Data 2/1/2018

|  |  |
| --- | --- |
| **Data**  In programs, the values (i.e., data) of variables change allowing programs to be able to handle multiple sets of data.  //All this shit on exam  //eg int myArray[20] = [0.0,0,0,123];  //question is describe each characteristic below of the var above  Type = int  Struct= homogeneous array  Size = 20 \* 4 = 80  Value = 0, 0, 0, 0, 123  String s = “hello”  Location = s in auto  Obj in heap  Value is in string literal pool  Type = string  Struct =  Size =  Value = hello  Characteristics of data:  **Location** where it is located  **Data type** describes acceptable values (char, integer, float, boolean)  **Structure** primitive, homogeneous array, record structure; self-referencing; object with attributes x, y, and z  **Size** size can be in bits or bytes; fixed or variable length; lower bounds and upper bounds; unbounded vs bounded  **Value** can vary or it can be immutable;  Additionally, we have the concept of a **descriptor** which describes the data, and often includes data type, structure, and size. Almost all languages have descriptors during translation. A language can be less efficient (time) if descriptors are necessary during execution.  Strings are represented differently in various programming languages. (Please see the examples to the right and later in this document.)  What are the advantages/disadvantages of how C represents strings?  In Java, a String variable's value can be changed; however, an actual string is immutable (i.e., cannot be changed). Why? | **static/ global variables -** given location at compile/load time  **automatic variables** - given location from the runtime memory stack as an offset within an activation record  FORTRAN   * data type, structure and size are bound at compile time * Except for parameters, variables are bound to locations at compile/load time   C   * data type, structure and *size* are bound at compile time * arrays can be given a size at runtime (with C99, size can change at runtime) * static and global variables are bound to location at compile/load time * automatic variables are bound to locations at runtime (offsets within an activation record)   PL/I   * data type and structure are bound at compile-time * static and automatic variables * array size can be bound at execution time * array and string parameters can receive the maximum size from the arguments   so   * descriptors are available at runtime to describe arrays (lower bound, upper bound) * descriptors are available at runtime to describe strings (current size, maximum size)   Java   * datatype and structure are bound at translation time * size of arrays can be provided at runtime * parameters receive a reference to an argument which can be an object * objects contain information which describes what they need:   + strings have current value location, offset, and length   + other classes whatever they need (e.g., size)   LISP   * most bindings (location, structure, size, data type) are done at execution time * data requires runtime descriptors |
| **Primitives**  **Boolean**  **Integer**  **Float**  **Decimal**  **Character**  **Enumerations** |  |
| **Primitives**  **Boolean -** logic value of true or false.   * true is usually 1 (or non-zero) * false is usually 0   **Integer -** positive and negative numbers   * typically represented as 2's complement (see CS3843) * sizes range from 1 byte to 8 bytes * may be in little endian or big endian order (see below)   **Hex Constants**  Many languages allow specification of integer values as hexadecimal constants (base 16 instead of base 10).  0x23C is 2\*162+3\*161+12=572  Suppose the 4 byte integer value 0x789ABCDE needs to be stored in memory. There are two different well known hardware-dependent representations for storing in memory:  **little endian** store the least significant digit first. This is used by Intel chips.  **big endian** store the least significant digit last.  When storing 0x789ABCDE in memory  little endian: DE BC 9A 78  big endian: 78 9A BC DE  Integers, 2's comp, hexadecimal, little endian, and big endian are covered more thoroughly in CS3843 Computer Organization. | **Boolean** data types in various languages:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Language | Data Type | True | False | Notes | | PL/I | BIT(1) | '1'B | '0'B |  | | C | int | 1 | 0 | No actual boolean type. | | Java | boolean | true | false | Constants true and false print as **true** and **false**. | | Python | bool | True | False | Constants True and False have values 1 and 0 respectively, but print as **True** and **False**. |   **Integer** data types in various languages:  32-bit C (most hardware, compilers):   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Data Type** | **Size in Bytes** | **Size in Bits** | **Smallest Value** | **Largest Value** | | short | 2 | 16 | -32,768 | 32,767 | | int | 4 | 32 | -2,147,483,648  (approx -2 billion) | 2,147,483,647  (approx +2 billion) | | long | 4 | 32 | -2,147,483,648  (approx -2 billion) | 2,147,483,647  (approx +2 billion) |   64-bit C (most hardware, compilers):   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Data Type** | **Size in Bytes** | **Size in Bits** | **Smallest Value** | **Largest Value** | | short | 2 | 16 | -32,768 | 32,767 | | int | 4 | 32 | -2,147,483,648  (approx -2 billion) | 2,147,483,647  (approx +2 billion) | | long | 8 | 64 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |   PL/I:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Data Type** | **Size in Bytes** | **Size in Bits** | **Smallest Value** | **Largest Value** | | FIXED BINARY(15) | 2 | 16 | -32,768 | 32,767 | | FIXED BINARY(31) | 4 | 32 | -2,147,483,648  (approx -2 billion) | 2,147,483,647  (approx +2 billion) |   Java:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Data Type** | **Size in Bytes** | **Size in Bits** | **Smallest Value** | **Largest Value** | | byte | 1 | 8 | -128 | 127 | | short | 2 | 16 | -32,768 | 32,767 | | int | 4 | 32 | -2,147,483,648  (approx -2 billion) | 2,147,483,647  (approx +2 billion) | | long | 8 | 64 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |   Python (with 3.x):   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Data Type** | **Size in Bytes** | **Size in Bits** | **Smallest Value** | **Largest Value** | | int | unlimited | unlimited | unlimited | unlimited | |
| **Primitives - Floating Point**  Floating Point provides representation for very large values and fractional values (close to 0). Most computers use the IEEE 754 Standard.  It represents floating point binary values in a manner similar to scientific notation. This allows the point to float.   |  |  |  | | --- | --- | --- | | *s* | *exp* | *mantissa* |   where  *s* sign  *exp* exponent which can be negative  *mantissa* coefficient (aka, significand)  **Note that floating point is not decimal.**  This is covered more thoroughly in CS3843. | Formats:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **format** | **number of bits** | | | | | sign | exp | mantissa | total | | float | 1 | 8 | 23 | 32 | | double | 1 | 11 | 52 | 64 | | extended | 1 | 15 | 64 | 80 |   Floating point data types:   |  |  |  | | --- | --- | --- | | **Language** | **float** | **double** | | PL/I | FLOAT DEC(6) | FLOAT DEC(16) | | COBOL | USAGE IS COMP-1 | USAGE IS COMP-2 | | C | float | double | | Java | float | double | | Python | n/a | float |   Floating point constants  3.14159 (Note that this can be confused with decimal values in languages supporting decimal.)  1.8318528e9  3.141159E0 (PL/I floating point constant for PI)  The value (0.3)10 cannot be represented accurately with floating point. It is binary 0.0100. |
