Programming Languages:

Lecture 6

Chapter 6: Data Types

Jinwoo Kim
jwkim@jjay.cuny.edu

Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- 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 (abstract data) type
- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

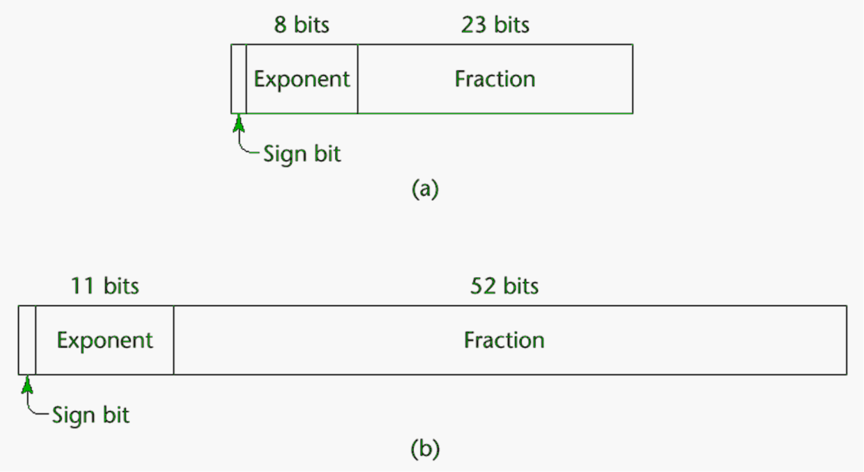
- Almost all programming languages provide a set of *primitive data types*
  - Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require little non-hardware support

Primitive Data Types: Integer

- Most common primitive numeric data type
  - Many computers support several sizes of integers
  - Almost always an exact reflection of the hardware so the mapping is trivial
- Java's signed integer sizes: `byte, short, int, long`
- C++ and C# include unsigned integer types

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**
- Most newer machines use IEEE Floating-Point Standard 754 format
  - Single and Double precision



Primitive Data Types: Decimal

- Store a fixed number of decimal digits
  - With decimal point at a fixed position in the value
- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all types
- 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
- An alternative, 16-bit coding: Unicode
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode

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 Types Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Certain Languages

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

Character String Length Options

- Static: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In C-based language, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Character String Type Evaluation

- Aid to writability
  - Dealing with strings as arrays can be more cumbersome than dealing with a primitive string type
- As a primitive type with static length, they are inexpensive to provide--why not have them?
  - Addition of strings as a primitive type to a language is not costly in terms of language and compiler complexity
- Dynamic length is nice, but is it worth the expense?
  - Advantage: flexibility
  - Disadvantage: overhead from its implementation
  - Often included only in languages that are interpreted

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/deallocation is the biggest implementation problem

Compile- and Run-Time Descriptors

Static string
Length
Address

Compile-time descriptor for static strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor for limited dynamic strings

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

- Examples of primitive ordinal types in Java
  - `integer`
  - `char`
  - `Boolean`

- In some languages, users can define two kinds of ordinal types
  - Enumeration
  - Subrange

Enumeration Types

- All possible values, which are named constants, are provided in the definition

- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  - 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:
    - **operations (don't allow colors to be added)**
    - **No enumeration variable can be assigned a value outside its defined range**
    - **Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types**

Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type

- Ada’s design

```
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;
```

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

Array Types

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

Array 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?

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  `array_name (index_value_list) → an element`

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

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Pascal: any ordinal type (integer, Boolean, char, enumeration)
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- C, C++, Perl, and Fortran do not specify range checking
- Java, ML, C# specify range checking

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)

- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound, and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility
    - **Size of an array need not be known until the array is to be used**

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation
  - i.e., binding is done when requested and storage is allocated from heap, not stack

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- Ada arrays can be stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl and JavaScript support heap-dynamic arrays

Static in C or C++

- The **static** keyword has several distinct meanings
  - Life time of variable declared locally to function is only during function calls
  - What should I do if I want to retain values between function calls?
  - Why not use global variables then?

```
// Using a static variable in a function
void func() {
    static int i = 0;
    cout << "i = " << ++i << endl;
}
int main() {
    for(int x = 0; x < 10; x++)
        func();
}
```

Static in C or C++ (Continued)

- When **static** is applied to a function name or to a variable that is outside of all functions, it means “This name is unavailable outside of this file.”
  - The function name or variable is local to the file

```
// File scope means only available in this file:
static int fs;
int main() {
    fs = 1;
}
```

```
// Trying to reference fs in another file
extern int fs;
void func() {
    fs = 100;
}
```

