Chapter 6

Data Types
Chapter 6 Topics

• Introduction
• Primitive Data Types
• Character String Types
• User–Defined Ordinal Types
• Array Types
• Record Types
• Union Types
• Pointer and Reference Types
Introduction

• A data type defines a collection of data objects and a set of predefined operations on those objects

• A descriptor is the collection of the attributes of a variable

• An object represents an instance of a user-defined type

• One design issue for all data types: What operations are defined and how are they specified?
Primitive Data Types

- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware; Others require little non-hardware support
- Examples:
  - numeric types (integer, floating point, decimal)
  - Boolean types
  - Character types
Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- Most computers support several sizes of integers (e.g., byte, short, int, long)
- Signed and Unsigned integer
- Negative integer representation
  - Sign–magnitude
  - 2’s complement (most computers use this)
  - 1’s complement
- Java’s signed integer sizes: byte, short, int, long
Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`, sometimes more)
- IEEE Floating-Point Standard 754
Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
  - Using binary codes for the decimal digits. These are called binary coded decimal (BCD).
  - Can be implemented as one digit per byte, or two digits per byte.

- **Advantage**: accuracy (not for floating-point type)
- **Disadvantages**: limited range, wastes memory
  - E.g., 6-digit BCD requires 24 bits, but it only takes 20 bits to store the same number in binary.
Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
  - Advantage: readability
Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
  - uses 0 to 127 to code 128 different characters
- Ada 95 uses ISO 8859–1
  - uses 8 bits for 256 different characters
- An alternative, 16-bit coding: Unicode
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode
Chapter 6 Topics

• Introduction
• Primitive Data Types
• Character String Types
• User-Defined Ordinal Types
• Array Types
• Record Types
• Union Types
• Pointer and Reference Types
Character String Types

• Values are sequences of characters
• Design issues:
  – Is it a primitive type or just a special kind of array?
  – Should the length of strings be static or dynamic?
Character String Type in Certain Languages

- C and C++
  - Not primitive
  - Use char arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including pattern matching
- Java
  - Primitive via the String class
Character String Types Operations

• Typical operations:
  – Assignment and copying (e.g., in C library, strcpy)
  – Comparison (=, >, etc.) (e.g., in C library, strcmp)
  – Catenation (e.g., in C library, strcat)
  – Substring reference (e.g., in Ada)
  – Pattern matching
    • Perl, JavaScript, PHP: built-in pattern match operation
    • C++, Java, C#: support in the class libraries
Character String Length Options

• **Static:**
  - Length is set when the string is created
  - COBOL, Java’s `String` class

• **Limited Dynamic Length:**
  - C and C++
  - The fixed maximum is set by variable’s definition
  - A special character is used to indicate the end of the string. So, the variable can store any number of characters between zero and the maximum.

• **Dynamic length (no maximum):**
  - SNOBOL4, Perl, JavaScript

• Ada supports all three string length options
  - E.g., String, Bounded_String, Unbounded_String
Character String Type Evaluation

- String type as a primitive type -- Aid to writability
- As a primitive type with static length, they are not costly to provide -- why not have them?
- Dynamic length is nice, but is it worth the expense?
  - Dynamic length requires the overhead of dynamic storage allocation and deallocation
  - But provide the maximum flexibility
Character String Implementation

- **Static length:**
  - compile-time descriptor
- **Limited dynamic length:**
  - may need a run-time descriptor for length (but not in C and C++)
- **Dynamic length:**
  - need run-time descriptor;
  - allocation/de-allocation is the biggest implementation problem
    - Strings are stored in a linked list
    - Strings are stored in adjacent storage cells
Compile– and Run–Time Descriptors

- Compile-time descriptor for static strings
  - Static string
  - Length
  - Address

- Run-time descriptor for limited dynamic strings
  - Limited dynamic string
    - Maximum length
    - Current length
    - Address