| **Primitives - Decimal**  The decimal data type is used to more accurately represent decimal values. This makes it easier when dealing with money values.  Some hardware (e.g., IBM mainframes) support decimal arithmetic providing more efficient and accurate decimal operations.  With **packed numeric** (i.e., **Binary Coded Decimal**) representations, each digit is stored as 4 bits. | PL/I:  /\* declare Account Balance to have 8 digits to the left and 2 digits to the  \*\* right of the decimal point  \*/  DECLARE ACCOUNT\_BALANCE FIXED DEC(10,2);  COBOL:  05 ACCOUNT-BALANCE PIC S9(8)V99 USAGE IS COMP-3. |
| **Primitives - Character**  (We will discuss character strings after primitives.)  In some languages (PL/I, FORTRAN, COBOL, C), characters are 8 bits and cannot support natural languages with more complex alphabets/symbols. Depending on the platform, the characters use either ASCII or EBCDIC (IBM).  In some other languages (Java, Python), characters are unicode values which take multiple bytes for a single symbol. Most Java implementations use UTF-16 (2 bytes per character). | |  |  |  |  | | --- | --- | --- | --- | | **Letter** | **ASCII Decimal** | **ASCII (Hex)** | **EBCDIC (HEX)** | | A | 65 | 41 | C1 | | B | 66 | 42 | C2 | | C | 67 | 43 | C3 | | D | 68 | 44 | C4 | | E | 69 | 45 | C5 | | F | 70 | 46 | C6 | | G | 71 | 47 | C7 | | H | 72 | 48 | C8 | | I | 73 | 49 | C9 | | J | 74 | 4A | D1 | | K | 75 | 4B | D2 | | L | 76 | 4C | D3 | | M | 77 | 4D | D4 | | N | 78 | 4E | D5 | | O | 79 | 4F | D6 | | P | 80 | 50 | D7 | | Q | 81 | 51 | D8 | | R | 82 | 52 | D9 | | S | 83 | 53 | E2 | | T | 84 | 54 | E3 | | U | 85 | 55 | E4 | | V | 86 | 56 | E5 | | W | 87 | 57 | E6 | | X | 88 | 58 | E7 | | Y | 89 | 59 | E8 | | Z | 90 | 5A | E9 | |
| **Primitives - Enumerations**  **Enumerations** provide the entire set of possible values as named constants.  Internally, each value of an enumeration is assigned an integer which makes it more usable in **case** statements. | **Example #1: Enumerations in C, C++, and Java.**  C:  typedef enum {Sunday, Monday, Tuesday, Wednesday  , Thursday, Friday, Saturday} Days;  Days day = Sunday;  …  switch(day);  {  case Sunday:  printf("Week begins\n");  break;  case Wednesday:  printf("Hump Day\n");  break;  }  C++:  enum Colors {blue, green, yellow, red};  Colors colorChoice = green;  Java:  public enum Days  {  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday  }  …  Days day; |
| **Data Representations Beyond Primitives (i.e., data structures)**  **Strings**  **Homogeneous Arrays**  **Heterogeneous Arrays**  **Associative Arrays**  **Record Structures**  **Unions** |  |
| **Data Structure Considerations**   1. **What is the number of components?** 2. **Can it grow?** 3. **Can the data type of each component vary?** 4. **How are the components referenced?** 5. **How can modifications be made to the components?** 6. **How are referencing errors handled?** 7. **How can code access metadata (current characteristics) about the data structure?** | **Example #2: C - homogeneous arrays**   * Declared array size is known during compilation. (With C99, array size can be provided at runtime.) * An array in C is declared with a fixed number of elements and cannot grow   int scoreM[30]; // declares an array of 30 integers   * Components (aka, elements) are referenced by subscripts which vary from 0 to (but not including) a maximum number.   scoreM has subscripts from 0 to 29   * References: scoreM[*subscript*]   + the *subscript* can be any expression which results in an integer value * Assignment: scoreM[*subscript*] = *expression*;   + replaces the value of scoreM[*subscript*] * Parameters do not know the array bounds * C does not detect out of bounds conditions; therefore, code can reference invalid elements and assign values outside the range of the array * Limited metadata at runtime. sizeof can be used where the array was declared (but not for a parameter). sizeof is bound at compile time.   What is the advantage of C not detecting out of bounds conditions? |
