each sub record or structure, processing embedded structures depth first.

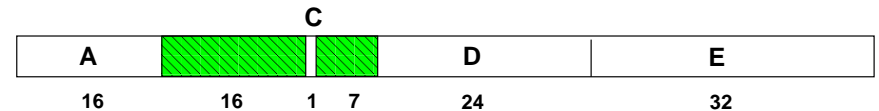
^aThe algorithms presented here do not take advantage of the ability to trade a non-zero offset for a more compact structure layout. A more sophisticated structure layout algorithm is described in: M.D. MacLaren, *Data Matching, Data Alignment and Structure Mapping in PL/I*, ACM Sigplan Notices, Dec. 1970

275

Structure Mapping Example - Depth First

```
struct { char A[ 2 ]; /* item 1,1 align=8 length=16 */
        struct { unsigned C:1; /* item 2,1 align=1, length=1 */
                  char D[ 3 ]; /* item 2,2 align=8, length=24 */
                  int E; /* item 2,3 align=32, length=32 */
        } B; /* item 1,2 */
} X;
```

name	i	align	length	offset	align _i	length _i	fill _i	offset _i
C	2,1	32	1	0	1	1	0	0
D	2,2	32	32	0	8	24	7	0
E	2,3	32	64	0	32	32	0	0
A	1,1	32	16	0	8	16	0	0
B	1,2	32	96	0	32	64	16	0



276

How to Verify Structure Mappings

- Assume that the address given by *start_of_structure* - *structure_offset* is properly aligned according to *structure_alignment*.
In the preceding example, the address of the start of the structure should be evenly divisible by 32.
- Find the *relative location* of each atomic field in the structure by adding up the offset, length of all preceding fields and length of all preceding fill.
- For a correct structure mapping this relative location should be evenly divisible by the *alignment constraint* for the atomic item.

- Examples from the structure in the previous slide

field	relative location	alignment constraint
A	0	0 mod 16 = 0
D	16+16+1+7 = 40	40 mod 8 = 0
E	16+16+1+7+24 = 64	64 mod 32 = 0

277

Unions

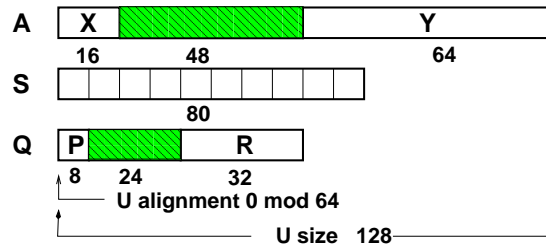
- The algorithm used for records and structures can also be used for unions and similar constructs like Pascal variant records.
- Strategy:
 - Process each alternative as a separate (implicit) record/structure declaration.
 - Alignment of the entire union is the alignment of the most constrained alternative.
 - Length of the union is the length of the longest alternative.
 - It may be necessary to add fill before alternatives at the start of a union to satisfy the alignment constraints for the alternative.
This *might* affect the length of the union.
 - Record separately in the symbol table the offset of each field in each alternative relative to the start of the union.

278

Union Mapping Example

```
union {
    struct {      short X;   double Y;   } A;
    char S[ 10 ];
    struct {      char P;    int R;      } Q;
} U;
```

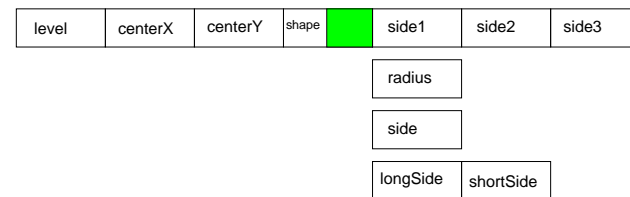
Union Field Mappings



279

Pascal Variant Record Mapping Example

```
type shape = record
    level : integer ;
    centerX , centerY : real ;
    case shape : ( triangle, circle, square, rectangle ) of
        triangle : side1, side2 , side3 : real ;
        circle : radius : real ;
        square : side : real ;
        rectangle : longSide , shortSide : real ;
    end
end
```



280

Other Record/Structure/Union Issues

- Can implementation reorder the fields in a record or structure to minimize internal fill and/or make access more efficient.? .e.g. *Turing*
- Rubber objects (e.g. arrays of dynamic size) in records.
 - One rubber object or many?
 - Can rubber objects be moved to the end?
 - e.g. *variant records* in Pascal
 - Can records containing rubber object be embedded in other types?
 - Can records containing rubber objects be *equivalent* to statically defined records.
- Value of fill and record comparisons.
 - Can entire records be compared?
- Initialization of record variables, initialization in record *type* declarations.
- input and output of records
 - Embedded pointers, non-compact storage of fields

281

Run Time Storage Organization

- The implementation of a programming language must include a plan for how the program and its data will be represented during execution of the program.
- Program Data Issues
 - Allocating storage for variable local to procedures and functions.
 - Addressing of procedure and function local storage.
 - Allocating storage for dynamically allocated variables.
 - Managing storage used for dynamically allocated variables.
- Types of program data
 - Global and static data
 - Procedure and function local data
 - Dynamically allocated data
- Program code representation issues will be discussed later.