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators
  - For example, to reverse column elements
- Ada allows array assignment but also catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

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*
  - User defined keys must be stored

- Design issues: What is the form of references to elements

Associative Arrays in Perl

- Names begin with %

```
%hi_temps = ("Mon" => 77, "Tue" => 79, “Wed”
  => 65, …);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

Elements can be removed with `delete`

```
  delete $hi_temps{"Tue"};
```

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed?

Definition of Records in Cobol

- COBOL uses level numbers to show nested records
  - Level numbers in COBOL shows hierarchical structure
  - Other languages usually use recursive definition

```
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.
```

Definition of Records in COBOL (Continued)

- Record Field References
  - field_name OF record_name_1 OF ... OF record_name_n
    - **record_name_1 : innermost record that contains the field_name**
    - **record_name_n : outermost record that contains the field_name**
  - Example
    - `MID OF EMP-NAME OF EMP-REC`

Definition of Records in Ada

- Record structures are indicated in an orthogonal way

```
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

- Most language (including C and C++) use dot notation

  `Emp_Rec.Emp_Name.Mid`

- ***Fully qualified references*** must include all record names

- ***Elliptical references*** allow leaving out record names as long as the reference is unambiguous
  - For example, in COBOL

  `FIRST`, `FIRST OF EMP-NAME`, and `FIRST of EMP-REC` are elliptical references to the employee's first name

Operations on Records

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

MOVE CORRESPONDING in COBOL

```
01 INPUT-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 EMP-NUMBER.
   02 HOURS-WORKED PIC 99.
```

```
01 OUTPUT-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 EMP-NUMBER.
   02 GROSS-PAY PIC 999V99
   02 NET-PAY PIC 999V99.
```

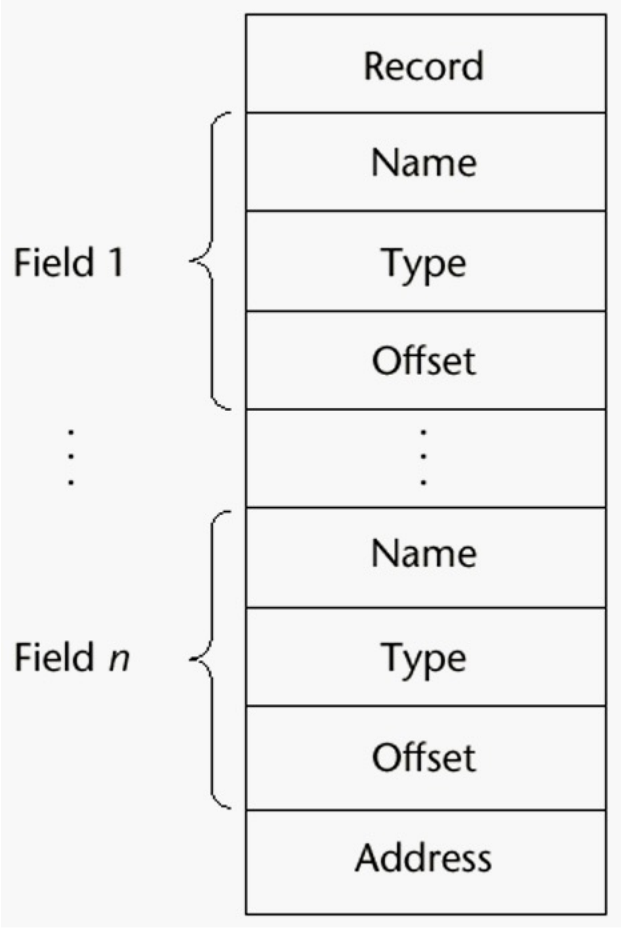
```
MOVE CORRESPONDING INPUT-REC TO OUTPUT-REC.
```

Records Evaluation and Comparison to Arrays

- Records and arrays are closely related structural forms
  - Arrays are used when all the data values have the same type and are processed in the same way
  - Records are used when collection of data values is heterogeneous and different fields are not processed in the same way
- Field names in Record are usually static
  - So, it is efficient
- Access to array elements is also efficient in static arrays
  - But, much slower than access to record fields
    - **When subscripts are dynamic**
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type: Compile time descripton for a Record

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



Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
  - Store different data types in the same memory location
  - Union can be defined with many members, only one member can contain a value at any given time
  - Efficient way of using the same memory location for multiple-purpose