- Note: The limited dynamic string of C and C++ do not require run-time descriptor because the end of string is marked with null char. Maximum length field is not needed because index values are not range-checked.
Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Record Types
- Union Types
- Pointer and Reference Types
User–Defined Ordinal Types

• An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
• In some languages, users can define two kinds of ordinal types: enumeration and subrange
User–Defined Ordinal Types – Enumeration

- All possible values, which are named constants, are provided in the definition
- C# example
  ```csharp
  enum days {mon, tue, wed, thu, fri, sat, sun};
  ```
- Design issues
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?
Evaluation of Enumerated Type

• Aid to readability, e.g., no need to code a color as a number
• Aid to reliability, e.g., compiler can check:
  – No arithmetic operations are legal on enum type
  – No enum variable can be assigned a value outside its defined range
  – C treats enum variable like integer: not good
  – C++ allows numeric values be assigned to enum variable with range check: not good enough
  – Ada, C#, Java 5.0: enum variables are not coerced to integer types: good
User–Defined Ordinal Types – Subrange

• An ordered contiguous subsequence of an ordinal type
  – e.g., 12..18 is a subrange of integer type

• Ada’s design

```ada
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;
Day1: Days;
Day2: Weekday;
Day2 := Day1;  // legal unless the value of Day1 is Sat or Sun.
```

– Type check at compile time, subrange check at run–time.
Subrange Evaluation

• Aid to readability
  – Make it clear to the readers that variables of subrange can store only certain range of values

• Reliability
  – Assigning a value to a subrange variable that is outside the specified range is detected as an error
Implementation of User–Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables
Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Record Types
- Union Types
- Pointer and Reference Types
Array Types

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

- **Array Indexing** (or subscripting)
  - a mapping from indices to elements
  
  ```
  array_name (index_value_list) \rightarrow \text{an element}
  ```

- **Index Syntax**
  - FORTRAN, PL/I, Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls
  - Most other languages use brackets
Array Types

- Design issues
  - What types are legal for subscripts?
  - Are subscripting expressions in element references range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?
  - Are any kind of slices allowed?
Arrays Index Types and Range Check

- **Index (Subscript) types**
  - FORTRAN, C: integer only
  - Pascal: any ordinal type (integer, Boolean, char, enumeration)
  - Ada: integer or enumeration
  - Java: integer types only

- **Range check**
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
1. **Static**
   - subscript ranges are statically bound
   - storage allocation is static (before run-time)
   - Advantage: efficiency (no dynamic allocation)
   - E.g., C and C++ arrays that use `static` modifier

2. **Fixed stack–dynamic**
   - subscript ranges are statically bound,
   - storage allocation is done at declaration time
   - Advantage: space efficiency
   - E.g., C and C++ arrays *without* `static` modifier
Subscript Binding and Array Categories

3. **Stack–dynamic.**

- subscript ranges are dynamically bound
- storage allocation: at declaration time of the block
- storage deallocation: reach the end of the block
- Advantage: flexibility (the size of an array need not be known until the array is to be used)
- E.g., Ada arrays can be stack–dynamic

```plaintext
Get(List.Len);
declare
    List : array (1 . . List.Len) of Integer;
begin
    ...
end;
```
4. **Fixed heap–dynamic.**
   - Subscript ranges are dynamically bound
   - Storage binding is dynamic, but *fixed after allocation*
     - Binding is done when requested and storage is allocated from heap, not stack.
   - E.g., C and C++ provide fixed heap–dynamic arrays
     - C uses functions *malloc* and *free* for heap management.
     - C++ uses operators *new* and *delete*.
   - E.g., Java, C# also provides fixed heap–dynamic arrays.
Subscript Binding and Array Categories