282

Addressing of Data

- Most CISC computers provide several different *addressing modes*, some possibilities are listed below

Absolute	main memory address
Register	memory address in hardware register
Register + displacement	Effective address is sum of memory address in register and a constant displacement.
Indexed	Contents of index register is added to register + displacement address
- RISC computers usually provide only one or two addressing modes, typically Register or Register+displacement
- The language implementor must use the addressing modes on the target machine to provide access to the program and its data.
- Register + displacement, perhaps with indexing is the most common form of addressing.

283

Dynamically Allocated Storage

- Many languages allow the programmer to dynamically allocate storage for arbitrary data structures. e.g. **new** in C++.
- The compiler typically turns programming language storage allocation constructs into calls on a compiler generated internal function that actually manages the storage.
- Possible storage allocation strategies:
 - Let the underlying OS do all the storage management.
 - Allocate a large block of storage at program initialization, and suballocate parts of it to the program on request. One typical approach is to allocate a large block of memory and grow the stack of activation records from one end and dynamically allocate memory from the other end.
- Depending on the programming language, the programmer may be responsible for deallocating memory (C, C++) or deallocation may be done automatically (Java). Automatic storage deallocation is hard^a because it requires the ability to find *all* live pointers to dynamically allocated storage.

^aSee R. Jones and R. Lins, *Garbage Collection - Algorithms for Dynamic Memory Management*, J. Wiley, 1997 for a very thorough discussion of dynamically allocated storage.

285

Global and Static Storage

- Storage for global (main program) data is allocated once at the start of program execution.
- Some languages allow (e.g. C, P/I) or require (e.g. Fortran) data to be given the *static* storage attribute.
Storage for static data is also allocated once at the start of program execution.
- Compiler generated tables on constants and initial values for variables may also be treated as static data.
- If the available addressing modes allow it (i.e. maximum allowable displacement is large enough) one hardware register might be used to provide addressability for global data.

284

Local Storage

- The variables declared in a procedure or function are the *local storage* of the procedure or function.
- In most modern languages, local storage is allocated when the procedure or function is called and deallocated when the procedure or function returns.
- In programming languages that allow recursion, there may be many instances of the local storage for a procedure or function allocated at a given point in the programs execution.
Each instance is associated with a particular call of the procedure or function.
- In almost all languages, storage allocation for local variables follows a *stack-like* (LIFO) discipline, i.e. the most recently allocated storage is the first to be deallocated.
- Therefore, conceptually at least, a stack is used to manage the allocation and deallocation of local storage.

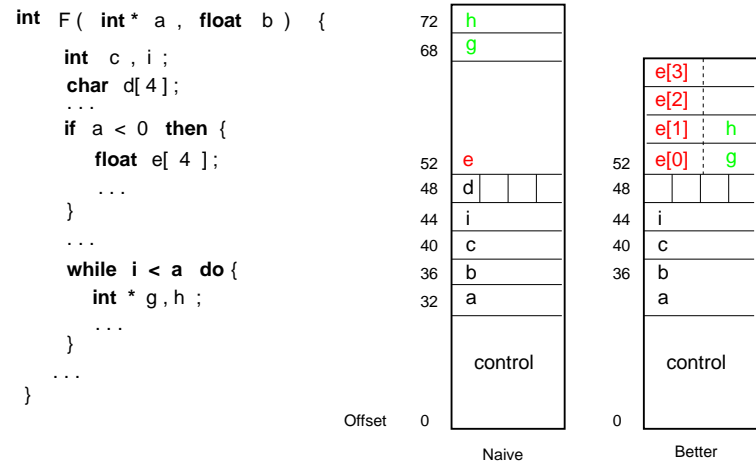
286

Local Storage - Activation Records

- The term *activation record* refers to the local storage that is allocated when a function or procedure is called.
- A typical activation record contains
 - The return address for the call
 - Control information required to maintain addressing of local storage.
 - Actual parameters passed to this particular call of the procedure or function
 - Storage for the local variables declared in the procedure or function.
 - Pointers to storage for variables that were dynamically allocated, .e.g arrays with dynamic bounds.
 - Storage for the dynamic arrays may also be in the activation record.
 - Temporary storage required for expression evaluation.
 - Storage for variables from *micro scopes* contained in the procedure or function.
- Typically one (or more) hardware registers are used to address the activation record for each procedure or function.

