ICOM 4036
Programming Languages

Data Types

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

This lecture covers review questions 21-29
And problems 5-18

Dr. Amirhossein Chineai, ECE, UPRM Spring 2010
***Some slides are adapted from the Sebesta’s textbook

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*
  – 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*)

• Increase writability

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 *

\[
j = \ast \text{ptr}
\]
sets \( j \) to the value located at the address that is stored in \( \text{ptr} \)
Pointer Assignment Illustrated

The assignment operation \( j = *\text{ptr} \)

Problems with Pointers

- **Dangling pointers** (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated (often due to explicit deallocation)
- **Lost heap-dynamic variable**
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - 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
    - The process of losing heap-dynamic variables is called *memory leakage*

Pointers in Ada

- Are called *access* types
- Some dangling pointers are disallowed because dynamic objects are implicitly deallocated at the end of pointer’s type scope
- Some dangling pointers still possible (via UNCHECKED_DEALLOCATION)
- The lost heap-dynamic variable problem is not eliminated by *Ada*

Pointers in C and C++

- Extremely flexible but must be used with extra care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for both dynamic storage management and indirect addressing
- Pointer arithmetic is possible
- Explicit dereferencing and *address-of* operators
- Domain type need not be fixed (*void* *)
  - Provides generic pointers
    - *void* can point to any type and can be type checked (cannot be dereferenced)
Pointer Arithmetic in C and C++

```c
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
  - It’s constant, and dereferenced implicitly
  - Advantages of pass-by-reference (in addition to pass-by-value)
- Java extends C++’s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
  - DEALLOCATED IMPLICITLY
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- **Dangling pointers** and **dangling objects** (garbage) are problems as is heap management
- Pointers are like **goto’s**—they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures—so we can’t design a language without them
- **Representation of Pointers**
  - Intel microprocessors use two values:
    - **segment** and **offset**

Solutions to Dangling Pointer

- **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**
  - Costly in time and space
- **Locks-and-keys**: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer
  - Still costly
- The best solution is to **disallow explicit deallocation**
Heap Management

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

Reference Counter

- **Reference counters**: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - **Disadvantages**: space required, execution time required, complications for cells connected circularly
  - **Advantage**: it is intrinsically incremental, so significant delays in the application execution are avoided

Garbage Collection

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - **Disadvantages**: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

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