5. Heap–dynamic:
   - binding of subscript ranges is dynamic
   - storage allocation is dynamic, and can change any number of times
   - Advantage: flexibility (arrays can grow or shrink during program execution)
   - E.g., Perl and JavaScript support heap–dynamic arrays
     ```perl
     @list = (1,2,4,7,10);  //create an array in Perl
     push(@list,13,17);  // @list = (1,2,4,7,10,13,17) now
     ```
   - E.g., C# includes a second array class `ArrayList` that provides heap–dynamic arrays.
     ```csharp
     ArrayList intList = new ArrayList();
     IntArray.Add( nextOne );
     ```
Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example
    ```c
    int list[] = {4, 5, 7, 83}; // example in C
    ```
  - Character strings in C and C++
    ```c
    char name[] = "freddie";
    ```
  - Arrays of strings in C and C++
    ```c
    char *names[] = {"Bob", "Jake", "Joe"};
    ```
  - Java initialization of String objects
    ```java
    String[] names = {"Bob", "Jake", "Joe"};
    ```
Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators.
  - A x B
    - If A and B are vectors, it will calculate the mathematical inner product of A and B.
  - A +.x B
    - If A and B are vectors, it is the sum of the inner product of A and B.
    - If A and B are matrixes, it means the matrix multiplication of A and B.
  - APL also provides unary operators to reverse the elements of a vector, or columns of a Matrix, or rows of a Matrix.
Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements.
- A jagged matrix has rows with varying number of elements.
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays.
Slices

- A **slice** is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations
- Example: Fortran 95

```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3, 4) :: Cube
```

Vector(3:6) is a four element array

What do they mean?
Mat(1:3, 2), Mat(2:3, 1:3), Cube(2,::,:)
Slices Examples in Fortran 95

MAT (1:3, 2)

MAT (2:3, 1:3)

CUBE (2, 1:3, 1:4)

CUBE (1:3, 1:3, 2:3)
Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:
  \[ \text{address(list}[k]\text{)} = \text{address (list}[\text{lower_bound}\text{)}] + ((k-\text{lower_bound}) \times \text{element_size}) \]
Accessing Multi–dimensioned Arrays

- Map multi–dimension onto single–dimension
  - because hardware memory is linear – a sequence of bytes
- Two common ways:
  - **Row** major order – used in most languages
  - **column** major order – used in Fortran
  - Example: if matrix had values below
    
    3  4  7
    6  2  5
    1  3  8

  - Row major order: 3, 4, 7, 6, 2, 5, 1, 3, 8
  - Column major order: 3, 6, 1, 4, 2, 3, 7, 5, 8
### Locating an Element in a Multi-dimensioned Array

- **General format (Row major order)**
  
  \[
  \text{Location (a}[i,j]) = \text{address of a [row\_lb, col\_lb]} + (((i - \text{row\_lb}) \times n) + (j - \text{col\_lb})) \times \text{element\_size}
  \]

---

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>j-1</th>
<th>j</th>
<th>\ldots</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>\times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

---

---
Compile–Time Descriptors

<table>
<thead>
<tr>
<th>Array</th>
<th>Multidimensioned array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Element type</td>
</tr>
<tr>
<td></td>
<td>Index type</td>
</tr>
<tr>
<td></td>
<td>Number of dimensions</td>
</tr>
<tr>
<td></td>
<td>Index range 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index range n</td>
</tr>
<tr>
<td></td>
<td>Address</td>
</tr>
</tbody>
</table>

**Single–dimensioned array**

**Multi–dimensional array**
Associative Arrays

- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**.

- **Associative Arrays in Perl (called hash)**
  - Variable names begin with `%`
    
    ```perl
    %hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, ...);
    ```
  
  - Subscripting is done using braces and keys
    ```perl
    $hi_temps{"Wed"} = 83;
    ```
  
  - Elements can be removed with `delete`
    ```perl
    delete $hi_temps{"Tue"};
    ```
Chapter 6 Topics

• Introduction
• Primitive Data Types
• Character String Types
• User–Defined Ordinal Types
• Array Types
• Record Types
• Union Types
• Pointer and Reference Types
Record Types

• A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
  – Difference between a record and an array:
    • Array consists of homogeneity of elements, and are usually referenced by indexes.