287

Activation Record Example



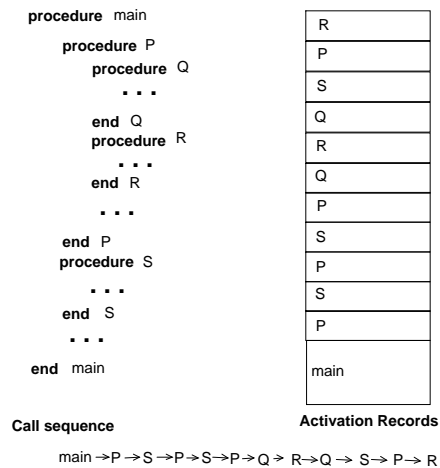
288

Addressing Activation Records

- During the execution of a program each procedure needs to be able to access
 - Its own local data *corresponding to its invocation*.
 - The local data of any containing procedures.
 - Global data in the main program.
 - Any static data that it is allowed to access according to the scope and visibility rules of the language.
- Due to recursion and deeply nested procedure calls it is not practical (on most hardware) to permanently allocate a hardware register for every procedure and function.
- The number of hardware registers that is needed to provide complete addressability is determined by the *depth of static nesting* of procedures in the program.
- An addressing mechanism called a **display** is used to describe the addressing for local storage of procedures and functions.

289

Activation Record Example



Fischer/Leblanc Figure 9.5

290

Lexical Levels

- Define: Redlexical level is the depth of static nesting of scopes (usually major scopes like procedures and functions) in a program.
- In normal procedural languages, at any point during the execution of a program at most one activation record at each lexical level is visible.
- For the example in Slide 290

Name	Level	Visible Scopes
main	0	main
P	1	main, P
Q	2	main, P, Q
R	2	main, P, R
S	1	main, S

291

Displays

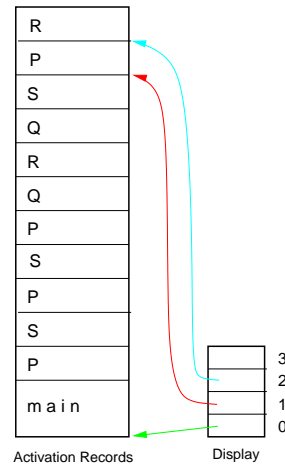
- Define: A **display** is a vector of addresses of activation records. The display is indexed by lexical level.
- Define: An **Order Number** is a non-negative integer that specifies the order in which objects are defined in a scope.
- If all objects are of unit size then the order number is the items relative offset in the scopes activation record.
In most modern languages all objects are not of unit size and the order number is translated into a relative byte offset into the activation record.
- At any point during a programs execution, one display entry for each active lexical level is sufficient to provide addressing to all visible activation records.

292

Activation Records and Display

```

procedure main
  procedure P
    procedure Q
      . . .
    end Q
    procedure R
      . . .
    end R
    . . .
  end P
  procedure S
    . . .
  end S
  . . .
end main
  
```



Call sequence:

main → P → S → P → S → P → Q → R → Q → S → P → R

Fischer & LeBlanc Figure 9.5

293

Display Update

- To provide correct addressing during program execution, the display must be updated every time that the list of visible scopes changes.
- Micro scopes are usually subsumed in the nearest enclosing major scope (procedure or function) so that the display only needs to be updated on entry to or exit from a procedure or function.
- Types of procedure/function calls

same level	caller and callee have the same lexical level	Q calls R
up level	lexical level of callee greater than level of caller	P calls Q
down level	lexical level of caller greater than lexical level of callee	Q calls S
- The different types of calls require different amounts of the display to be saved and restored. Defer parametric calls to Slide 306.
- A algorithm for updating the display should be efficient and *correct* for all possible sequences of calls.

294

Simple Display Update

- Define: A **dynamic link** is a pointer from an activation record to the previous activation record on the stack of activation records.
The dynamic link is useful in deallocating an activation record on return from a procedure or function.
- Define: The **block mark** is the control information that occurs at the start of every activation record. The block mark typically contains the return address for the invocation of the procedure or function, links for maintaining the display
- Simple block mark

```
struct blockMark {
    void * returnAddress; /* return address */
    void * dynamicLink; /* dynamic link */
    short lexicLevel; /* lexical level of this scope */
};
```

295

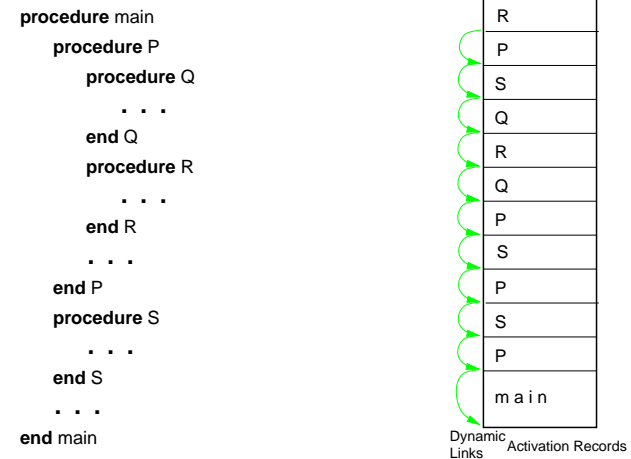
Simple Display Update Algorithm

```
P = address of topmost block mark ;
L, LL = P -> lexicLevel ;
display[ * ] = empty ;
while( L != 0 ) {
    if( L <= LL && display[ L ] == empty )
        display[ L ] = P ;
    P = P -> dynamicLink ;
    L = P -> lexicLevel ;
}
```

- Intuitively: chain down dynamic links and put into display the first entry encountered at each lexic level.
- Simple algorithm is very inefficient if the chain of dynamic links becomes long (e.g. deep recursion).