| **Data Representations - Character Strings**  1. Fixed declared length - size, data type, and structure are known. Location may be static or automatic.   * Compiler knows everything * Runtime descriptors are not necessary   2. Variable-length within a declared maximum - max string length is specified. Current length is known by either   * current length field   + compiler can generate code which understands max length * current length marker (aka, sentinel)   + compiler can generate code which understands max length   + For C, compiler doesn't generate code which understands max length and there isn't a runtime descriptor. | **Example #3: Character String Representations**  1. Fixed declared Length    2. Variable length within a declared max - using current length    Variable-length with a marker |
| 3. Unbounded length - can vary without bound   * current length and string location   + contiguous memory for value   + This is used by both mutable and immutable implementations. * current length, offset, and string location   + contiguous memory for value   + This is used by immutable implementations. * segmented - current length, pointer to linked list of partial values   + value represented by a linked list   + less fragmentation of memory   + links (i.e., pointers) take memory | **Example #3 continued: Character String Representations**  3. unbounded length    current length, offset, and string location    segmented: |
|  | **Example #4: character strings in various languages**  Representing strings varies in languages  COBOL - fixed-length; parameters must be declared with same size  01 STUDENT.  02 ABC123 PIC X(6).  02 NAME PIC X(30). \*> padded on right with spaces.  PL/I - fixed-length or variable-length (size and value within a declared max size); parameters can receive descriptors (specifying max size and location)  DCL ABC123 CHAR(6), /\* padded on right with spaces. \*/  NAME CHAR(30) VARYING;  DCL NAME CHAR(\*) VARYING; /\* receives a descriptor \*/  To access current length: LENGTH(NAME)  To access individual characters: SUBSTR(NAME,I,1)  To access maximum size of NAME in original declaration: MAXLENGTH(NAME)  To access maximum size of NAME in parameter: MAXLENGTH(NAME)  C - variable-length (marker); parameters don't know maximum size.  char szName[31] = "Joe King"; // C uses a zero byte for markers  void func(char \*pszName)  void func(char szNameParm[])  To access current length: strlen(szName)  To access individual characters: szName[i]  To access maximum size of szName in original declaration: sizeof(szName)  To access maximum size of szNameParm: not possible  What does sizeof(szNameParm) return?  C++ - unbounded-length (current length, current allocated size (most implementations), location); if more space is needed, it grows automatically  std::string s1 = "Lee King";  std::string s2("Rea King");  To access current length: s1.length()  To access individual characters: s1[i]  To access maximum size:  s1.max\_size() - really huge  s1.capacity() - currently allocated size although when exceeded, it can grow  Note: the variables will be in either static or automatic memory. The string value itself will be in either the heap. |
|  | **Example #4 continued: character strings in various languages**  Java - unbounded-length, immutable (String class) has size, offset, and location; char arrays allow changes  String name = "Ray";  name = "Roy";  name = name + " King";  To access current length: name.length()  To access individual characters: name.charAt[i]  char nameChArray[] = "Telly Phone".toCharArray();  nameChArray[6] = 'G';  nameChArray[7] = 'r';  nameChArray[8] = 'a';  nameChArray[9] = 'p';  nameChArray[10] = 'h';  To access current length: nameChArray.length  To access individual characters: nameChArray[i]  Note: the variable, name, will be in automatic memory. The string object will be in heap memory, and the value itself will be in either the heap or literal string pool.  Python - unbounded-length, immutable has size and location;  name = "Faye";  name = name + " King";  To access current length: len(name)  To access individual characters: name[i] |
