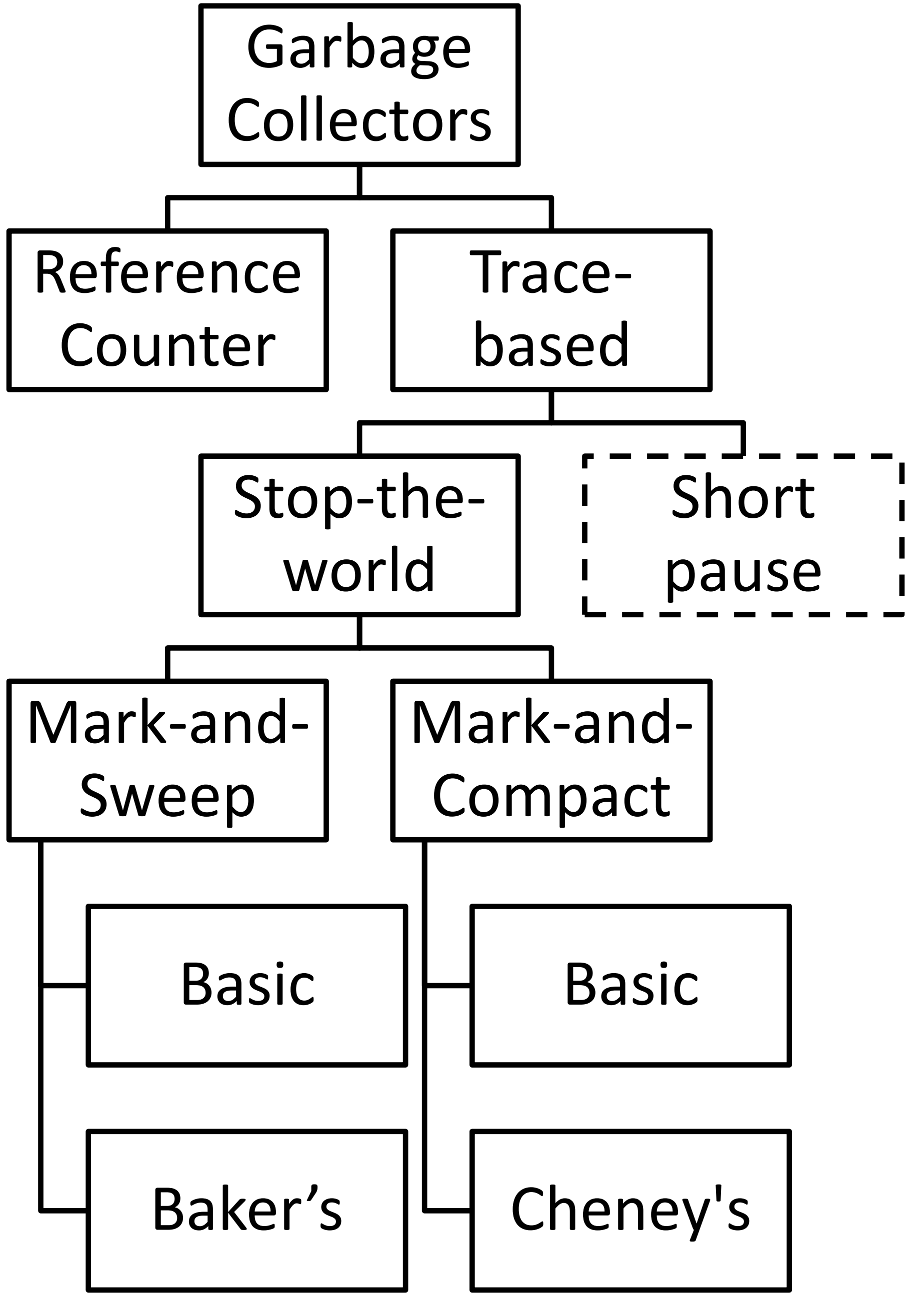
CC Lecture 16

Prepared for: 7th Sem, CE, DDU
Prepared by: Niyati J. Buch



http://infolab.stanford.edu/~ullman/dragon/w06/lectures/gc.pdf

Relocating Collectors

- **Relocating collectors** move reachable objects around in the heap to eliminate memory fragmentation.
- It is common that the space occupied by reachable objects is much smaller than the freed space.
- Instead of freeing the holes individually, relocate all the reachable objects into one end of the heap, leaving the entire rest of the heap as one free chunk.
- As GC already analyzed every reference within the reachable objects
- So this and references in root set is required to be changed.