297

Activation Records and Dynamic Links



Call sequence:

main → P → S → P → S → P → Q → R → Q → S → P → R

Fischer & LeBlanc Figure 9.5

296

Better Display Update

- Define: A **static link** is a pointer from an activation record to the activation record of its statically containing scope (parent).
- On procedure call to a procedure/function at level L, the static link field in the block mark is the value of display[L - 1]
- Better block mark

```
struct blockMark {
    void * returnAddress; /* return address */
    void * dynamicLink; /* dynamic link */
    void * staticLink; /* static link */
    short lexicLevel; /* lexical level of this scope */
};
```

- Retain dynamic link for resetting the stack pointer on return.

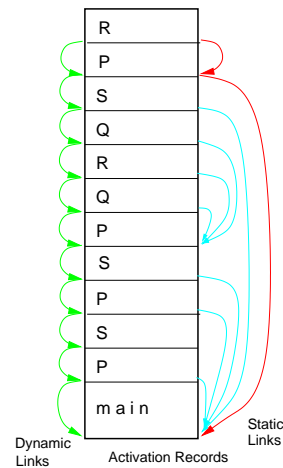
298

Activation Records and Static/Dynamic Links

```

procedure main
  procedure P
    procedure Q
      . . .
    end Q
    procedure R
      . . .
    end R
    . . .
  end P
  procedure S
    . . .
  end S
  . . .
end main

```



Call sequence:

main → P → S → P → S → P → Q → R → Q → S → P → R

Fischer & LeBlanc Figure 9.5

299

Better Display Update Algorithm

```

P = address of topmost block mark ;
display[ * ] = empty ;      /* Neatness only */
L = P -> lexicalLevel ;
while( L != 0 ) {
  display[ L ] = P ;
  P = P -> staticLink ;
  L = P -> lexicalLevel ;
}

```

- Intuitively: chain down static links copying entries into display. Order is child, parent, grandparent, greatgrandparent .. ultimate ancestor.
- Cost depends only on depth of lexical nesting.

300

Constant Cost Display Update

- The cost of the display update algorithm in Slide 300 varies with the difference of the lexical levels of the calling and called procedure. An alternative is the *constant cost* display update algorithm.
- Constant Cost Display Update
 - On a call from level N to level M
 - Save display[M] in the local storage of the calling procedure.
 - Called procedure sets display[M] to point to its activation record.
 - On return, restore the saved value of display[M].
- Intuitively, display[M] will be modified, so save it first. If this algorithm is applied consistently, display will always be correct.
- Advantages: display update code is small, fast and easy to generate. Downlevel calls are cheap. Can extend for parametric procedures.
- Disadvantage: logically unnecessary save/restore on uplevel calls.

301

Parametric Procedures

- Many languages (C, Pascal, Fortran, PL/I) allow the *name* of a function or procedure to be passed as an actual parameter to some other function or procedure. Some languages allow procedure and function *variables* or pointers to functions.
- These constructs cause problems for display update algorithms because they allow a procedure or function to be executed far from the environment of their parent scopes.
- Intuitively, the correct display for a procedure or function passed as a parameter is the display as it existed at the point where it was originally passed as a parameter.
- Passing the entire display around with the parametric procedure or function is possible, but there are better solutions.
- The next two slides illustrate the problems caused by parametric procedures.

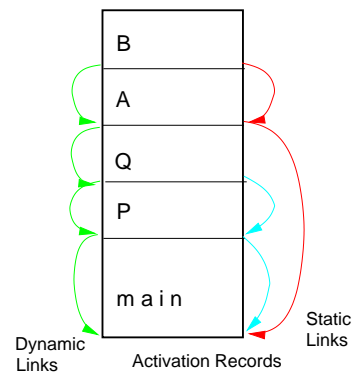
302

Parametric Procedure Example

```

procedure main
  procedure P
    procedure Q
      procedure R
        . . .
      end R
      A( R )
    end Q
    Q
  end P
  procedure A( F )
    procedure B( G )
      G
    end B
    B( F )
  end A
  P
end main

```



Call sequence:
main → P → Q → A → B

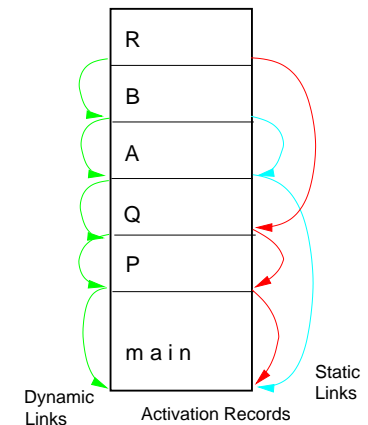
303

Parametric Procedure Example – continued

```

procedure main
  procedure P
    procedure Q
      procedure R
        . . .
      end R
      A( R )
    end Q
    Q
  end P
  procedure A( F )
    procedure B( G )
      G
    end B
    B( F )
  end A
  P
end main