| **Data Representations - Homogeneous Arrays**  All the elements in homogeneous arrays have the same structure and data type.   * Array elements are referenced by subscripts which vary from some lower bound to an upper bound.   1. **Fixed declared length** - size, data type, and structure are known. Location may be static or automatic.   * Compiler knows everything * Runtime descriptors are not necessary   2. **Dynamic declared length** - data type and structure are known at compile time; size is provided at runtime | **Example #5: homogeneous arrays**  FORTRAN, COBOL - only support fixed declared length  int scoreM[50]; // C  DECLARE TEMPERATURE(30:130) FIXED BIN(31); /\* PL/I \*/  PL/I supports fixed declared length and dynamic declared length; it also checks bounds  DECLARE SCORE(NUM\_STUDENTS) FIXED BIN(31); /\* PL/I bounds from 1:NUM\_STUDENTS \*/  Runtime descriptors are necessary:  location of SCORE(I) = origin + (I-LB1)\*element size  Non-test question: why does PL/I use parentheses instead of square brackets when referencing an array element?  C (with C99) supports dynamic declared length  int scoreM[strlen(szValue)];  Java Array supports dynamic declared length  double [] temperatureM = {65.8, 72.9, 85.0, 99.8};  double [] temperatureM = new double[n]; // n element array  The variable temperatureM (in both cases) is in automatic memory containing a pointer to the Array object which is in the heap. The Array object contains its length (i.e., its declared length) and the array. |
| 3. **Unbounded length** - size grows, but data type and structure are known at compile time   * Approach #1: contiguous memory with resize * Approach #2: segmented   Compare these two implementations of unbounded length. | **Example #6: unbounded homogeneous arrays**  **Approach #1: contiguous memory with resize**    When a 6th element is added, allocate new contiguous memory, copy the data, and free the old array    **Approach #2: segmented**    When a 6th element is added, allocate another segment.    Java ArrayList uses contiguous memory and keeps tracks of the capacity and current number of elements. |
| **Multidimensional Arrays**  **1. Contiguous**   * Stored in either row-major or column-major order   + row-major - column varies within row   + column-major - row varies within column * Not used for unbounded length   **How to reference an element in two-dimensional arrays**  Given array A having   * β as its beginning address * element size of E * R rows * C columns   How do you find the address of element A[i][j]?  β + rowSize\*(i-LB1) + E\*(j-LB2)  β + C\*E\*(i-LB1) + E\*(j-LB2) | **Example #7: multidimensional arrays**  C uses row-major order:  int A[4][3];  Shown as two-dimensional:   |  |  |  |  | | --- | --- | --- | --- | |  | Column 0 | Column 1 | Column 2 | | Row 0 | 0 | 1 | 2 | | Row 1 | 10 | 11 | 12 | | Row 2 | 20 | 21 | 22 | | Row 3 | 30 | 31 | 32 |   Shown contiguous in memory:   |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 | | [0][0] | [0][1] | [0][2] | [1][0] | [1][1] | [1][2] | [2][0] | [2][1] | [2][2] | [3][0] | [3][1] | [3][2] | | first row  row 0 | | | second row  (row 1) | | | third row  (row 2) | | | fourth row  (row 3) | | |   Why does C use 0 for lower bounds? |
| **2. Arrays of arrays**   * can be used for bounded or unbounded * allows the sizes of embedded arrays to vary (i.e., jagged) | **Example #8: Arrays of arrays** |