Advantages

- Having all the reachable objects in contiguous locations reduces fragmentation of the memory space.
- Also, by making the data occupy fewer cache lines and pages, relocation improves a program‘s temporal and spatial locality, since new objects created at about the same time are allocated nearby chunks.
- Objects in nearby chunks can benefit from prefetching if they are used together.
- Further, the data structure for maintaining free space is simplified; instead of a free list, all we need is a pointer free to the beginning of the one free block.

Types of Relocating Collectors

- **Relocating collectors** vary in whether they relocate in place or reserve space ahead of time for the relocation:

1. A **Mark-and-Compact collector**, described in this section, compacts objects in place.
   - Relocating in place reduces memory usage.

2. The more efficient and popular **Copying Collector** moves objects from one region of memory to another.
   - Reserving extra space for relocation allows reachable objects to be moved as they are discovered.

3 phases of Mark-and-Compact collector

1. First is a **marking** phase, similar to that of the mark-and-sweep algorithms described previously.
2. Second, the algorithm **scans** the allocated section of the heap and **computes a new address** for each of the reachable objects.
   - New addresses are assigned from the low end of the heap, so there are no holes between reachable objects.
   - The new address for each object is recorded in a structure called NewLocation.
3. Finally, the algorithm **copies** objects to their new locations, updating all references in the objects to point to the corresponding new locations.
   - The needed addresses are found in NewLocation.

Phase 1: Mark-and-Compact collector

```
/* mark */
Unscanned = set of objects referenced by the root set;
while (Unscanned ≠ ∅) {
        remove object o from Unscanned;
        for (each object o′ referenced in o) {
                if (o′ is unreached) {
                        mark o′ as reached;
                        put o′ on list Unscanned;
                }
        }
}
```

This is just like mark phase in mark and sweep algorithm.

Phase 2: Mark-and-Compact collector

```
/* compute new locations */
free = starting location of heap storage;
for (each chunk of memory o in the heap, from the low end) {
        if (o is reached) {
                NewLocation(o) = free;
                free = free + sizeof(o);
        }
}
```

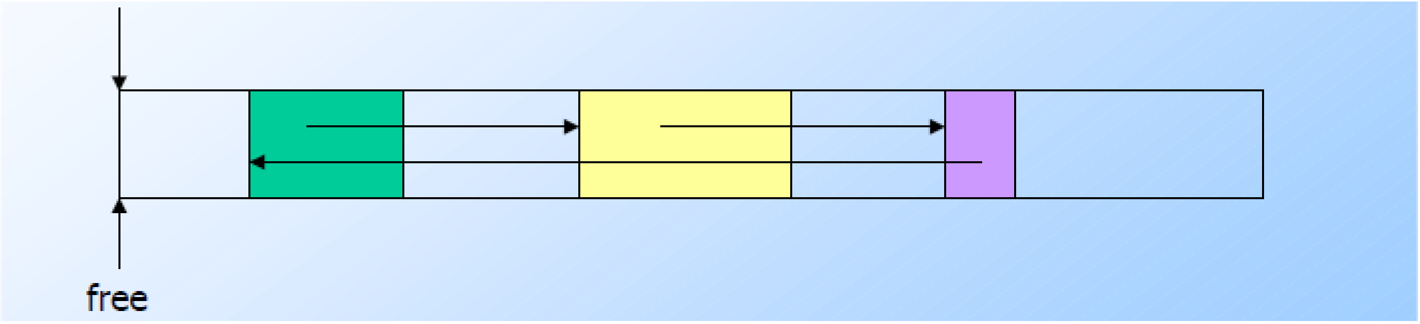
•Maintain a table (hash?) from the reached chunks to new locations for the objects in those chunks.
•Scan chunks from the low end of the heap.
•Maintain the pointer free that counts how much space is used by the reached objects so far.