```



Call sequence:
main → P → Q → A → B → R

304

Parametric Procedure Issues

- The presence of parametric procedures implies that the compiler cannot determine the to/from lexic level of calls.
- Parameter counting and/or parameter type checking may have to be done at run time.
- *Name* of parent procedure is not adequate to identify parent since multiple instances of the parents activation record may exist on the stack when the parametric procedure is called.
- Therefore , the internal representation of a parametric procedure must include
 - The code address for the entry point of the procedure.
 - some form of information that will allow the correct addressing environment for the parametric procedure to be established when it is called.

305

Parametric Procedures - Solutions

- There are several possible solutions for providing a correct addressing environment for parametric procedures.
The solutions differ in how parametric procedures are represented internally.
1. Save entire display when parametric procedure is passed and pass it along as part of the parametric proc.
Trivial to set up display, makes parametric procedures large objects.
 2. Save entire display in local storage of caller and pass address to save display as part of parametric procedure.
 3. Represent parametric procedure as address of parents local storage, and offset in this local storage of code address for parametric procedure.
The address of parents local store is the correct static link for the parametric procedure.

306

Dynamically Created Variables

- Languages like Icon, Snobol^a, Prolog allow variables to be created dynamically during program execution.
- Example: in Icon the programmer can concatenate some strings together to create the *name* of a new variable.
- At the point where this string is used for the first time as a variable, name, the variable is dynamically created by the RTE and initialized to a null value. The variable remains in existence until it is explicitly destroyed by the programmer.
- Most languages that have this kind of feature are also typeless (or weakly typed). Usually the dynamically created variables have global scope.
- Programmers often use dynamically created variables as *associative arrays*

^aSee: R.E. Griswold, The Macro Implementation of Snobol4, W.H. Freeman Co, 1972 for a much longer discussion of how to implement dynamically created variables.

Dynamic Memory Management

- Management of dynamically created runtime storage is a central issue in the implementation of many programming languages.
- Most implementations allocate storage in a *heap* with some mechanism for reclaiming memory that is no longer in use.
- **Garbage collection^a** is a tool for automatically managing dynamically allocated memory.
 - Identify dynamically allocated memory that is no longer in use.
 - Recycle that memory for later reuse
- Some programming language require Garbage Collection (Java, Lisp), many others allow it.
- The presence of Garbage Collection adds runtime overhead and may introduce unpredictable *stalls* to program execution

^aRichard Jones, Antony Hoskin, Elliot Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Chapman & Hall, 2011

Dynamically Created Variables Implementation

- The RTE maintains a runtime *hashtable* that stores all variables in the program as (*name* , *value*) pairs. The variable name is the hash key.
- The code generated for every reference to a variable is a call to an RTE hash table lookup routine.
 - If the name is found as an existing variable, its address is returned.
 - If the name is not found, then a new empty variable is created and its address is returned.
- The compiler may optimize access to variables that are known at compile time. Usually the places where dynamically created variables may be used can be detected syntactically.
- Another useful optimization is to detect *probing*, i.e. creating variable names and then testing if the variable has a non-null value (e.g associative array search). This programming technique can overload the hash table with null entries if it is not optimized.

Dynamic Storage Design Issues

- Roots and pointer finding.
Is it possible to find all pointers (roots) to allocated objects?
This is really hard to get correct.
- Mobility of data. It is only possible to move a data object if it is possible to (eventually) update *all pointers* to the object to point at the new location.
- Immediacy of storage reclamation. Is recyclable memory made available immediately or only periodically.
- Program tolerance to interruption and/or arbitrary delay.
Can mutation and collection proceed concurrently?
- Processing Cost – Time and Space Overhead
- Fragmentation of heap memory.
Performance Degradation as heap utilization increases.
- Recovering cyclic data structures.

Basic Principle

- The key concept for garbage collection is **reachability**. A program can only access some object if it has a memory address (i.e. pointer) for the object. An object that has been allocated but is no longer reachable, is *garbage* and may be recycled.
- *Roots* are base locations that may hold the address of dynamically allocated objects. Possible roots include: processor registers, all kinds of variables and temporaries in activation records.
- Dynamically allocated memory can only be accessed through some chain of pointers starting with some root. An object is *reachable* if and only if there is some path from a root to the object.
- Two separate issues:
 - identifying reachable objects
 - managing and recycling memory efficiently.

