CSC 301: Data Structures and Algorithms

Abstract Data Types

Denis L. Nkweteyim

Abstract Data Type (ADT)

- A data type (a set of values and collection of operations on those values) that is accessed ONLY through an interface
- Client
  - Program that accesses ADT
- Implementation
  - Program that specifies the ADT
- Use of ADT
  - Allows us to build programs that use high level abstractions
  - We can separate the conceptual transformations that our programs perform on our data from any particular data structure representation and algorithm implementation
    - Higher level abstract mechanisms expressed in terms of more primitive ones
    - This frees us from detailed concern about how they are implemented
  - Help in large software development since
    - ADTs provide a means to limit the size and complexity of the interface between algorithms and their data structures, and the programs that use the algorithms and data structures

C storage classes

- Every C variable and function has two attributes: type and storage class
- Storage classes
  - automatic, external, register, and static
  - Corresponding keywords: auto, extern, register, and static

Storage class `auto`

- Most common of the storage classes
  - Default for variables declared within function bodies and within compound statements (i.e., statements enclosed within braces {})
  - Keyword `auto` need not be explicitly specified and is usually not specified by programmers
  - If explicitly specified, add keyword `auto` in front of variable declaration
  - Examples: `auto int a, b, c; auto float f;`
- When a block is entered or function called, the system allocates memory for automatic variables
  - The variables are local to the block or function
  - When the block or function is exited, the system releases the memory that was set aside, and so these variables are lost
  - If the block is re-entered, the system once again allocates memory, but previous values are unknown

Storage class `extern`

- External variables
  - Enable information to be transmitted across blocks and functions
  - Variables declared outside a function
    - Have storage class `extern`
    - Have memory permanently assigned
    - Have global scope to all functions and blocks that come after it
    -
    - In the program (next slide) for example, variables a, b, and c are external (i.e., global) variables
- All C functions have storage class external
  - Keyword extern is usually omitted in function declarations and definitions, however

Storage class `extern`

```c
#include <stdio.h>
int a = 1, b = 2, c = 3; /*global variables*/
int addition(void);
int main(void) {
  printf("Before function call\n");
  printf("a=%d b=%d c=%d  ",a, b, c);
  printf("sum = %d\n",addition());
  printf("\nAfter function call\n");
  printf("a=%d b=%d c=%d\n",a, b, c);
}
int addition(void) {
  int b, c; /*local variables*/
  a = b = c = 4;
  return (a + b + c);
}
```

```
Before function call
a=1 b=2 c=3  sum = 12

After function call
a=4 b=2 c=3
```

Storage class `extern`

- Line 2 could have been
  - `extern int a = 1, b = 2, c = 3;`
- Keyword `extern`
  - Tells the compiler to look for the variable elsewhere either in this file or in some other file
  - If the variable is declared in the same file, the keyword extern is optional
  - Program on next slide gives same result as before

Typical use of `extern` keyword

extern2.c

```
#include <stdio.h>
int a2 = 1, b2 = 2, c2 = 3; /*global variables*/
int addition2(void);
int main(void) {
  printf("Before function call\n");
  printf("a=%d b=%d c=%d  ",a2, b2, c2);
  printf("sum = %d\n",addition2());
  printf("\nAfter function call\n");
  printf("a=%d b=%d c=%d\n",a2, b2, c2);
}
```

extern_implementation.c

```
int addition2(void) {
  extern int a2; /*look for a elsewhere*/
  int b2, c2; /*local variables*/
  a2 = b2 = c2 = 4;
  return (a2 + b2 + c2);
}
```

Storage class `register`

- Tells the compiler to store the associated variables in high speed memory registers, if possible
  - Registers are very few and may not be available
    - In such cases, the storage class defaults to `auto`
- Example

```
{
  register int i;
  for (i=0; i<MAX; i++) {
     .....
  }
}  /*block exit will free the register*/
```

Storage class `static`

- Unlike automatic variables variables, static variables retain their values when a block or function is exited

```
#include <stdio.h>
void count(void) {
  static int cnt = 5; printf("%3d",cnt); cnt++;
}
int main(void) {
  int i;
  for (i = 0; i < 10; i++)
    count();
}

  5  6  7  8  9 10 11 12 13 14
```

Stack ADT Example

```
Header File item.h
typedef int Item;
```

```
Interface file stack.h
void stackInit(int);
int stackEmpty();
void stackPush(Item);
Item stackPop(void);
```

- Interface
  - Acts as a contract between the client and the implementation
  - Function declarations in the interface ensure that the calls in the client program and in the implementation match
    - But the interface contains no information about how the functions are implemented
    - Different programmers may implement the functions differently, but the interface remains the same
    - So, the client program(s) can be written independently.