Phase 3: Mark-and-Compact collector

```
/* retarget references and move reached objects */
for (each chunk of memory o in the heap, from the low end) {
    if (o is reached) {
        for (each reference o.r in o)
            o.r = NewLocation(o.r);
        copy o to NewLocation(o);
    }
}
for (each reference r in the root set)
    r = NewLocation(r);
```

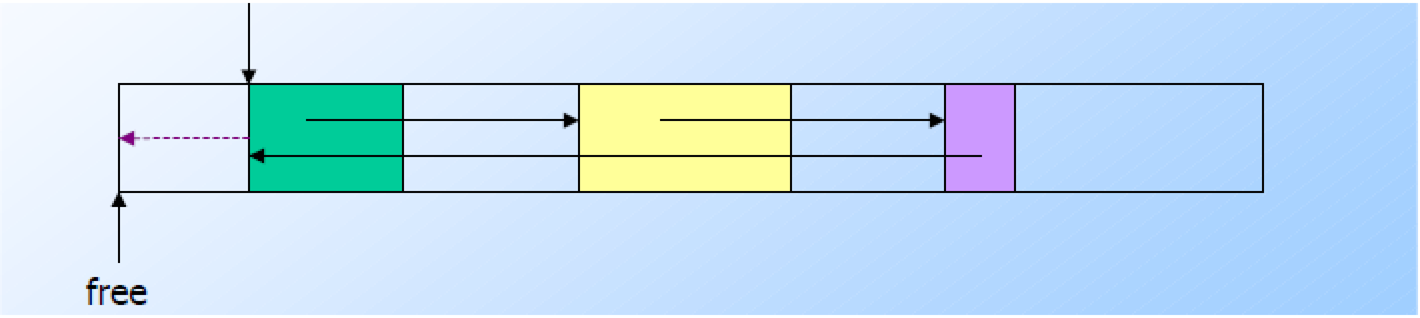
- Move all the reached objects to their new locations, and also retarget all the references in those objects to the new locations.
- Use the table of the new locations.
- Retarget the root references.