311

Compacting and Non-compacting Collectors

- Once recyclable memory has been identified, there are two main options for managing this memory.
- Compacting collectors move all live objects into a block of contiguous memory. This leaves a large block of memory available for reuse.
- Non-compacting collectors leave the live objects in place, and (somehow) link all the recycled memory for later use.
- Compacting advantages
 - Little effort required to reclaim objects.
 - New allocations are easy with a large block of contiguous free memory.
 - Moving object together may improve cache performance.
- Compacting Disadvantages
 - All pointer access must be managed through the collector.
 - Moving large blocks of memory may impact memory performance.
 - May require twice as much memory for compacting.

313

Classic Dynamic Memory Design Approaches

- *Reference Counting* - Every allocated object contains a *reference count*. This count is incremented when a new pointer to the object is created and decremented when a pointer to the object is destroyed. A reference count of zero implies the object has no pointers pointing at it and so it can be reclaimed. Incremental algorithm.
- *Mark and Sweep Collection* - A garbage collector algorithm starts at all roots recursively marking all nodes that can be reached. When this algorithm finishes, any nodes that are not marked can be reclaimed.

Conservative marking, takes more space, but use a simpler algorithm to identify pointers. Strategy:

- Sweep all available memory, test each word for the *isPointer* property
- Unless the collector can (heuristically) decide that a word does not contain a pointer, assume that it is a pointer.
- If the collector knows the range of valid memory addresses, it can efficiently exclude, small integers, real numbers and strings

312

Reference Counting

- Used in many languages and applications:
Smalltalk, InterLisp, Java, Adobe PhotoShop
- Storage management runs concurrently intermixed with program execution.
- Invariant: the reference count of each node is equal to the number of pointers pointing to the node.
- Can do optimizations on pointers local to a routine to reduce the amount of reference counting.

314

Reference Counting - Advantages

- Memory management overhead is distributed throughout the computation. May yield smoother response time.
- Locality of reference no worse than application.
- Data from real languages suggests that few nodes are shared and many nodes have a short lifetime. Reference counting lets these nodes be reused as soon as they are free.
This may lead to better cache/virtual memory behavior.
This may allow *update in place* optimizations.
- Performance relatively independent of heap occupancy fraction.

315

Reference Counting - Disadvantages

- High processing cost. Every pointer assignment requires reference count updating.
- Correctness is difficult.
Missing even one reference counter update could lead to errors.
- Difficult to deal with embedded pointer assignments, e.g. bulk assignment of a structure containing pointers.
- Reference count roughly doubles the size of every pointer.
- Reference counting schemes can not reclaim cyclical data structures, e.g. a doubly linked list.
- Need some strategy to deal with reference counter overflow.

316

Naive Mark Sweep Algorithm^a

- Basic algorithm:
 - Suspend program processing.
 - Start at all roots. The algorithm must be able to find **all** roots, e.g. even roots in registers.
 - Trace nodes reachable from the roots and mark them.
 - If trace reaches a node that is already marked, stop tracing on that path.
 - After marking is complete, make another sweep through the heap unmarked nodes can be reclaimed.
reset mark on marked nodes for next sweep.
 - Restart program execution.
- Mark and Sweep is usually triggered asynchronously when the program makes a request for memory that cannot be satisfied.

^aUsed in original Lisp implementation.

317

Mark and Sweep Advantages

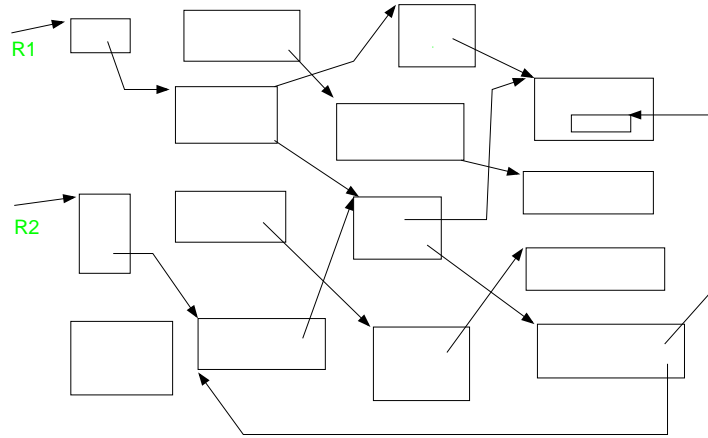
- Cycles handled easily.
- No overhead on pointer manipulation.
- No space overhead on pointers.
- Handles bulk assignment correctly.