- Design issues
  - Should type checking be required?
    - **Any such type checking must be dynamic**

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*
    - ***Since programmers are allowed complete freedom from type checking in their use***
- Type checking of unions require that each union include a type indicator called a ***tag*** or ***discriminant***
  - Supported by ALGOL 68 and Ada

Why Union type? (Example codes in C++)

- Binary tree implementation
  - Internal node
    - ***Two pointer members to two children, no data member stored***
  - *Leaf node*
    - ***Only contains data without pointers***

```
struct NODE {
    struct NODE* left;
    struct NODE* right;
    double data;
}
```

```
struct NODE {
    bool is_leaf;
    union {
          struct {
                 struct NODE* left;
                 struct NODE* right;
          } internal;
          double data;
    }info;
};
```

Free Unions can be Dangerous

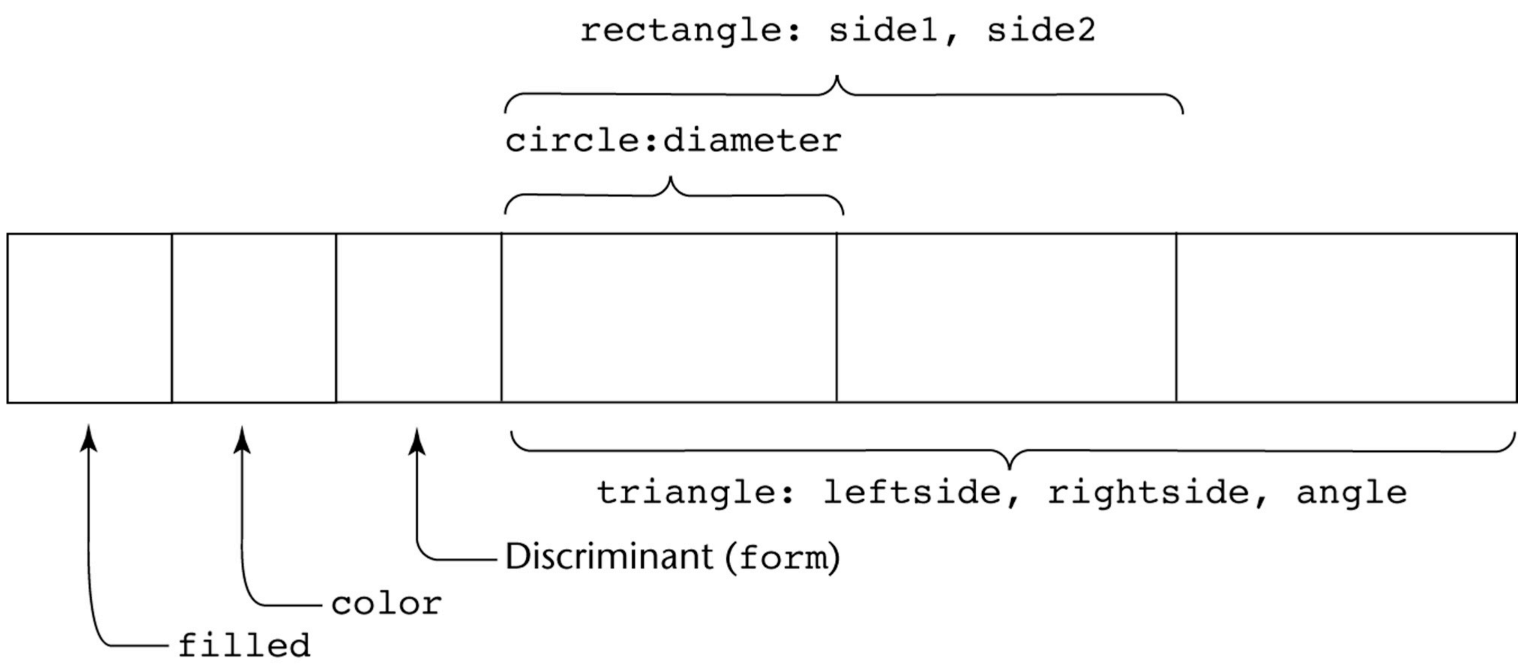
```
union flexType {
   int intEl;
   float floatEl;
};

union flexType element;

float x;
. . .
element.intEl = 27;
x = element.floatEl;
```

