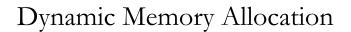
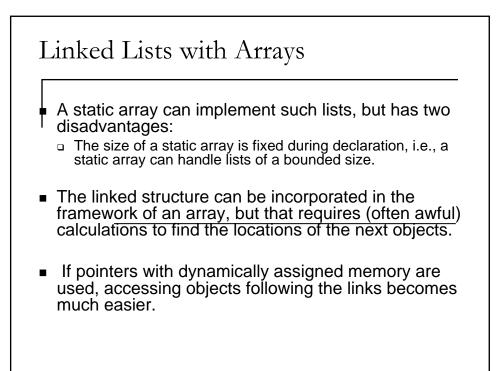
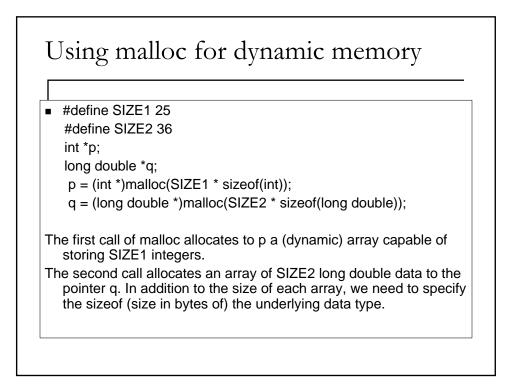
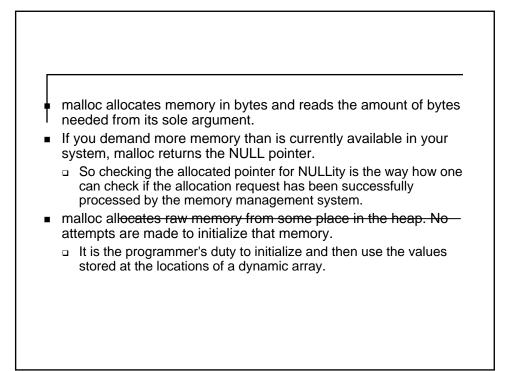
CS11001/CS11002 Programming and Data Structures (PDS) (Theory: 3-1-0)



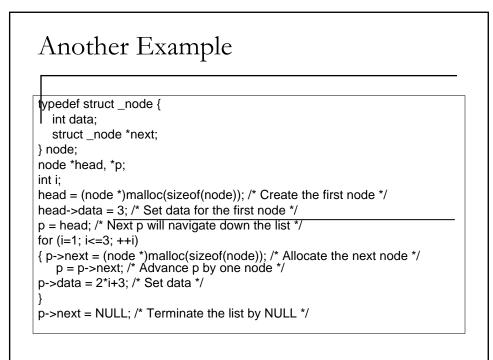
- All variables, arrays, structures and unions that we worked with so far are *statically* allocated, meaning that whenever an appropriate scope is entered (e.g. a function is invoked) an amount of memory dependent on the data types and sizes is allocated from the stack area of the memory.
- When the program goes out of the scope (e.g. when a function returns), this memory is returned back to the stack.
- There is an alternative way of allocating memory, more precisely, from the heap part of the memory.
- In this case, the user makes specific calls to capture some amount of memory and continues to hold that memory unless it is explicitly (i.e., by distinguished calls) returned back to the heap.
- Such memory is said to be *dynamically* allocated.