Mark and Sweep - Disadvantages

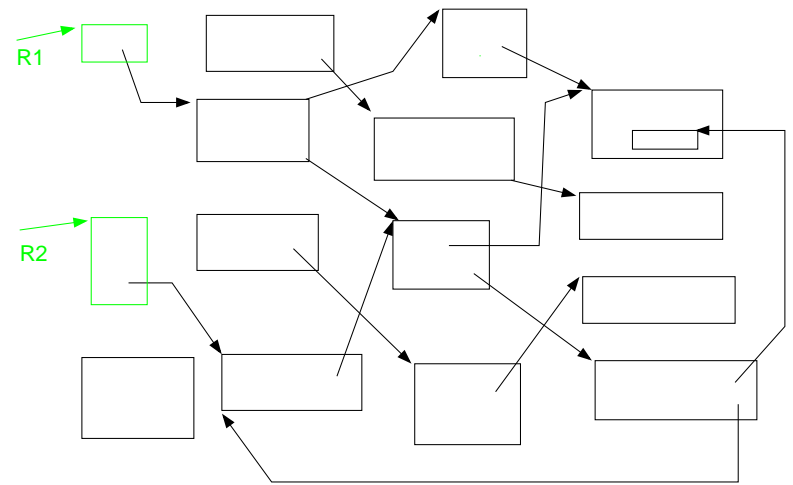
- Stop - Start algorithm.
Unsuitable for interactive applications.
- Guaranteeing that all roots can be found is *hard*
- Need to disable collector in critical sections.
- Cost of mark/sweep passes is high especially in modern memory hierarchies.
- Time is proportional to size of heap, not number of active cells.
Algorithm will tend to *thrash* as heap occupancy increases.
- Without optimizations leads to highly fragmented memory.

