PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT 5

Presentation Outline

- Abstract Data Type
- Information Hiding
- Encapsulation
- Type Definition
- Static and Stack-Based Storage Management
- Fixed and Variable size heap Storage Management
- Garbage Collection

Abstract Data Types

An abstract data type is:

- ✓**A set of data objects**,
- ✓**A set of abstract operations** on those data objects
- ✓**Encapsulation of the whole** in such a way that the user of the data object cannot manipulate data objects of the type except by the use of operation defined.

Information Hiding

When information is encapsulated in an abstraction, it means that the user of the abstraction

1. **Does not need to know** the hidden information in order to use the abstraction

2. **Is not permitted** to directly use or manipulate the hidden information even if desiring to do so.

Mechanisms that support Encapsulation

- ✓Subprograms
- ✓Type definitions

Encapsulation by Subprograms and Type Definitions

- Encapsulation by Subprograms
  - Subprograms as abstract operations
  - Subprogram definition and invocation
- Type Definitions

Subprograms as abstract operations

Subprogram:

A mathematical function that maps each particular set of arguments into a particular set of results.

Specification of a subprogram

- **the name** of the subprogram
- **the signature** of the subprogram: arguments, results
- **the action** performed by the subprogram

Type checking for subprograms

Type checking: similar to type checking for primitive operations.

Difference: types of operands and results are explicitly stated in the program

Problems when describing the function computed by a subprogram

- Implicit arguments in the form of non-local variables
- Implicit results – changes in non-local variables
- History sensitiveness – results may depend on previous executions

Implementation of a subprogram

- Uses the data structures and operation provided by the language
- Defined by the subprogram **body**
  - Local data declarations
  - Statements defining the actions over the data
- Interface with the user: arguments and returned result

Implementation of subprogram definition and invocation

A simple (but not efficient) approach:

Each time the subprogram is invoked, a **copy of its executable statements**, constants and local variables is created.

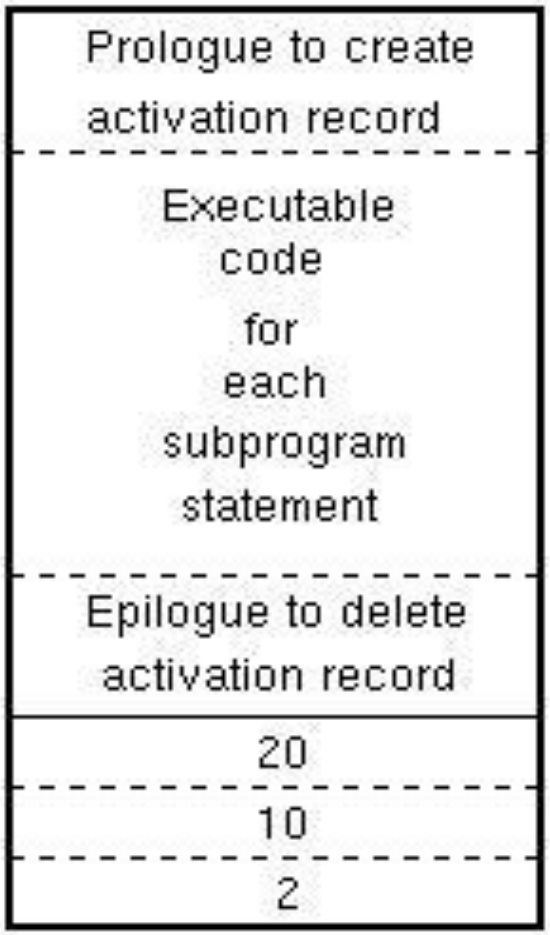
A better approach:

The executable statements and constants are invariant part of the subprogram - they do not need to be copied for each execution of the subprogram.