Stack ADT – array implementation

```
Implementaion file stackADTAr.c
#include <stdlib.h>
#include "item.h"
#include "stack.h"
static Item *s;
static int N;
void stackInit(int max) {
  s = malloc(max*sizeof(Item));
  N = 0;
}
int stackEmpty() {
  return N == 0;
}
void stackPush(Item item) {
  s[N++] = item;
}
int stackPop(void) {
  return s[--N];
}
```

Stack ADT – linked list implementation

```
Implementaion file stackADTll.c
#include <stdlib.h>
#include "item.h"
#include "stack.h"
typedef struct stackNode *LINK;
struct stackNode {
  Item item;LINK next;
};
static LINK head;
LINK newNode(Item item, LINK h) {
  LINK nnode = malloc(sizeof(*nnode));
  nnode -> item = item; nnode -> next = h; return nnode; }
void stackInit(int max) { head = NULL; }
int stackEmpty() {return head == NULL;
}
void stackPush(Item item) {head = newNode(item, head);
}
Item stackPop(void) {
  Item item = head -> item; LINK t = head -> next;
  free(head); head = t;return item;
}
```

Queue ADT Example

```
Header File item.h
typedef int Item;
```

```
Interface file queue.h
void queueInit(int);
int queueEmpty();
void enqueue(Item);
Item dequeue(void);
```

Queue ADT – array implementation

Implementaion file queueADTAr.c

```
#include <stdlib.h>
#include "item.h"
#include "queue.h"
static Item *q;
static int N, head, tail;
void queueInit(int max) {
  q = malloc(max+1*sizeof(Item));
  N = max+1; head = tail = 0;
}
int queueEmpty() {
  return head % N == tail;
}
void enqueue(Item item) {
  q[tail++] = item;
  tail = tail % N;
}
int dequeue(void) {
  return q[head++];
  head = head % N;
}
```

Queue ADT – linked list implementation

```
Implementaion file queueADTll.c
#include <stdlib.h>
#include "item.h"
#include "queue.h"
typedef struct queueNode *LINK;
struct queueNode {Item item;LINK next;};
static LINK head,tail;
LINK newNode(Item item, LINK nxt) {
  LINK nnode = malloc(sizeof(*nnode)); nnode -> item = item;
  nnode -> next = nxt; return nnode;
}
void queueInit(int max) { head = NULL;}
int queueEmpty() { return head == NULL;}
void enqueue(Item item) {
  if (head == NULL) {
     head = (tail = newNode(item, head));return;
  }
  tail -> next = newNode(item, tail->next); tail = tail -> next;
}
int dequeue(void) {
  Item item = head -> item; LINK t = head -> next;free(head);
head = t;return item;
}
```

First Class ADTs

- Interfaces and implementations of the stack and queue ADTs we have seen provide client programs with the possibility of using a single instance of a stack or queue
  - e.g., calling the function stackInit() initializes the one stack that our program deals with
  - Such ADTs are widely used, and simple to use because there is only one object of that type in the program
  - But situation is analogous to having a program that deals with only one integer instance, say.
- A first class ADT
  - ADT for which we can have potentially several instances, and which we can assign to variables which we can declare to hold the instances

Complex Number ADT Example 1

```
Interface file complex1.h
typedef struct{float Re;  float Im;
} Complex1;
Complex1 ComplexInit1(float,float);
float Re1(Complex1);float Im1(Complex1);
Complex1 ComplexMult1(Complex1,Complex1);
```

```
Implementation file complex1.c
#include "complex1.h"
Complex1 ComplexInit(float Re, float Im) {
   Complex1 t; t.Re = Re; t.Im = Im;return t;
}
float Re1(Complex1 z) {return z.Re;}
float Im1(Complex1 z) {return z.Im;}
Complex1 ComplexMult(Complex1 a, Complex1 b) {
  Complex1 t;
  t.Re = a.Re * b.Re - a.Im * b.Im;
  t.Im = a.Re * b.Im + a.Im * b.Re;
  return t;
}
```