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;
```

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

Ada Union Type Illustrated



A discriminated union of three shape variables

Evaluation of Unions

- Potentially unsafe construct
  - Do not allow type checking
    - **One of the reasons why Fortran, C and C++ are not strongly typed**
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- One exception
  - Unions in Ada can be safely used by its design

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*)

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?

Arrays

- Data structure used to store a group of objects of the same type sequentially in memory
  - All the elements of an array must me same data type
  - Since the elements of the array are stored in sequentially in memory, it allows convenient and powerful manipulation of array element using pointers
  - Datatype arrayName[size];
    - **Ex: int id[30];**
    - **char name[20];**
    - **float height[10];**
  - Array indices in C++ are numbered starting at zero not one!
    - **id[0], id[1], … , id[29] for above example**
  - Arrays cannot be copied using the assignment operator
    - **int a[5], b[5];**
    - **…**
    - **a = b; // illegal!!!**

Arrays (Continued)

- Arrays passed to functions can be modified

```
void foo(int arr[]) {
        arr[0] = 42; // modifies array
        return 0;
}

…
int my_array[5] = {1, 2, 3, 4, 5};
foo(my_array);
cout << “my_array[0] is “ << my_array[0];
```

Pointers

- Address
  - A location in memory where data can be stored
  - Ex: A variable or an array

- Pointer
  - A variable which holds an address

Pointers (Continued)

Example:

```
int i = 10;

int *j = &i;

cout << ″i = ″ << i << endl;

cout << ″j = ″ << j << endl;

cout << ″j points to: ″ << *j << endl;
```

Pointers (Continued)

- & is reference operator
  - **&i** is the address of variable i

- * is dereference operator
  - *j is the contents of the pointer variable j
    - **what j points to**
  - *j dereferences the pointer j
  - * is used as multiplication and when declaring a pointer variable also
    - **Ex:**

```
int i = 10;
int *j = &i;
int k = i * (*j);
```

Pointer arithmetic

- We can add/subtract integers to/from pointers

```
int i[5] = { 1, 2, 3, 4, 5 };
int *j = i;  // (*j) == ?
j++;         // (*j) == ?
j += 2;      // (*j) == ?
j -= 3;      // (*j) == ?
```

Arrays and pointers

- Arrays are pointers in disguise

```
int i[5] = { 1, 2, 3, 4, 5 };
cout << "i[3] = " << i[3] << endl;
cout << "i[3] = " << *(i + 3) << endl;
```

- i is same as &i[0] from above example

Arrays and pointers (Continued)

- Multi-dimensional array and pointer

```
int zippo[4][2];   // int array of 4 rows and 2 columns
int *pri;          // pointer to integer
pri = zippo;       // zippo == &zippo[0][0]
// pri == &zippo[0][0]          row 1, column 1
// pri + 1 == &zippo[0][1]      row 1, column 2
// pri + 1 == &zippo[1][0]      row 2, column 1
// pri + 1 == &zippo[1][1]      row 2, column 2
...
```

Arrays and pointers (Continued)

- Two versions of same operations

```
int array[1000];

for (i = 1; i < 999; i++) {
  array[i] = (array[i-1] + array[i]);
}

for (i = 1; i < 999; i++) {
  *(array+i) = (*(array+i-1) + *(array+i));
}
```

- But is it same in terms of performance?

Passing arguments into functions in C++

- The way variables or data maybe passed into a function in C++
  - Pass by value
  - Pass a pointer by value
  - Pass by reference

Pass by value

- The function receives a copy of the variable
  - This local copy has scope (exists only within the function)
  - Any changes to the variable made in the function does not affect the variable in the calling function
  - Good since it is simple and guarantees no change of the variable in the calling function
  - Bad in the following reasons
    - **Sometimes it is inefficient to copy a variable (when?)**
    - **Only way to pass information from called function to calling function is through return value (C/C++ has only 1 return value)**

Pass by value (Continued)

```
Example:

void IncreaseMe(int theInt);
main()
{
	int i;
	i = 5;
	IncreaseMe(i);

	cout << "i is " << i << " \n";
}

void IncreaseMe(int i)
{
	i = i + 1;
}
```

Passing a pointer by value

- A pointer to the variable is passed to the function
  - The pointer can then be manipulated to change the value of the variable in the calling function
  - The function cannot change the pointer itself since it gets the local copy the pointer
  - But the called function can change the contents of the memory location (variable) to which the pointer refers
  - Good in the following reasons
    - **Any changes to variables will be passed back to the called function**
    - **Multiple variables can be changed**

Passing a pointer by value (Continued)