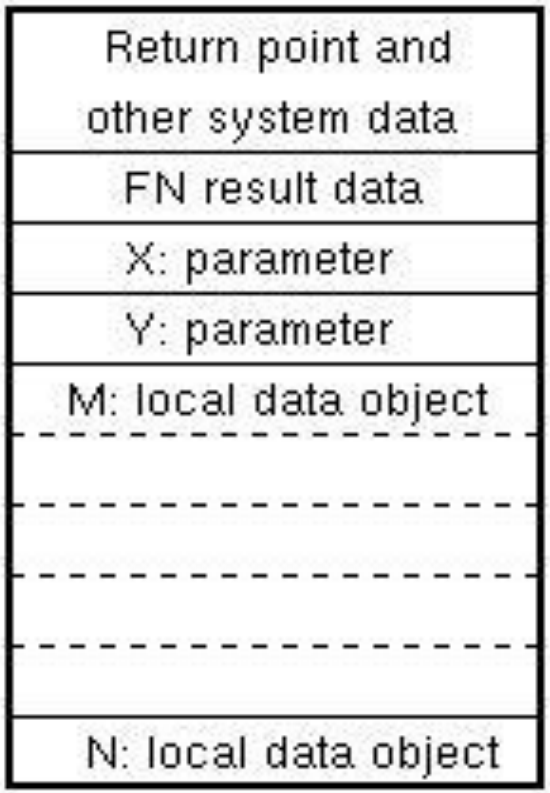
Subprogram Definition and Activation

Subprogram definition: the set of statements constituting the body of the subprogram.
Static property; the only information available during translation.

Subprogram activation: a data structure (record) created upon invoking the subprogram.
It exists while the subprogram is being executed. After that the activation record is destroyed.



Code segment for subprogram FN



Activation record for FN (Template)

The definition serves as a template to create the activa

Static code and dynamic activation record

A single copy is used for all activations of the subprogram. This copy is called **code segment**. This is the **static part**.

The **activation record** contains only the parameters, results and local data.

This is the **dynamic part**. It has same structure, but different values for the variables.

Type Definitions

Type definitions are used for definition of a new type in terms of already defined type.

They do not define a complete abstract data type, because the definitions of the operations are not included.

Format: **typedef** ***definition name***

Meaning: *definition* is already defined type.
name is substituted with ***definition*.**

Examples

```
typedef  int key_type;
key_type key1, key2;

struct rational_number
       {int numerator, denominator;}

typedef rational_number rational;

rational r1, r2;
```

Type equivalence and equality of data objects

Two questions to be answered:

- When are two types the same?
- When do 2 objects have the same value?

Name equivalence

Two data types are considered equivalent only if they have the same name.

Issues

Every object must have an assigned type, there can be no anonymous types.

A singe type definition must serve all or large parts of a program.

Structural equivalence

Two data types are considered equivalent if they define data objects that have the same internal components.

Issues

- Do components need to be exact duplicates?
- Can field order be different in records?
- Can field sizes vary?

Data object equality

Two objects are equal if each member in one object is identical to the corresponding member of the other object.

The compiler has no way to know how to compare data values of user-defined type. It is the task of the programmer that has defined that particular data type to define also the operations with the objects of that type.

Type definition with parameters

Parameters allow for user to prescribe the size of data types needed – array sizes.

Implementation

Type definition with parameters is used as a template as any other type definition during compilation.

Storage Management

Different features in a language causes different storage management techniques to be used.

FORTRAN: no recursive calls, no dynamic storage management.

Pascal: stack-based storage management.

LISP: garbage collection.

Language implementers decide about the details.

Programmers don’t know about it.

Storage Management Phases

- Initial allocation
- Recovery
- Compaction and reuse

Static Storage Management

Static allocation :

✓Allocation during translation that remains fixed throughout execution.

✓ Does not allow recursive subprograms

Static Storage Management

- Simplest
- static allocation
- no run-time storage management
- no concern for recovery and reuse
- efficient
- in COBOL and FORTRAN

Static Storage Management (Cont.)