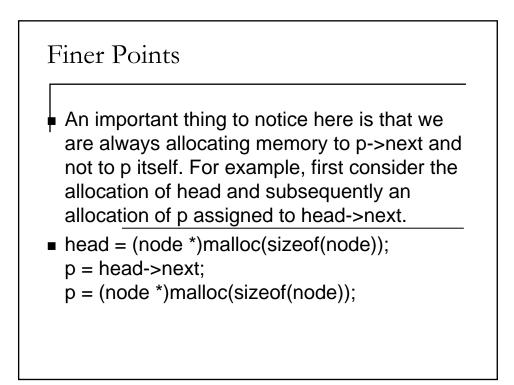


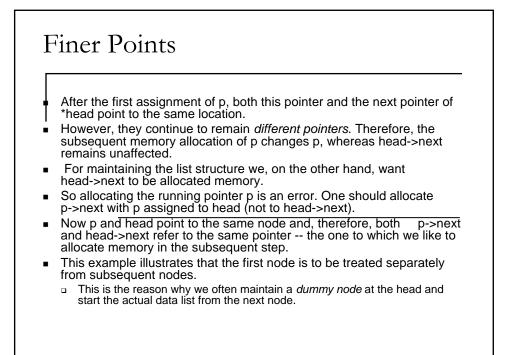


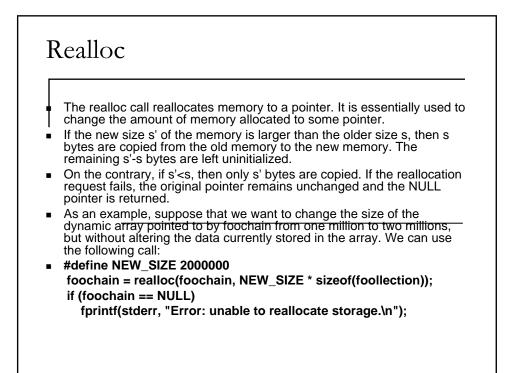


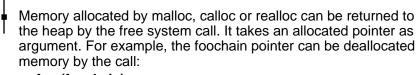
Example
foollection initfc (int type)
<pre>{ foollection fc; /* Set type of the collection */ fc.type = type; /* Allocate memory for the data</pre>
pointer */ if (type == 1)
fc.data = (int *)malloc(10*sizeof(int));
else if $(type == 2)$
fc.data = (int *)malloc(10000000*sizeof(int));
else fc.data = NULL; /* Check for error conditions */ if (fc.data == NULL)
fprintf(stderr, "Error: insufficient memory or unknown type.\n"); return fc; }





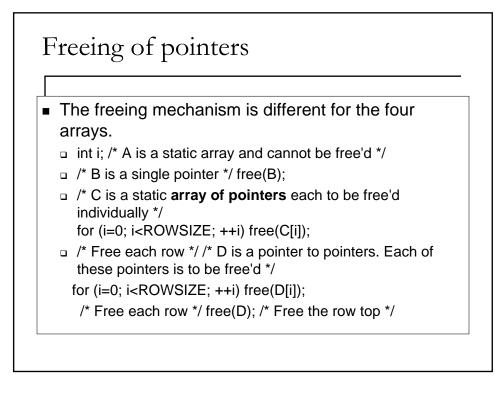






free(foochain);

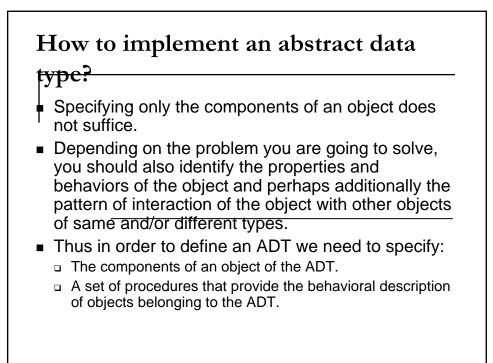
- When a program terminates, all allocated memory (static and dynamic) is returned to the system.
- There is no necessity to free memory explicitly.
- However, since memory is a bounded resource, allocating it several times, say, inside a loop, may eventually let the system run out of memory.
- So it is a good programming practice to free memory that will no longer be used in the program.

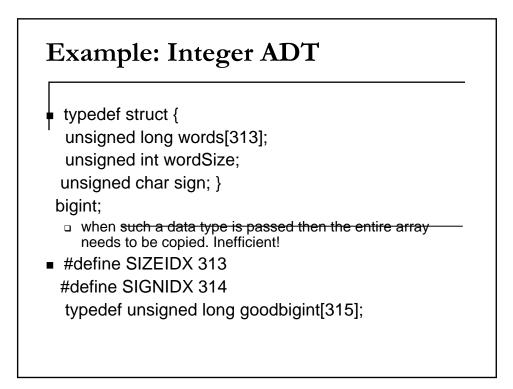


Abstract Data Type (ADT)

What is ADT?

- An abstract data type (ADT) is an object with a generic description independent of implementation details.
- This description includes a specification of the components from which the object is made and also the behavioral details of the object.