Example- Initial State of heap

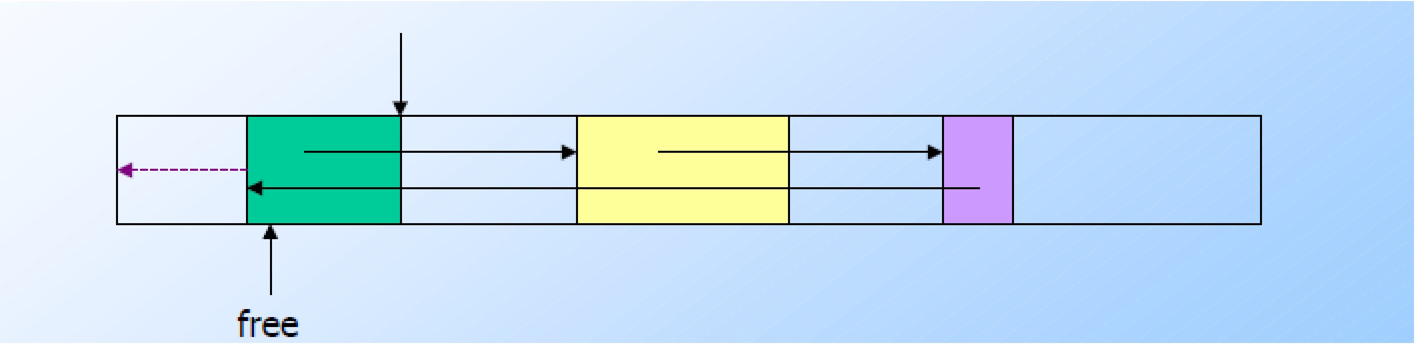


http://infolab.stanford.edu/~ullman/dragon/w06/lectures/gc.pdf

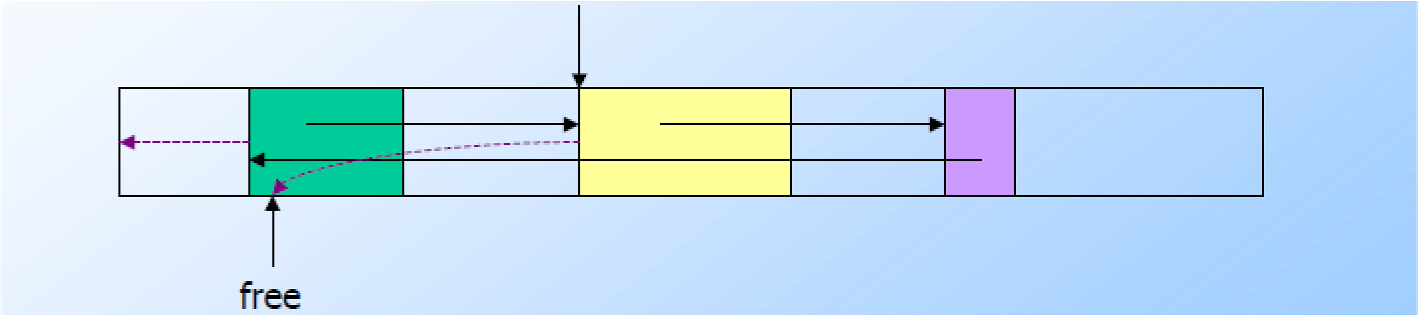
Example



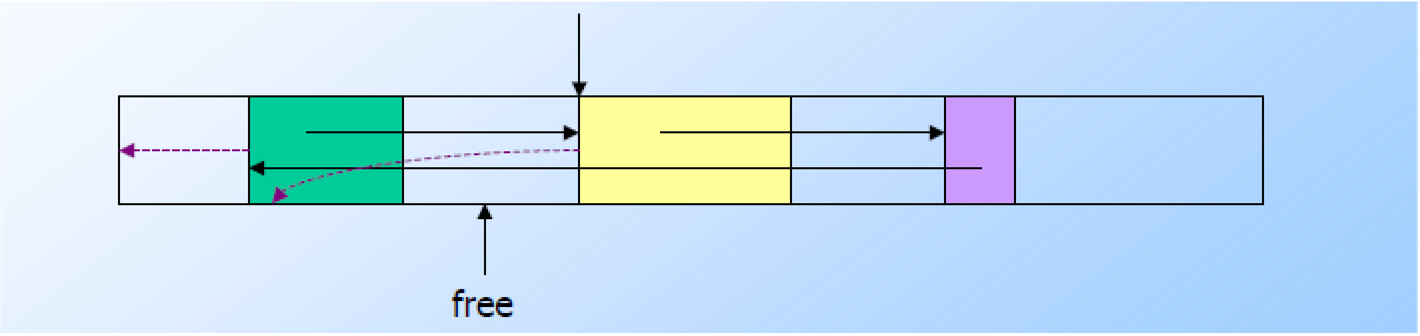
Example



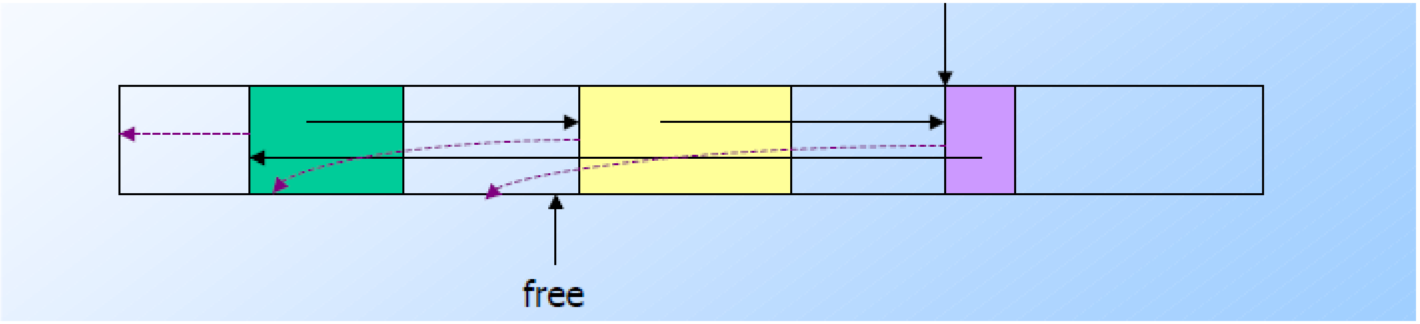
Example



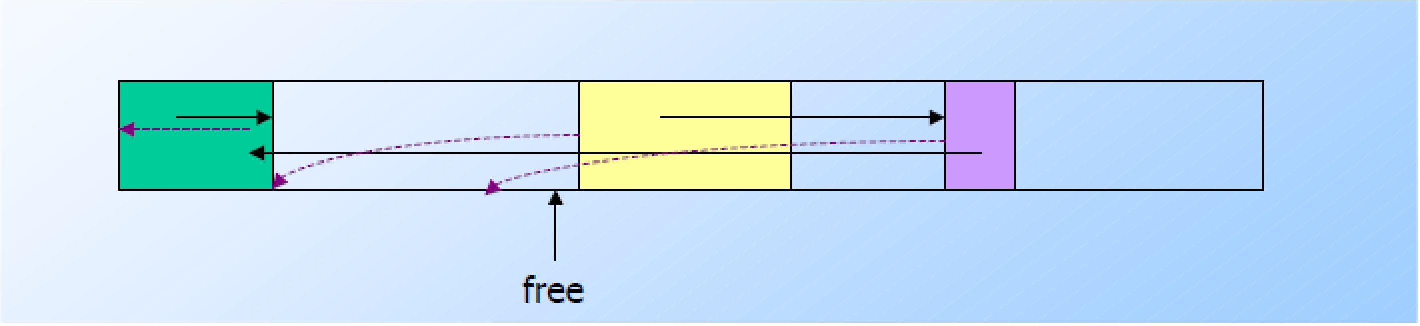
Example



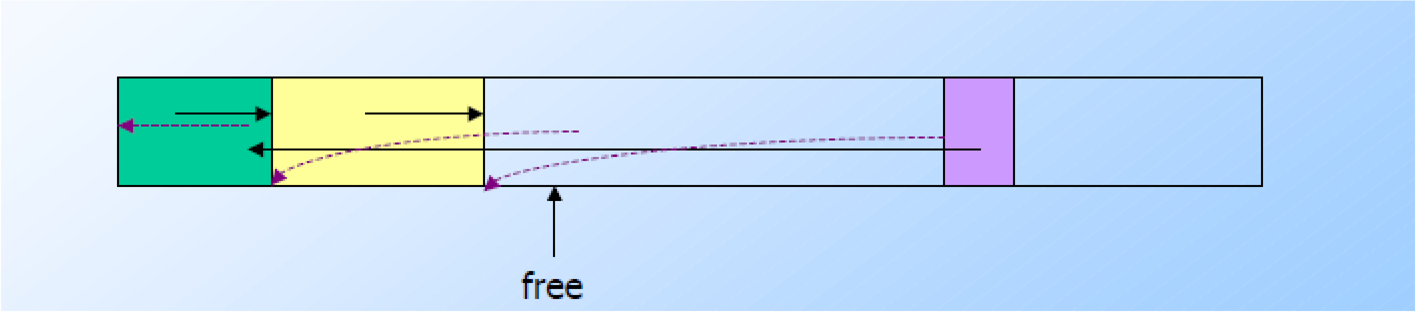
Example



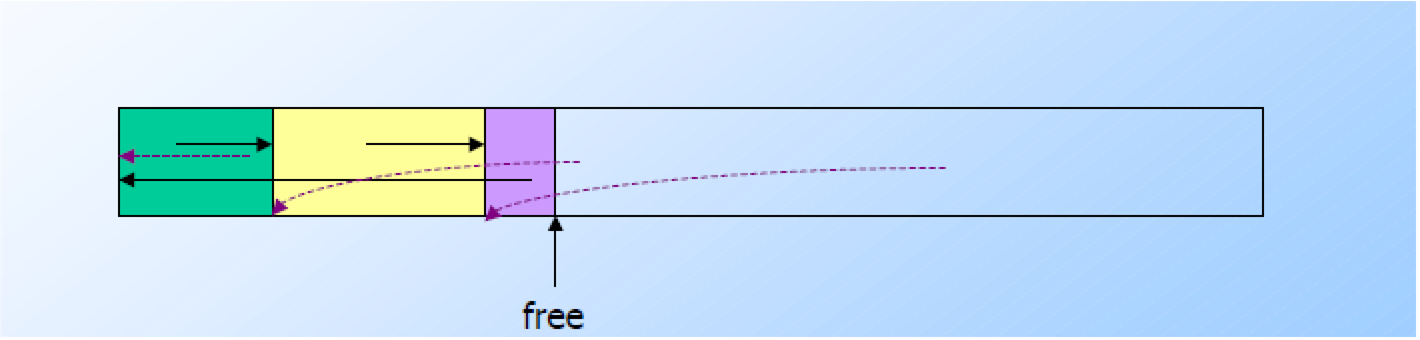
Example



Example



Example



Comparison

- **Basic Mark-and-Sweep**:
  - Cost is proportional to the number of chunks in the heap
- **Baker's Mark-and-Sweep**:
  - Cost is proportional to the number of reached objects
- **Basic Mark-and-Compact**:
  - Cost is proportional to the number of chunks in the heap plus the total size of the reached objects

- Cheney's Copying collector:
  - Cost proportional to the total size of the reached objects (it does not touch any of the unreachable object)

Symbol Table

*Ref: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

Introduction to Symbol Table

- **Symbol table** is an essential data structure used by compilers to remember information about identifiers in the source program.
  - Identifiers like variables, procedures, functions, constants, labels, structures, etc.