```
Example:
void IncreaseMe2(int *theInt);

main()
{
     int i;
     int *pt;
     i = 5;
     pt = &i;  // set the pointer to
               // the address of i

     IncreaseMe2(pt);
     cout << "i is " << i << " \n";
}

void IncreaseMe2(int *i)
{
     *i = *i + 1;
}
```

Pass by reference

- A reference in C++ is an alias to a variable
  - Any changes made to the reference in the called function will also be made to the original variable in the calling function
  - Good in the following sense
    - **It is avoiding complicated pointer notation**
    - **Efficient since no local copy of the variables are made in the called function**
    - **Multiple variables can be changed**

Pass by reference (Continued)

```
Example:

void IncreaseMe3(int &theInt);
main()
{
      int i;
      i = 5;
      IncreaseMe(i);

      cout << "i is " << i << " \n";
}

void IncreaseMe3(int &i)
{
      i = i + 1;
}
```

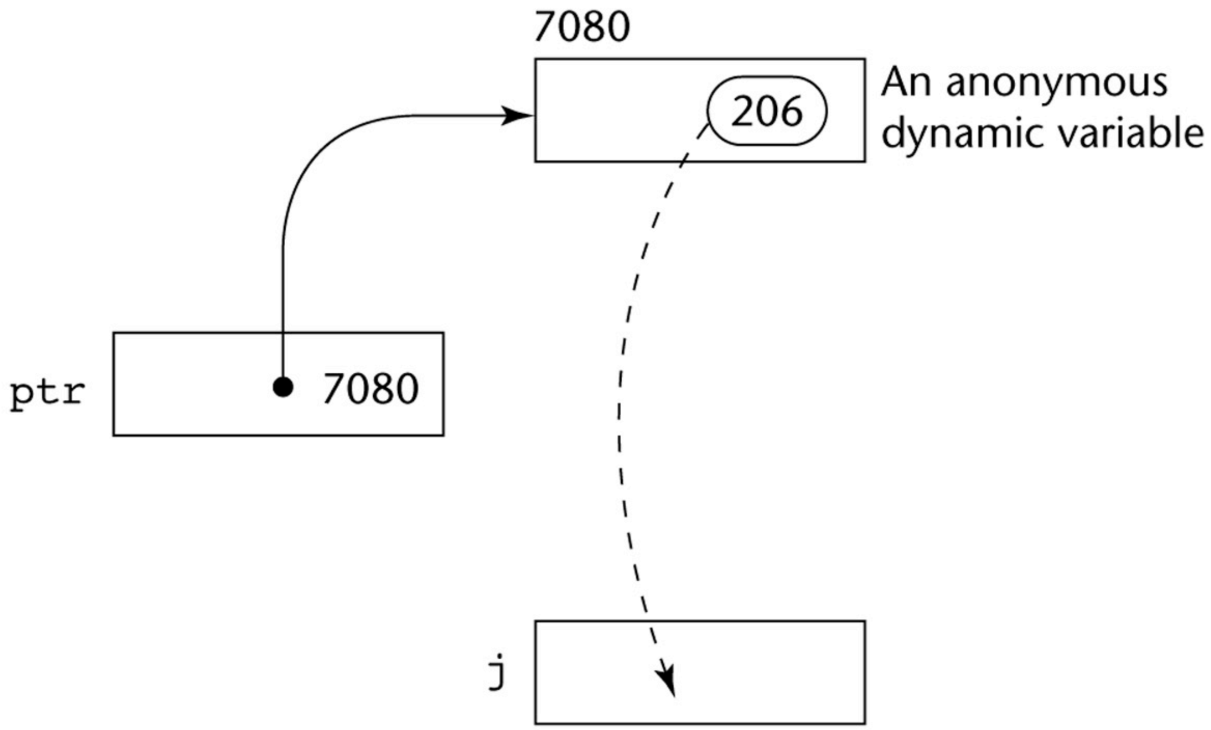
Pointer Operations

- 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 = *ptr`

    sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been de-allocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - **Pointer `p1` is set to point to a newly created heap-dynamic variable**
    - **Pointer `p1` 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

- 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 be type checked (cannot be de-referenced)

Pointer Arithmetic in C and 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]`

Pointers in Fortran 95

- Pointers point to heap and non-heap variables
- Implicit dereferencing
- Pointers can only point to variables that have the `TARGET` attribute
- The `TARGET` attribute is assigned in the declaration:

```
INTEGER, TARGET :: NODE
```

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
- Java extends C++'s reference variables and allows them to replace pointers entirely
  – References refer to call instances
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- Dangling pointers and dangling objects 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

Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

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
  - 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

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
  - Garbage collection (*lazy approach*): reclamation occurs when the list of variable space becomes empty

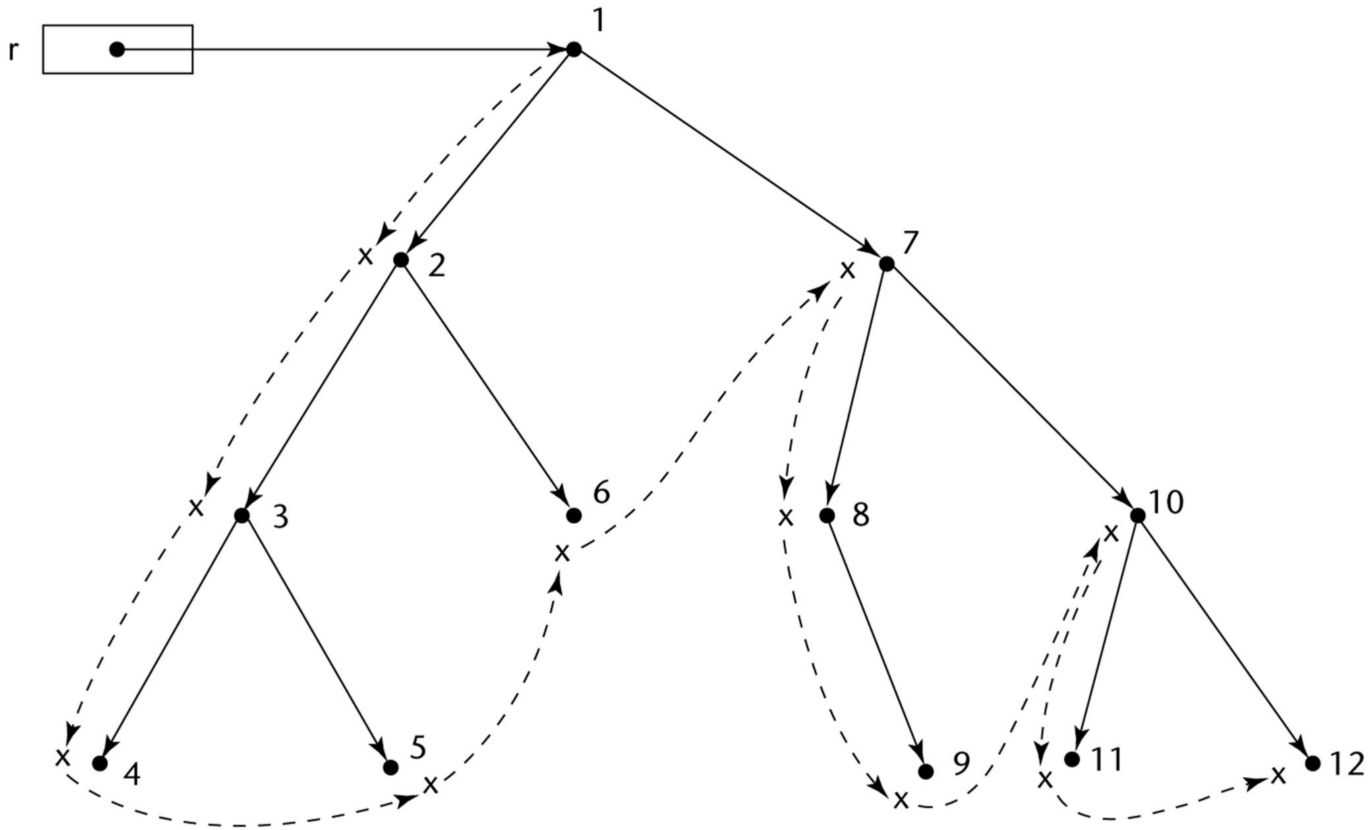
Reference Counter

- 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

Garbage Collection

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; garbage collection 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: when you need it most, it works worst (takes most time when program needs most of cells in heap)

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If garbage collection is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

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

Homework #3 (part 2)

- Programming Exercise (P.318 of class textbook)
  - Question 7
- Problem Solving (P. 316 of class textbook)
  - 2,15
- Due date: One week from assigned date
  - Please hand in printed (typed) form
    - **I do not accept any handwritten assignment**
    - **Exception: pictures**