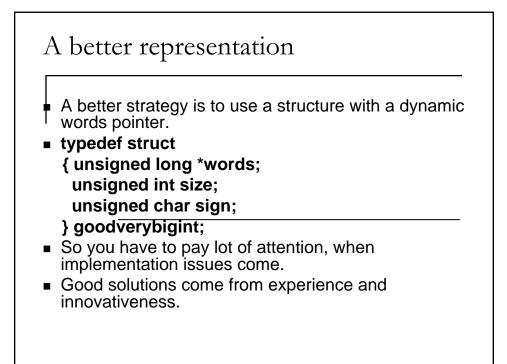
Using the integer data type

#include<stdio.h> #include<time.h>

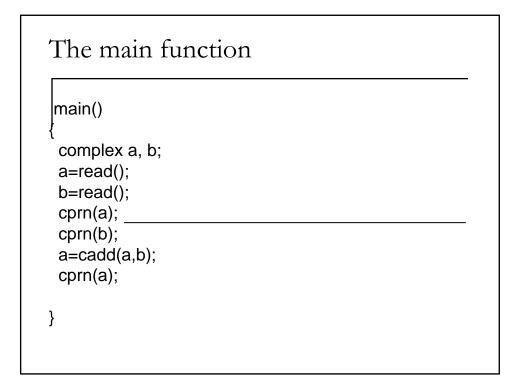
main()
{
typedef unsigned long int goodint[315];
int i;
goodint a;
srand((unsigned int)time(NULL));

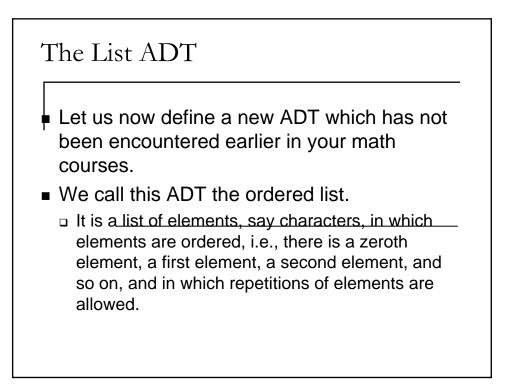
for(i=0;i<315;i++)
a[i]=1+rand()%99;
for(i=0;i<315;i++)
printf("%d ",a[i]);
printf("\n");
}</pre>

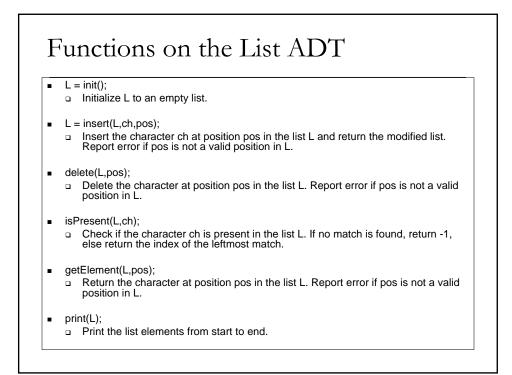




	complex codd(complex z1, complex z2
#include <stdio.h></stdio.h>	complex cadd(complex z1, complex z2
typedef struct{ double real; double imag; }complex;	complex z; z.real=z1.real+z2.real; z.imag=z1.imag+z2.imag; return(z); }
void cprn(complex z)	<pre>complex read() {</pre>
{	complex z;
<pre>printf("(%lf) + i(%lf)\n", z.real,</pre>	scanf("%lf",&z.real);
z.imag);	scanf("%lf",&z.imag);
}	return(z);
z.imag); }	







	olist insert(olist L , char ch , int pos)
#include <stdio.h></stdio.h>	
#define MAXLEN 100	{
	int i;
typedef struct {	
int len;	if ((pos < 0) (pos > L.len)) {
char element[MAXLEN]; } olist;	fprintf(stderr, "insert: Invalid index %d\n",
y olist,	pos);
olist init()	return L;
{	
olist L;	}
L.len = 0;	if (L.len == MAXLEN) {
return L;	fprintf(stderr, "insert: List already full\n");
}	return L;
void print(olist L)	}
	for (i = L.len; i > pos;i) L.element[i] =
int i:	L.element[i-1];
	L.element[pos] = ch;
for(i = 0; i < L.len; ++i) printf("%c", L.element[i]);	
}	++L.len;
	return L;