| **Data Representations - Heterogeneous Arrays**  The elements in heterogeneous arrays may have a different structure and/or data type.   * Array elements are referenced by integer subscripts which vary from some lower bound to an upper bound. * Data types for values can vary * Many uses of heterogeneous arrays are similar to using record structures. * Another representation is an array of objects where the objects are of different types (although they probably inherit from the same super class) | **Example #9: heterogeneous arrays**  FORTRAN, C, COBOL, and PL/I do not provide heterogeneous arrays. C, COBOL, and PL/I do provide record structures.  Python list:  gradeList = ['OS', 85, 'Arch', 80, 'PL', 95]; # list of 6 items  print ("For course " + gradeList[0] + " your grade is ", gradeList[1])  Object-oriented:  Suppose we have multiple classes of documents (e.g. SpreadSheet, Word, PowerPoint, Picture) and each inherits from Document. We can create an array of documents where each element can be of a different subclass of Document. This array represents various types of documents associated with a house fire. |
| **Data Representations - Associative Arrays**  The elements are referenced by a key instead of an integer subscript. Sometimes these are known as name-value-pair arrays. In many implementations, the key can be any type of object.  A common implementation is a **hash table**.  C++ provides classes to simplify storing lists of keys and values where the key doesn't have to be an integer.  std::**unordered\_map** hashes the key as the index into a hash table, storing the corresponding value  std::**map** uses a tree to represent the keys and values  **unordered\_map** should be faster for insertion and retrieval, but the keys are not ordered.  When declaring an unordered\_map, we can specify the datatype of the key and corresponding value.  Array syntax can be used to store and access values:  *mapInstance[key*] = *value*; // storing  The **for** statement provides an **auto**matic iterator to access the key and values. | **Example #10: associative arrays in C++**  C++ Associative Array example using an **unordered\_map**:  #include <string>  #include <unordered\_map>  using namespace std;  // Defaults hash function and initial hash table number of buckets  typedef std::unordered\_map<std::string, std::string> ContactHT;  int main(int argc, char \* argv[])  {  ContactHT contactM;  contactM["Board, Bill"] = "830-222-2222"; // new element inserted  contactM["Board, Peg"] = "830-222-3333"; // new element inserted  contactM["Board, Emory"] = contactM["Board, Peg"]; // existing element read,  // new element inserted  contactM["Barr, Ted E"] = "210-555-1111"; // new element inserted  contactM["Barr, Sal Ed"] = contactM["Barr, Candy"]; // referenced missing element  // output the contents of the unordered\_map  for (auto contact : contactM)  {  printf("%s:'%s'\n", contact.first.c\_str(), contact.second.c\_str());  }  }  **Output:**  Barr, Ted E:'210-555-1111'  Board, Bill:'830-222-2222'  Barr, Candy:''  Board, Peg:'830-222-3333'  Board, Emory:'830-222-3333'  Barr, Sal Ed:'' |
| **Associative Arrays in Java - HashMap**  Java provides HashMap for storing hashed keys and values. (HashTable class is provided when needed in a multi-threaded application.)  When declaring a HashMap, we can specify the datatype of the key and corresponding value.  **put**(*key*, *value*) and **get**(*key*) can be used to store and access values.  The **for** statement is used to iterate over the entrySet() to access the key and value pairs. | **Example #11: associative arrays in Java**  Java Associative Array example using a **HashMap**:  import java.util.\*;  public class ContactsMap  {  public static void main(String[ ] args)  {  HashMap <String, String> contactMap = new HashMap<String, String>();  contactMap.put("Board, Bill", "830-222-2222"); // new element inserted  contactMap.put("Board, Peg", "830-222-3333"); // new element inserted  contactMap.put("Board, Emory"  , contactMap.get("Board, Peg")); // existing element read,  // new element inserted  contactMap.put("Barr, Ted E", "210-555-1111"); // new element inserted  contactMap.put("Barr, Sal Ed"  , contactMap.get("Barr, Candy")); // referenced missing element  // output the contents of the HashMap  for (Map.Entry<String, String> entry : contactMap.entrySet())  System.out.println(entry.getKey() + ":" + entry.getValue());  }  }  **Output:**  Barr, Ted E:210-555-1111  Board, Peg:830-222-3333  Board, Bill:830-222-2222  Board, Emory:830-222-3333  Barr, Sal Ed:null |