• Design issues:
  – How to define nested record?
  – What is the syntactic form of references to the field?
  – Are elliptical references allowed
Definition of Nested Records

- COBOL uses level numbers to show nested records; others use recursive definition

- COBOL: using level number

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID       PIC X(10).
      05 LAST      PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

- Any line that is followed by a line with a higher-level number is itself a record (e.g., EMP-REC, EMP-NAME)
Definition of Nested Records

- COBOL uses level numbers to show nested records; others use recursive definition
- Ada: Recursive definition in orthogonal way

```ada
type Emp_Name_Type is record
  First: String (1..20);
  Mid: String (1..10);
  Last: String (1..20);
end record;

type Emp_Rec_Type is record
  Emp_Name: Emp_Name_Type;
  Hourly_Rate: Float;
end record;

Emp_Rec: Emp_Rec_Type;
```
References to Records

1. COBOL
   field_name OF record_name_1 OF ... OF record_name_n
   e.g., First OF Emp_Name OF Emp_Rec

2. Others (dot notation)
   record_name_1.record_name_2. ... record_name_n.field_name
   e.g., Emp_Rec.Emp_Name.First

• Fully qualified references
  – must include all record names

• Elliptical references
  – allow leaving out record names as long as it is unambiguous
  – e.g., First, First OF Emp_Name, and First OF Emp-Rec are elliptical references to First in COBOL
Operations on Records

- Assignment is common if the types are identical
- Ada allows record comparison
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

```cobol
01 OUTPUT-REC.
  02 EMP-NAME.
    05 LAST  PIC X(20).
    05 MID   PIC X(10).
    05 FIRST PIC X(20).
  02 NET-RATE PIC 99V99.
MOVE CORRESPONDING EMP-REC TO OUTPUT-REC
```

It copies the FIRST, MID, LAST from EMP-REC to OUTPUT-REC
Evaluation and Comparison to Arrays

- Records are used when collection of data values is heterogeneous.
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static).
Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field

Compile-time descriptor for a record
Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Record Types
- Union Types
- Pointer and Reference Types
Unions Types

• A *union* is a type whose variables are allowed to store different type values at different times during execution

• Design issues
  – Should type checking be required?
  – Should unions be embedded in records?
Discriminated vs. Free Unions

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

type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
A discriminated union of three shape variables
Evaluation of Unions

• Potentially unsafe construct
  – Do not allow type checking
• Java and C# do not support unions
  – Reflective of growing concerns for safety in programming language
Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Record Types
- Union Types
- Pointer and Reference Types
Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
  - *Heap-dynamic variable*
Design Issues of Pointers

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

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable’s value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer’s value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

\[ j = *\text{ptr} \]

sets \( j \) to the value located at \( \text{ptr} \)
Pointer Assignment Illustrated

- A normal reference to ptr yield 7080
- A dereferenced reference to ptr yield 206
- The dereference operator in C++ uses *
  - The assignment operation \( j = *\text{ptr} \)
Problems with Pointers

1. **Dangling pointers (dangerous)**
   - A pointer points to a heap–dynamic variable that has been de–allocated
     - Pointer p1 is set to point at a new heap–dynamic variable
     - Pointer p2 is assigned p1’s value
     - The heap–dynamic variable pointed to by p1 is explicitly de–allocated (setting p1 to nil), but p2 is not changed by operation. --> p2 is now dangling.
   - Example
     ```c
     Int *arrayPtr1, *arrayPtr2;
     arrayPtr1 = new int[100];
     arrayPtr2 = arrayPtr1;
     Delete [ ] arrayPtr1;
     // now, arrayPtr2 is dangling ...
     ```
Problems with Pointers

