**Storage Management**

|  |  |
| --- | --- |
| **Storage Classifications**  **static -** allocated on program load  **automatic** - allocated on function/block entry   * uses runtime memory stack   **dynamic -**  allocated when needed   * uses heap memory |  |
| **What needs memory?**  There are many runtime components that require memory:  **System Information**   * **System routines**    + library functions   + virtual machine software * **Referencing Environments**  - scope information for understanding what is active * **I/O buffers and DBMS resources**   + buffers for physical blocks of data   + buffers for stream I/O   + concurrency controls (e.g., locks) * **O.S. resources**    + processes   + threads   + pipes   + user information   + … (CS3733) * **Heap memory**    + used for dynamic memory allocation either by the system or the program   + includes data for managing the heap * **Runtime memory stack** (aka, call stack)   + used for automatic memory allocation   + each activation of a function can include use of automatic memory for locals and parameters   + includes function return information   + includes saving of hardware register values from calling functions which will be restored on return to the calling function * **Temporaries**   + expression intermediates   + cursors | **Example #1 Runtime Memory Stack**  The runtime memory stack (aka, call stack) contains an **activation record** for each executed call. On return from the function, the activation record is popped. A function that is invoked recursively can have multiple activation records on the stack. Each activation record may contain:   * Parameters * Return address to the caller * Automatic local variables * Saved hardware register values so that the caller can be restored |
| **User Information**   * **Runtime user programs**   + compiled to machine code instructions   + translated to virtual machine instructions   + source code for interpreting * **Program-defined Static Memory**    + program globals   + retained variables   + program constants * **Program-defined**    + program variables which might be placed in runtime memory stack or static memory; some languages (e.g., LISP) also place variables in heap memory   + program data structures which might be placed in runtime memory stack, static memory or heap memory | **Example #2: C's use of different storage classifications**  In C, your **variables** are in either static memory or automatic memory (locals or parameters) in the runtime memory stack. Pointer variables **can reference** memory that is static, automatic or dynamic in the heap.  void func(int x)  {  char szBuffer[100];  char \*pSzBuffer;  static int iDaysPerMonth[] =  { 0, 31, 29, 31  , 30, 31, 30  , 31, 31, 30  , 31, 30, 31 };  int \*piMonth;  TreeNode \*pRoot = NULL;  pszBuffer = &szBuffer[0];  piMonth = &iDaysPerMonth[1];  pRoot = malloc(sizeof(TreeNode));  }  Notes:   * x is a variable in automatic memory * iDaysPerMonth is a variable in static memory (it is retained) * the other variables are in automatic memory * pszBuffer points to data in automatic memory * piMonth points to data in static memory * pRoot points to data in heap memory   In C, where are the actual pointer variables located?  ?? |
|  |  |
| **Dynamic Allocation**  Most programming languages today provide a mechanism to dynamically allocate memory.  **Explicit Dynamic Memory Allocation**  Programmer explicitly requests the allocation of memory. This can be done by explicitly requesting memory from a memory manager (e.g., malloc(*sizeInBytes*) in C) or it can be done to dynamically create an object. | **Example #3: Explicit Dynamic Memory Allocation in multiple languages**  C:  struct Student \*pStudent = malloc(sizeof(Student));   * explicitly requests memory from C's heap memory manager * **malloc** returns a pointer to the allocated memory * allocates the specified bytes (which is the sizeof a Student) plus some additional bytes * pStudent is a variable in either static global memory or automatic memory   PL/I:  DCL P POINTER,  1 STUDENT BASED,  2 STUDENT\_ID CHAR(6), …  ALLOCATE STUDENT SET(P);   * explicitly requests memory from PL/I's heap memory manager * The ALLOCATE statement allocates dynamic memory and sets pointer P. * P is a pointer variable which is in automatic memory.   C++:  class Student  {  std::string id;  …  public:  Student()  {  id = NULL;  }  }  Student student;  Student \*pStudent = new Student();   * The variable student is allocated as either a static or an automatic. That declaration did not allocate heap memory. The variable student actually contains the Student object. * The variable pStudent is a pointer variable in either static global memory or automatic memory. pStudent references a Student instance which is in heap memory. * new Student() causes the Student() constructor to be invoked and it receives memory from C++'s heap memory manager. The code sets its instance variable, id, to NULL. |
| **Implicit Dynamic Memory Allocation**  Many languages support the allocation of dynamic memory when assignments are done to variables.  Python dynamically allocates memory because of assignments. This includes the creation of the variable.  Strings in C++ and Java have objects which point to the location of the string value which is implicitly allocated. | **Example #4: Implicit dynamic memory allocation**  Python Example:  movieList = ["Holy Grail", "Life of Brian", "The Meaning of Life"];  Most implementations of Python also place the variables in heap memory. How can they be referenced?  ?? |
