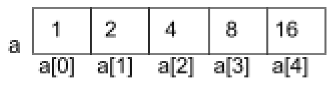
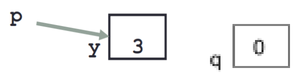
Name: ______________________________ Perm no. :___________________________

CMPSC16 Midterm-II Exam Notes

Arrays		Pointers	

Declaration	`int a[5];` // 'a' is an array of integers with 5 elements	**Declaration**	`int *mydata1, *mydata2;`
			`Node *mynode;`
			`int **px;`

Declaration and initialization	`int a[5]={1,2,4,8,16};`	**Declaration and initialization**	`int y, *p =&y;`**//p points to y** `int *q =0;` **// p is initialized to NULL** `Node *mynode = new Node;`
Representation	 a 1 2 4 8 16 a[0] a[1] a[2] a[3] a[4]	**Representation**	
Size	5 integer elements or 20 bytes	**Size of p or q or mynode**	4 bytes

Printing all elements	`for(int i =0;i<5;i++)`
	`    cout<<a[i];`
		Printing	`cout<<p;`//prints the address of y
			`cout<<q;` //prints 0

Declaration of functions that take arrays as arguments	int sum(int arr[ ], int len)
	int sum (int *arr , int len)
		Dereferencing	cout<<*p; //Prints 3
			cout<<*q; //Segfaults q is NULL
			cout<<mynode->data;
			//Prints the data element of Node pointed to by mynode

Passing an array to a function	`int mysum;`
	`mysum = sum(a, 5);`
	When you pass an array as a parameter to a function, it decays to a pointer to the first element of the array.
		Ways to change value of variable: (1) Directly using the name of the variable
		(2) Indirectly via the pointer
			(1) Directly
			`int * q, x = 10;`
			`q = &x;`
			`x = 10;`
			(2) Indirectly: via pointer
			`int * q, x = 10;`
			`q = &x;`
			`*q = 10;`

References	
Definition	A reference variable is an "alias for another variable."

Uses of the Ampersand (&)	(1) "address of": the pointer obtains the ADDRESS of the int x, **not** the value of x (5)
	`int x = 5;`
	`int *p = &x;`
	(2) "declaring a reference": y is another name for the variable x
	`int x = 5;`
	`int& y = x;`
	When dealing with reference variables, the **& is used in the second way.**
	It does not mean "get the address of"

Name: ______________________________ Perm no. :___________________________

CMPSC16 Midterm-II Exam Notes

Call by Reference vs. Call by Value
 	
Call by Reference vs. Call by Value
	In a **call by value** the parameter that is passed into a function is a COPY of the actual parameter that is passed in. E.g.
	`void func(int a) {`
	`      a += a;`
	`}`
	`int main() {`
	`  int x = 5;`
	`  cout <<”x =” <<  x << endl;`
	`  func(x);`
	`  cout << “x =” << x < endl;`
	`}`
	will output:
	x =5;
	x =5;
	
	The value of x does not change, since when x is passed to func in the third line of main, a COPY of x is passed, not the actual variable x.
		On the other hand, you can **call by reference** by changing the parameter of the functions:
		`void func(int &a) {`
		`      a += a;`
		`}`
		`int main() {`
		`  int x = 5;`
		`  cout <<”x =” <<  x << endl;`
		`  func(x);`
		`  cout << “x =” << x < endl;`
		`}`
		will output:
		x =5;
		x =10;
		
		Call by reference is useful when using large objects, since you can directly modify the object by passing it as a reference instead of a COPY of the object. Any changes you make to the object in the function will change the actual values in the object itself.

Structs		
Definition of struct	A struct is a data structure that contains simpler data types like ints, strings, etc. They are usually used as a way to define your own type.	

Declaring a struct data type	`struct Node{`
	`  int data;`
	`  Node *next;`
	`};`
		`struct LinkedList{`
		`  Node *head;`
		`  Node *tail;`
		`};`

Declaring an object of type struct stack vs. heap	`Node n;                    ` **// n is 8 bytes created on the stack** `Node *p;                   ` **// p is 4 bytes created on the** `stack` `Node *p = new Node;        ` **// p points to a Node on the heap**	

Accessing member variables	`n.data =5;`
	`n.next = NULL:`
	
	`p->data = 10;`
	`p->next =NULL;`
		If n is a struct object or a reference to a struct object use the dot operator.
		
		If you have a pointer to a struct like (p), use the arrow operator to access the member variables

Name: ______________________________ Perm no. :___________________________

CMPSC16 Midterm-II Exam Notes

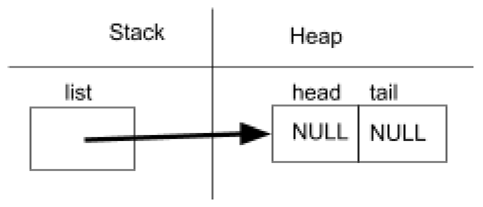
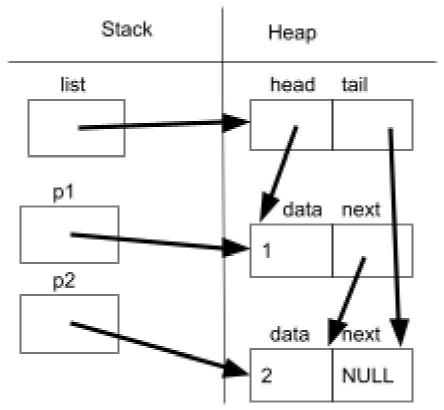
Linked-lists -> Will not be on the exam

Definition of linked-list	Stores a list of elements in Nodes that are not next to each other in memory A Linked List object consists of two Node pointers: one that points to the head (beginning) of the linked list, and one that points to the tail (last node). If the object is empty, both fields are null. If the object has only one Node, both fields point to that same Node. A Node is an object that can have any type of data fields, and a Node pointer field (that points to the next Node in that list).	

Initialization	`LinkedList *list = new LinkedList;`
	`list->head=list->tail = null;`	

Iterating through a Linked List	`Node *p = list->head;`
	`while(p != NULL){`
	`  p = p->next;`
	`  //you can use p to manipulate the data in every node`
	`  //by accessing p->data`
	`}`	

Deleting a Linked List	1. Delete every node
	` Node *p, *q;`
	` p= list->head;`
	` while(p){`
	`      q = p;`
	`      p = p->next;`
	`      delete q;`
	`   }`
	2. If it exists, delete the pointer that points to that list.
	` delete list;`	

Stack vs. Heap	`LinkedList *list = new`
	`LinkedList;`
	`list->head = list->tail = NULL;`
	
	`Node *p1 = new Node;`
	`p1->data = 1;`
	`Node *p2 = new Node;`
	`p2->data = 2;`
	`p1->next = p2;`
	`p2->next = NULL;`
	`list->head = p1;`
	`list->tail = p2;`
		
		 Stack
		Heap
		list
		head
		tail
		p1
		p2
		data
		next
		1
		data
		next
		2
		NULL