2. Lost heap–dynamic variable
   - An allocated heap–dynamic variable that is no longer accessible to the user program (often called *garbage*).
   - Example:
     - Pointer \( p_1 \) is set to point to a newly created heap–dynamic variable.
     - Pointer \( p_1 \) is later set to point to another newly created heap–dynamic variable.
Pointers in Ada

• Some dangling pointers are disallowed because dynamic objects can be automatically de-allocated at the end of pointer's type scope (no need explicit deallocation by programmers)

• The lost heap–dynamic variable problem is not eliminated by Ada
Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (void *)
- void * can point to any type and can not be type checked (cannot be de-referenced before type assigned)
- Pointer can point to function
- Pointer can be used for parameter passing
- ...
float stuff[100];
float *p;
p = stuff;

*(p+5) is equivalent to stuff[5] and p[5]
*(p+i) is equivalent to stuff[i] and p[i]
Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
  - In C++, it is constant reference and implicitly dereferenced
  - Declared with &
    ```
    int result = 0;
    int &ref_result = result;
    ```
- Java extends C++’s reference variables and allows them to replace pointers entirely
  - References refer to class instances
  - In Java, it is not constant reference, so it can be re-assigned to different class instance.
- C# includes the references of Java and the pointers of C++
Dangling Pointer Problem

- **Tombstone**: extra heap cell that is a pointer to the heap–dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap–dynamic variable de–allocated, tombstone remains but set to nil
    - This prevents a pointer from pointing to a deallocated variable.
    - Any reference to any pointer that points to a nil tombstone can be detected as an error.
  - Costly in space and time
    - Tombstones are never deallocated, and their storage is never reclaimed.
    - Every access to a heap dynamic variable through a tombstone requires one more level of indirection.
Dangling Pointer Problem

- **Locks–and–keys:**
  - Pointers: (key, address) pairs
  - heap–dynamic variables: (lock, variable) pairs
  - When heap–dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer
  - When dereferencing a pointer, compare the key of the pointer to the lock of the heap–dynamic variable.
  - When a heap–dynamic variable deallocated with dispose, its lock value is cleared to an illegal lock value.
  - If a pointer other than the one specified in the dispose is dereferenced, its key value will not match the lock in variable, so the access is disallowed.
Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers or references are necessary for dynamic data structures—so we can't design a language without them
- The best solution to dangling pointer problem is to take deallocation of heap–dynamic variables out of the hands of programmers.
Heap Management

• A very complex run–time process
• Single–size cells vs. variable–size cells
• Single–size cells
  – Two approaches to reclaim garbage
    • Reference counters (*eager approach*): reclamation is gradual
    • Garbage collection (*lazy approach*): reclamation occurs when the list of variable space becomes empty
Single-size cells – Reference Counter

• Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  – Disadvantages:
    • space required (for the counter),
    • execution time required (to maintain counter values),
    • complications for cells connected circularly.
Single-size cells – Garbage Collection

- The run-time system allocates storage cells and disconnects pointers from cells later without storage reclamation (so, garbage accumulate)
- Garbage collection begins if no available cells.
  - Every heap cell has an extra bit as a flag
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All the cells that are not marked as reachable are garbage and are returned to list of available cells
- Disadvantages:
  - when you need it most, it works worst
Variable-Size Cells

• More difficulties than single-size cells, but most programming languages required it
  – Maintaining the list of available space is costly.
    • After the list becomes a long list of various-size blocks, slow allocation occurs because requests cause the list to be searched for sufficiently large blocks.
  – If garbage collection is used, more problems:
    • The initial setting of the indicators of all cells in the heap is difficult, because the cells are different sizes
      – Solution: each cell have cell size as its first field
    • The marking process is nontrivial.
  – Fragmentation
Summary

• The data types of a language are a large part of what determines that language’s style and usefulness.
• The primitive data types of most imperative languages include numeric, character, and Boolean types.
• The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs.
• Arrays and records are included in most languages.
• Pointers are used for addressing flexibility and to control dynamic storage management.