318

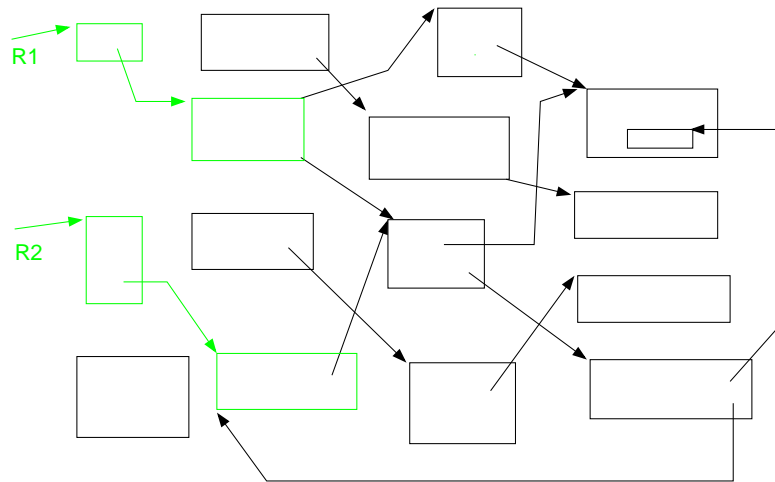
Naive Mark and Sweep Example



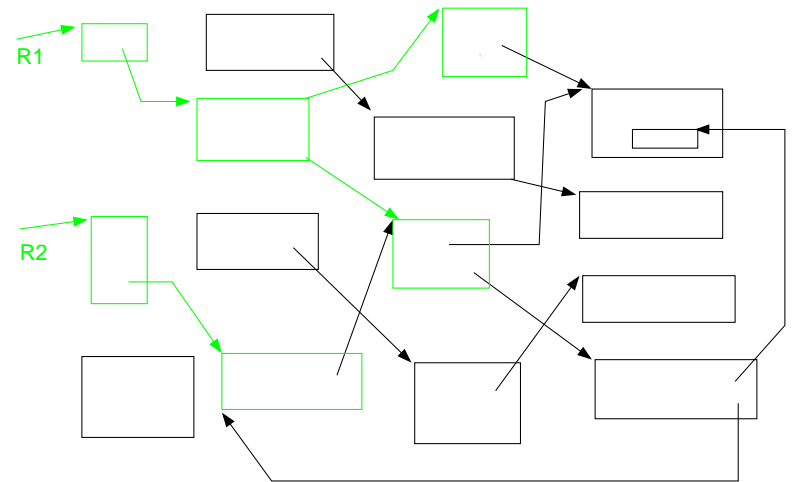
319



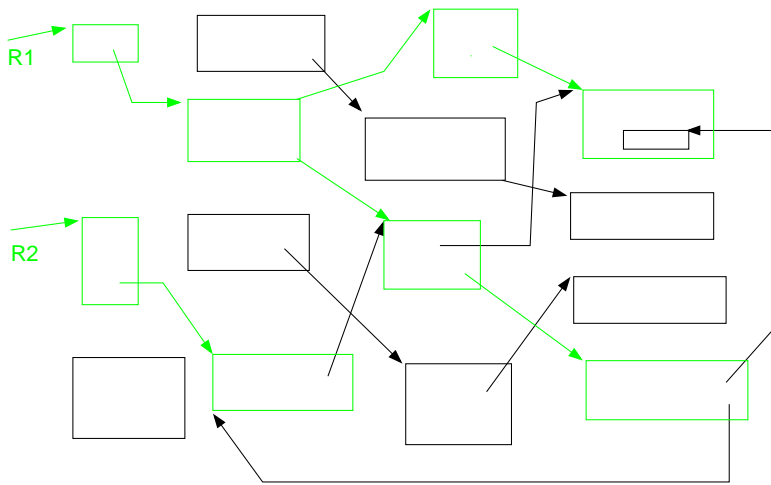
320



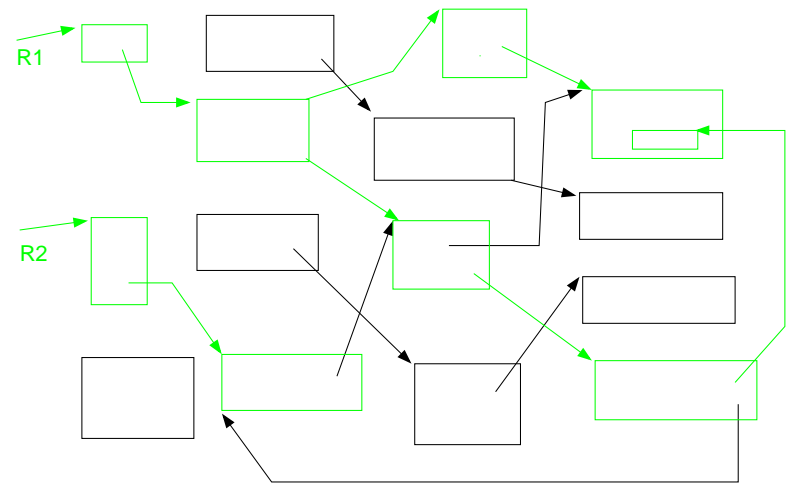
321



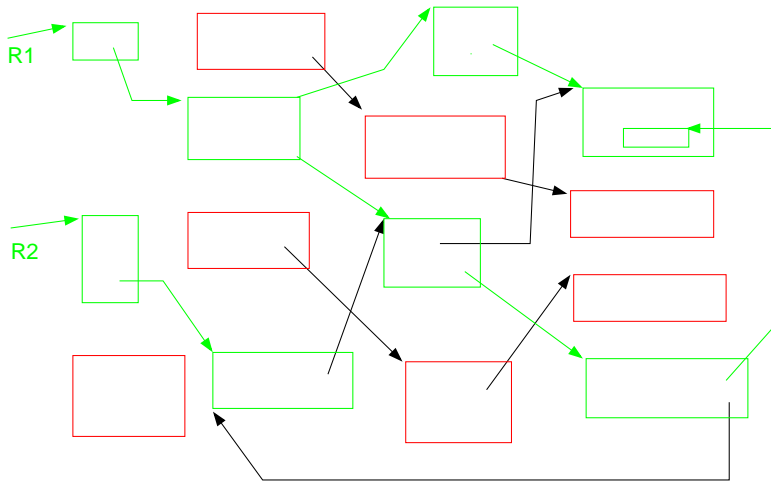
322



323



324



325

Tri-color Marking

- A modern version of Mark/Sweep. This is an *on the fly* algorithm. Allocation and mutation can continue during collection.
- Start with three sets:
 - **White set** contains objects that can have their memory recycled.
 - **Black set** contains objects that have no pointers to objects in the white set but are not available for recycling.
 - **Gray set** contains all objects reachable from root references. Objects referenced from the grey set haven't been scanned yet. Grey set objects will eventually move to the Black set.
- Initially
 - The Black set is empty
 - The Grey set is initialized to all objects that can be reached *directly* from roots
 - The White set contains all objects not in the grey set.

326

- Algorithm
 - Objects only move from white to grey to black.
 - Pick an object (*victim*) from the Grey Set.
Move all objects in the White set that *victim* references *directly* to the Grey set.
Move object to the Black Set.
 - Iterate until the Grey set is empty.
Then all objects in the White set can be recycled.
- **Invariant** No object in the Black Set points directly to an Object in the White set.
Any allocation or mutation during collection must maintain this invariant.

327

Compacting Collectors

- Widely used due to superior performance over reference counting and mark/sweep.
- Uses two large regions of memory *fromSpace* and *toSpace*.
One region is active, containing data objects and new allocations
- Compacting collector algorithm
 - Suspend program execution.
 - Flip identity of two semi-spaces. $toSpace \leftrightarrow fromSpace$
 - Trace from roots all live data structures in *fromSpace*.
 - Copy live nodes to *toSpace* when the node is first visited.
 - Update all pointers on the fly to point to new location of node.
Assume *fromSpace* and *toSpace* pointers can be distinguished.
Requires mapping table of old/new pointer values.
 - *fromSpace* becomes inactive until the next collection
 - Restart program execution.

328

- Compacting Collection - Advantages
 - Good time performance. Efficient allocation.
 - Asymptotic complexity less than marks/sweep collectors.
 - Fragmentation is eliminated since live data is copied compactly into the *toSpace*.
 - Can handle bulk pointer assignment.
- Compacting Collection - Disadvantages
 - Must be able to identify and update *all* pointers to data objects being moved.
 - May degrade paging in virtual memory systems.
Collector may get more page faults than other techniques.
 - Requires potentially large mapping table during copying to update pointers.
 - Stop - start algorithm. Not suitable for interactive or real-time systems.
 - Requires double the address space of other techniques.

329