| Many implementations of **unbounded arrays** will implicitly allocate additional memory when new elements are added. If using contiguous, the algorithm might double the size of the array. If using segmented, a new segment is added. | **Example #5: Implicit dynamic memory allocation using continguous memory reallocation**  In this example, we initially have allocated 5 elements (size) and have populated 5 elements.    When a 6th element is added, allocate new contiguous memory, copy the data, and free the old array |
| **Storage Management Phases**  1. **Allocation** - explicit or implicit  2. **Recovery** - recover unused storage that was previously allocated  3. **Compaction** - construct large blocks of free storage by shifting free items to one end  We will examine how each type of memory handles those storage management phases. |  |
| **Static Memory**   * Static memory is allocated on program load and freed with the program exits. * Does not require runtime storage management software   Languages:  C global extern and static  C++  PL/I static  COBOL  FORTRAN | Adv:   * storage management not required at runtime * addresses are bound to variables at compile/load; therefore, more efficient code   Disadv:   * cannot have multiple copies of data for recursion * the size of data cannot vary past the initial allocation   Variables declared outside of functions in C are globals. C has globals which are static, extern basis or extern references. **C static** means the variables are *private* to the file. **extern references** use the extern keyword. Global variables coded without static and without extern are **global extern basis** variables. What is the purpose of distinguishing global extern reference vs global extern basis?  ??  C also has local static variables. A better term would have been retained. Unlike automatic local variables which are initialized on each entry of the function/block, retained variables are only initialized once. They retain the value they had from the last call and can then be changed. |
| **Automatic Memory**   * Allocated from the runtime memory stack (at the *top*) * Recovery from the top of the stack * Commonly used for locals and parameters in languages that support recursion * Allocated on function/block entry and freed on function/block exit   Languages:  C automatics  C++ automatics  PL/I automatics  Java method locals, method parameters  … | Adv:   * Size of data may vary (note that C initially did not take advantage of this) * Storage management fairly easy to implement using a stack   + allocation - move the stack pointer   + recovery - move the stack pointer the opposite direction   + compaction - not necessary   Disadv:   * Must free in a particular order (reverse of allocation). If those variables are pointing to dynamic memory (in a language which uses explicit dynamic memory mgt), the reference could be lost (causing a leak). * Addresses are bound at runtime as offsets within an activation record. |
| **Exercise #1 Storage Classifications in C**  Consider the code for files exAc. and exB.c.  /\* file exB.c \*/  extern int iMyVar;  static int iMyVar2;  void bfunc()  {  int iHey;  /\* statements \*/  } | /\* file exA.c \*/  int iMyVar;  int iMyVar2;  void myFunc (int iParm)  {  int iHello = 10;  static int iElectricity = 100;  /\* statements \*/  }  int anotherFunc ()  {  int iHey;  static int iKite;  /\* statements \*/  } |
| What are the automatic locals or parameters in exB.c?  ??  What are the global variables in exB.c? Which are C static, extern reference vs extern basis?  ??  What are the static locals in exB.c?  ?? | What are the automatic locals or parameters in exA.c?  ??  What are the global variables in exA.c? Which are C static, extern reference vs extern basis?  ??  What are the static locals in exB.c?  ??  Is iHello re-initialized to 10 on every call of myFunc?  ??  Is iElectricity re-initialized to 100 on every call of myFunc?  ?? |
| **Dynamic Memory**   * Allocated on explicit or implicit request * Freed by either explicit freeing or system recovery * Memory management made more complicated due to the different sizes of the allocated items   Languages:  C malloc  C++ malloc and objects  PL/I based using ALLOCATE  Java  Python  LISP | Adv:   * Size of data may vary and can be unbounded * Can free when necessary   Disadv:   * Storage management is more complex   + storage management software is required   + handle different sizes of allocated items   + possibly handle recognition of recovery * Explicit freeing can cause dangling references - program frees something, but there are still possible access paths to the data * When explicit freeing are expected, unusable storage can happen when the program removes all access paths without freeing the item * Addresses are bound at runtime via pointers; less efficient code |
| **Different-sized Items**  If the items being allocated and recovered are of different sizes, storage management is impacted.   * Need to know the size of an item when it is allocated and freed. * Adjacent freed items need to be combined * Fragmentation can occur | See the PowerPoint presentation for an example of the C approach. |