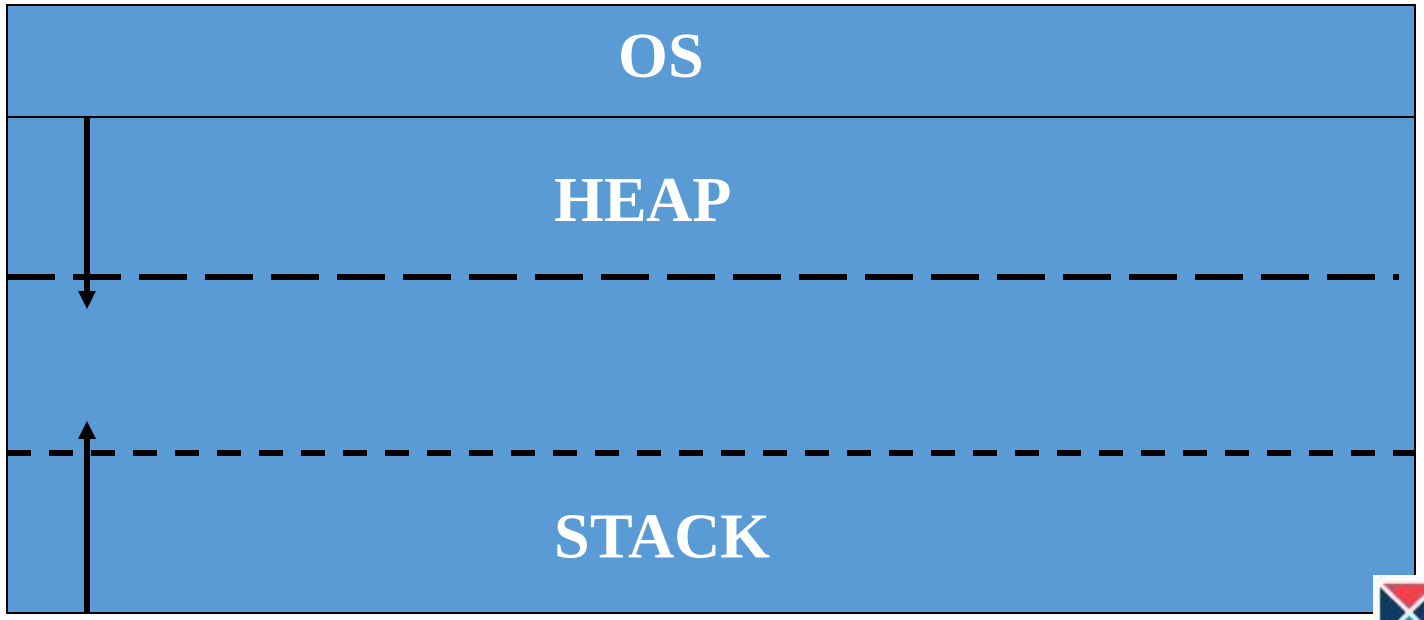
- In FORTRAN
  - each subprogram is compiled separately,
  - the code segment includes an activation record
    - compiled program,
    - its data areas,
    - return point location,
    - miscellaneous items of system data.

Stack-Based Storage Management

- Simplest run-time storage management technique.
- Based on the nested last in first out structure in subprograms calls and returns.
- Automatic compaction.
- In Pascal : a single central stack of activation records, and a statically allocated area for subprogram code segments and system programs.

Dynamic Allocation: Heap Storage Management

Memory used for dynamic allocation of data objects in somewhat unstructured manner is called **heap storage**.



Heap Storage Management

Tasks:

- allocation,
- recovery,
  - dangling references
  - garbage collection
- compaction,
- reuse

Fixed size elements

Variable size elements

Heap Storage Management: Fixed-Size Elements

- A heap is a block of storage within which pieces are allocated and freed in some relatively unstructured manner.
- Need for heap , when a language permits storage to be allocated and freed at execution time.
- Fixed size elements allocated => no need for compaction.

Recovery

The problem: identification of reusable element, solutions:

- Explicit return by programmer or system.
  - Natural, but cause garbage and dangling reference.
- Reference counts.
  - Cost of maintaining.
  - popular with parallel processing systems.
- Garbage collection.

Garbage Collection

- Dangling references more dangerous
- Two stages
  - Mark
    - garbage collection bit, set off if it is active.
  - Sweep
    - links the “on” elements to the free list.

  When is a heap element active?
  - There is a pointer to it from
    - outside the heap
    - another active heap element

Garbage Collection (Cont.)

- Three critical assumptions
  - any active element must be reachable by a chain of pointers beginning outside the heap.
  - It must be possible to identify every pointer outside the heap that points to an element inside the heap.
  - It must be possible to identify within any active heap element the fields that contain pointers to other heap elements.

Heap Storage Management: Variable-Size Elements

- More difficult
- if space for programmer defined data structures is sequential, like arrays or activation records.
- Major difficulty : reuse of recovered space.
- Initial allocation and reuse.
- reuse directly from a free-space list.
  - First-fit method
  - best-fit method

  keeping free-space list in size order.
- Recovery with variable-size blocks.
  - In first word of each block: a length indicator.
- Compaction and memory fragmentation problem.

Variable-Size Elements (Cont.)

- Compaction approaches:
  - Partial compaction
    - only adjacent free blocks
  - Full compaction
    - active blocks may be shifted

THANK YOU.. !!