- Good representation of symbol table → fast access of symbol table → fast compiler

- Symbol table must have efficient representation of scoped variables (global & local)

Introduction to Symbol Table

- Usually lexical analyzer and parser fill up the entries in the table.
  - Whenever the lexical analyser comes with a new token, if it finds it is an identifier, it installs it in the symbol table and returns the index of that symbol table as an attribute to the token or ID.
  - Parser added information like type of identifier.

- E.g. int x, y, z;
  - Tokens which identifiers like x y z are added to the symbol table by lexical analyzer
  - Its type INT will be added by parser

Introduction to Symbol Table

- Code generator and optimizer makes use of this information stored in the Symbol table.

- The symbol tables may vary from implementation to implementation even for the same language.

Information in Symbol Table

- **Name**
  - Name of the identifier
  - may be stored directly or as a pointer to another character string in an associated string table.

- **Type**
  - Type of the identifier like variable, label, procedure name, etc
  - For variables
    - <u>Basic</u> type like integer, real, float, etc
    - <u>Derived</u> type

Information in Symbol Table

- **Location**
  - offset within the program where the identifier is defined.
- **Scope**
  - Region of the program where the current definition is valid
- **Other attributes**
  - Array limits, parameters, return values, etc

Usage of Symbol Table information

- **Semantic Analysis**
  - Check correct semantic usage of language constructs
  - E.g. types of identifiers
    - x = y Are the types compatible for assignment?
    - a + b Are the types compatible for + operator?

- **Code Generation**
  - Types of variables provide their sizes
  - As individual variable needs to be allocated space during code generation

Usage of Symbol Table information

- **Error Detection**
  - Undefined variables
  - Recurrence of error messages can be avoided by marking the variable types as undefined
  - E.g. if a variable **x** is used in the program and its type is not defined then this will give error <u>everywhere</u> **x** is used.
  - Solution can be at first place where it found that x is undefined, add the type in symbol table as <u>undefined</u>

- **Optimization**
  - Two or more temporaries can be merged if their types are same and they need not be given space simultaneously .

Operations on the Symbol Table

- **Lookup**
  - Most frequent operation
  - Whenever an identifier is seen, it is needed to **check** its types, or **create** a new entry

- **Insert**
  - Second important operation
  - **Adding** new names to the table, happens mostly in lexical and syntax analysis phases

Operations on the Symbol Table

- **Modify**
  - When a name is defined, all the information may not be available.
  - The information may be **updated** later.

- **Delete**
  - Not very frequent
  - will occur particularly when any procedure ends .

Issues for Symbol Table design

- **Format of the entries**
  - Various formats like linear array, tree structure, table, etc
- **Access methodology**
  - Linear search, Binary search, Tree search, Hashing, etc.
- **Location of storage**
  - Primary memory, partial storage in secondary memory
- **Scope Issues**
  - In block-structured language, a variable defined in upper blocks must be visible to inner blocks
  - Not vice versa.

Simple Symbol Table

- Commonly used techniques:
  - Linear table
  - Ordered list (language dependent)
  - Tree (binary tree or similar)
  - Hash table

- Works well for single scope languages
  - All variables have single scope.
  - All variables are global.
  - It is not dependent on the position of the program at which those variables are defined.

Linear Table

- It is a simple array of records corresponding to an identifier in the program.

- **Example**:

  int x, y

  real z

  …

  procedure abc

  …

  L1: …

  …

Name	**Type**	**Location**
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1

Linear Table

- If there is no restriction in the length of the string for the name of an identifier, a string table may be used with the name field holding the pointers.
- Lookup, insert, modify take O(n) time
- Insertion can be made O(1) by remembering the pointer to the next free index.
- Scanning most recent entries first may probably speed up the access
  - Due to program locality, a variable defined just inside a block is expected to be referred to more often than some earlier variables.

Ordered List

- It is a variation of linear tables in which the list organization is used.
- List is sorted and then binary search can be used with O(log n) time.
- Insertion needs more time.
- A variant of ordered list – self organizing list
  - A self-organizing list is a list that reorders its elements based on some self-organizing heuristic to improve average access time.
  - The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list.
  - A self-organizing list achieves near constant time for element access in the best case.