| **Recovery**  Two categories of recovery:  **explicit return** programmer returns unused storage   * system can mark it as free or include it on the free list   **system recovery** system recovers items that can't be accessed by the program. Techniques:   * reference counts * garbage collection * combination of reference counts and garbage collection |  |
| **Reference Counts**  This technique for automatically recovering data items includes a Reference Count in allocated items.  typedef struct  {  short shAllocSz; // Allocated size in bytes  // includes memory for  // shRefCount, shAllocSz,  // and shNodeType  short shRefCount; // Reference count, a  // negative value indicates  // free  short shNodeType; // Type of data in sbData  char sbData[MAX\_DATA\_SZ]; // The allocated  // data can be binary. The  // actual size is  // shAllocSz minus 6  } AllocNode;  In the example on the right, we didn't show the shNodeType. | **Example #6: Singly Linked List with variable references**    Programmer who allocated nodes for his/her linked list used something like this for the structure:  typedef struct Node  {  int iGrade;  struct Node \*pNext;  } Node;  Let's assume that Node structure is 8 bytes (4 byte int, 4 byte pointer). Using our internal AllocNode structure:   * sbData would contain that Node structure. * internal overhead would be sizeof the length attribute plus the size of the reference count plus the sizeof the node type (assuming each of those are short, this is 6 bytes of overhead) * shAllocSz would be 14 bytes (size of sbData + 6)   Each time we add a new reference, the reference count is increased. When we remove a reference, the reference count is decreased. If it reaches 0, the item can be recovered.  Suppose we see this assignment in a program:  p = gradeList;  What happens?   1. If p is non-NULL before the assignment:    1. Access the item pointed to by p.    2. Decrement that item's reference count.    3. If that reference count reaches 0, recover the item. 2. Copy the pointer from gradeList to p. 3. If p is now non-NULL:    1. Access the item pointed to by p.    2. Increment that item's reference count. |
| When a node's reference count reaches zero what do we do?  ?? | **Example #7: Removing the last reference.**  Suppose a node containing a pointer has a ref count that reaches zero. If variable **x** no longer points to the node containing the grade 90, its reference count decreases to 0. What do we do?    Since the reference count reaches zero, what it references directly will also be decremented.    Notice that this didn't impact the node containing 80 since it is still referenced by the node containing 100. |
| Although not shown in the diagrams for example #7, why does the AllocNode contain shNodeType?  ?? |  |
| **Exercise #2: Ref Count** | See StgMgtRefCnt pptx |
| **Reference Counts Gone Bad**  If we only rely on reference counts, we can't always reach an shRefCount of zero when all program references are removed. | **Example #8: Rec Count with a circular list**  Example of a circular list:    When we remove the reference by cirList, the first item's reference count goes to 1 and the memory becomes unusable. |
| **Garbage Collection**  When free space is exhausted, recover any unreferencable free space.   * The system must be capable of determining which storage elements are accessible by the programmer. In many languages, this is the set of locals, parameters, and globals. * Each storage item must have a Garbage Collection Bit. | **Garbage Collection Algorithm:**   1. **Mark Subphase**. Sequentially traverse through all heap memory marking the Garbage Collection Bit ON. 2. **Follow Subphase.** For each variable referencable by the programmer, follow pointers to each referencable storage item and turn the bit OFF. 3. **Collection Subphase.** Sequentially traverse through all heap memory. If a storage item's Garbage Collection Bit is still ON, recover it. |
| **Garbage Collection Follow Subphase**  We must know which variables are possibly referencable by the programmer. For most languages, this subphase must be able to reference all global and automatic variables. If those variables reference the heap, we must follow them.  If a variable references heap memory, we must also know whether that memory references additional items in the heap. This requires that we understand the contents of each *storage item* allocated in the heap. At runtime, we need metadata (i.e., data about the data) that specifies any embedded pointers to additional *storage items.* | **Example #9: Metadata for Garbage Collection**  For a storage item, we must understand each of its attributes:    Customer Metadata   |  |  |  |  | | --- | --- | --- | --- | | AttrNm | Type | Offset | Size | | customerId | string | 0 | 8 | | name | string | 8 | 16 | | balance | double | 24 | 8 | | pFirstItem | pointer | 32 | 8 | | pNextCust | pointer | 40 | 8 |   PO Line Item Metadata   |  |  |  |  | | --- | --- | --- | --- | | AttrNm | Type | Offset | Size | | productId | string | 0 | 10 | | qtyReq | integer | 12 | 4 | | unitCost | double | 16 | 8 | | pNextItem | pointer | 24 | 8 | |
| **See the GC ppt for an example of all the subphases.** |  |
| **Compaction**  Due to fragmentation, it may be necessary to compact memory, causing free space to be contiguous at one end of the heap. | Since allocated items would be moved, what also has to change? |
|  |  |

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