| **Associative Arrays in Python - Dictionary**  Python's dictionary is an associative array. It is created by assigning a dictionary to a variable. Dictionaries are syntactically name-value pairs:   * surround the dictionary in {} * separate key from value using a colon * separate different dictionary entries using commas   The **for in** statement is used to iterate over the items() to access the key and value pairs. | **Example #12: associative arrays in Python**  Python Associative Array example using a **Dictionary**:  contactM = {"Board, Bill": "830-222-2222"  , "Board, Peg": "830-222-3333"  , "Barr, Ted E": "210-555-1111"}  contactM["Board, Emory"] = contactM["Board, Peg"]; #copy Peg's phone to Emory  for name, phone in contactM.items():  print (name, phone)  **Output:**  Board, Emory 830-222-3333  Board, Bill 830-222-2222  Board, Peg 830-222-3333  Barr, Ted E 210-555-1111 |
| **Data Representations - Record Structure**  Record structures are aggregates of named attributes which are accessed based on an offset from the beginning of the structure.   * Each attribute can have a different data type, size, and structure. * Record structures can contain other record structures   Record structures originated in COBOL, and they are also provided in PL/I and C. In PL/I and COBOL, structures use outline numbering; whereas, C uses braces and nesting of struct statements.  **Referencing Attributes in C**  C uses dot notation, listing from outer most to inner most:  employee.w2.cFillingStatus | **Example #13: C record structure**  C struct example:  struct Employee  {  char szSSN[10];  char szFullName[41];  double dHourlyRate;  struct  {  int iExemptionCnt; // Number of exemptions  char cFillingStatus; // M - Married, S - Single  // X - married but filling  // as single  double dWithholdExtra; // extra amount to withhold  } w2;  };  struct Employee =  { "123456789", "Anita Break", 10.50, 1, 'S', 0.0};  employee.dHourlyRate \*= 1.05;  employee.w2.dWithholdExtra = 10.0;  printf("Name is %s, hourly rate is %10.2lf, withhold extra is %.2lf\n"  , employee.szFullName, employee.dHourlyRate  , employee.w2.dWithholdExtra); |
| To reference an attribute, PL/I uses dot notation, listing from outer most to inner most:  EMPLOYEE.W2.FILLING\_STATUS | **Example #14: PL/I record structure**  PL/I record structure example:  DCL 01 EMPLOYEE,  02 SSN CHAR(9) INIT('123456789'),  02 FULL\_NAME CHAR(40) VARYING INIT('Anita Break'),  02 HOURLY\_RATE FIXED DEC(15,2) INIT(10.5),  02 W2,  03 EXEMPTION\_CNT FIXED BIN(15) INIT(1),  03 FILLING\_STATUS CHAR(1) INIT('S'),  03 WITHHOLD\_EXTRA FIXED DEC(7,2) INIT(0.0);  EMPLOYEE.HOURLY\_RATE = EMPLOYEE.HOURLY\_RATE \* 1.05;  EMPLOYEE.W2.WITHHOLD\_EXTRA = 10.0;  PUT SKIP EDIT("Name is ", EMPLOYEE.FULL\_NAME  , ", hourly rate is ", EMPLOYEE.HOURLY\_RATE  , ", withhold extra is ", EMPLOYEE.W2.WITHHOLD\_EXTRA)  (A, A, A, F(10,2), A, F(7,2)); |
| To reference an attribute, COBOL uses "of", listing from inner most to outer most:  FILLING\_STATUS OF W2 OF EMPLOYEE | **Example #15: COBOL record structure**  COBOL record structure example:  01 EMPLOYEE.  03 SSN PIC 9(9) VALUE "123456789".  03 FULL-NAME PIC X(40) VALUE "Anita Break'.  03 HOURLY-RATE PIC S9(13)V99 USAGE IS COMP-3 VALUE 10.5.  03 W2.  05 EXEMPTION-CNT PIC S9(5) USAGE IS COMP VALUE 1.  05 FILLING-STATUS PIC X VALUE "S".  05 WITHHOLD-EXTRA PIC S9(5)V99 USAGE IS COMP-3 VALUE 0.0.  MULTIPLY HOURLY-RATE OF EMPLOYEE BY 1.05 GIVING HOURLY-RATE OF EMPLOYEE.  SET WITHHOLD-EXTRA OF W2 OF EMPLOYEE TO 10.0. |
| **Structures - Address Calculation**  Structures can contain many attributes. Each attribute has an offset from the beginning of the structure.  Consider the Student and StudentData typedefs:  typedef struct  {  char szName[18]; //0  char szStudentId[7];//18  below 18 + 7 is 25 not div by  4 so we need to add slack bytes  28 is next closest that is div  By 4  int iExam1; //28  int iExam2;//32  int iFinalExam;//36  } Student;//total size is 40bytes  typedef struct  {  int iStudentCnt;//0  Student studentM[20];//4  } StudentData;//total size is  **Variable declaration:**  StudentData data; | **Example #16: Address calculations in record structures**  Assumptions:   * int is 4 bytes, float is 4 bytes, double is 8 bytes * int and float must be aligned to addresses which are multiples of 4 * double must be aligned to addresses which are multiples of 8 * slack bytes might be necessary to align the values   Questions:  1. How many slack bytes must be added after szStudentId to align iExam1 on an address that is a multiple of 4?  ??  2. What is the size of Student?  ??  3. What is the offset to iStudentCnt in StudentData?  ??  4. What is the offset to reach array studentM from the beginning of StudentData?  ??  5. What is the offset for each of these attributes in Student?   * szName - ?? * szStudentId - ?? * iExam1 - ?? * iExam2 - ?? * iExam3 - ??   6. How big is each element of the studentM array?  ??40  7. Show a formula to access data.studentM[i].iExam1.  ??address of data + 4(this is studentM Offset) + i(sizeof(student)) + offset to IExam1  Addr of data + sizeof(int) + (i\*sizeof(Student) + 28)   1. Show a formula to access data.studentM[i].szStudentIdj];   Addr of data + offset to studentM + size of student \* I + offiset to studentID + size of char \* j  ??addr of data + 4 + I \* 40 + 18 + j \* 1 |
| **Data Representations - Unions and Redefines**  Unions allow the same memory to have different type values at different times during execution. **This conserves memory.**  COBOL uses REDEFINES to map the same memory to different datatypes. PL/I has a similar capability. C uses unions.  In C, suppose we want to have a structure that sometimes has integers and sometimes has floats.  union  {  int iValue;  float fValue;  } myFirstValue;  The C compiler determines the type of value being stored based on whether you reference iValue or fValue.  myFirstValue.iValue = 10;  myFirstValue.fValue = 24.99;  If we want to know the type of data in that union, we could add another variable  typedef struct  {  int bFloatInd;  union  {  int iValue;  float fValue;  } value;  } MyValueType;  MyValueType myValue;  myValue.bFloatInd = FALSE;  myValue.value.iValue = 100;  myValue.bFloatInd = TRUE;  myValue.value.fValue = 19.99; | **Example #17: unions in C**  // During translation, we need information about each symbol (aka, token):  // - Token Category:  // Operator (e.g., +, -, \*, /, = , <, >)  // Separator (e.g., comma, semicolon, dot)  // Identifier (e.g., variable name, constant)  // - Token Subcategory:  // For identifiers:  // variable  // integer constant  // float constant  // character constant  // string constant  // - Symbol  // - Value: varies depending on the Token Subcategory  typedef struct  {  int iCategory; // CAT\_OPERATOR, CAT\_SEPARATOR, CAT\_IDENTIFIER  int iSubCategory; // SUB\_VARIABLE, SUB\_INT, SUB\_FLOAT, SUB\_CHAR, SUB\_STRING  char szSymbol[MAX\_TOKEN\_SZ+1];  union  {  int iValue;  float fValue;  char cValue;  char szValue[MAX\_STRING\_SZ+1];  } value;  } Token;  Token tokenM[10];  tokenM[0].iCategory = CAT\_IDENTIFIER;  tokenM[0].iSubCategory = SUB\_INT;  tokenM[0].value.iValue = 10;  tokenM[1].iCategory = CAT\_IDENTIFIER;  tokenM[1].iSubCategory = SUB\_STRING;  strcpy(tokenM[1].value.szValue, "HELLO");  printf("integer is %d, string is %s\n", tokenM[0].value.iValue  ,tokenM[1].value.szValue); |
|  |  |

**©2018 Larry W. Clark,** UTSA CS students may make copies for their personal use