Complex Number ADT
Client Example 1

- Program determines the N complex roots of unity using the formula

$$\cos\left(\frac{2\pi k}{N}\right)+i\sin\left(\frac{2\pi k}{N}\right),\ for\ k=0\ \text{to}\ N-1$$

- and then multiplies the complex root by itself N times to get unity (i.e., 1.00 + i0.00)

Complex Number ADT Client Example 1

```
Client file croots1.c
#include <stdio.h>
#include <math.h>
#include "complex1.h"
#define PI 3.141592625
int main(int argc, char *argv[]) {
  int i, j, N = atoi(argv[1]); /*N is # of complex roots to find*/
  Complex1 t, x;
  printf("%d Complex roots of unity\n",N);
  for (i = 0; i < N; i++) {
    float r = 2.0 * PI * i/N;
    t = complexInit1((float)cos(r), (float)sin(r));
    printf("%2d %6.3f %6.3f ",i, Re1(t), Im1(t));
    for (x = t, j = 0; j < N - 1; j++)
       x = complexMult1(t, x);
    printf("%d %6.3f %6.3f\n",i, Re1(x), Im1(x));
  }
}
```

Complex Number ADT Client Example 1

```
croots 8
8 Complex roots of unity
 0  1.000  0.000 0  1.000  0.000
 1  0.707  0.707 1  1.000  0.000
 2 -0.000  1.000 2  1.000  0.000
 3 -0.707  0.707 3  1.000  0.000
 4 -1.000 -0.000 4  1.000  0.000
 5 -0.707 -0.707 5  1.000 -0.000
 6  0.000 -1.000 6  1.000  0.000
 7  0.707 -0.707 7  1.000 -0.000
```

```
croots 7
7 Complex roots of unity
 0  1.000  0.000 0  1.000  0.000
 1  0.623  0.782 1  1.000  0.000
 2 -0.223  0.975 2  1.000  0.000
 3 -0.901  0.434 3  1.000 -0.000
 4 -0.901 -0.434 4  1.000  0.000
 5 -0.223 -0.975 5  1.000 -0.000
 6  0.623 -0.782 6  1.000 -0.000
```

```
croots 10
10 Complex roots of unity
 0  1.000  0.000 0  1.000  0.000
 1  0.809  0.588 1  1.000 -0.000
 2  0.309  0.951 2  1.000  0.000
 3 -0.309  0.951 3  1.000 -0.000
 4 -0.809  0.588 4  1.000  0.000
 5 -1.000 -0.000 5  1.000  0.000
 6 -0.809 -0.588 6  1.000 -0.000
 7 -0.309 -0.951 7  1.000 -0.000
 8  0.309 -0.951 8  1.000  0.000
 9  0.809 -0.588 9  1.000 -0.000
```

```
croots 5
5 Complex roots of unity
 0  1.000  0.000 0  1.000  0.000
 1  0.309  0.951 1  1.000  0.000
 2 -0.809  0.588 2  1.000  0.000
 3 -0.809 -0.588 3  1.000 -0.000
 4  0.309 -0.951 4  1.000  0.000
```

Difference with earlier examples

- User-defined type Complex1 created (in interface file)
  - Variables of this type declared
- In the earlier programs, the stack and queue were defined in the implementation with no opportunity to create other instances

Some notes on the interface file used

- Type Complex1 is not an ADT
  - The data type is exposed in the interface
  - For an ADT, the interface should give access to the data type, and not expose it
  - Knowing that complex numbers are defined algebraically, i.e., a struct with two float fields (as seen in the interface)
    - Client programmers can go ahead and use the same algebraic representation for its complex numbers
    - Problem is, if for any reason, the implementation file changes from algebraic to some other representation, say using polar coordinates
    - All the hundreds of existing client programs would need to be updated

Complex Number ADT Example 2

```
Interface file complex2.h
typedef struct complex *Complex2;
Complex2 ComplexInit2(float,float);
float Re2(Complex2);
float Im2(Complex2);
Complex2 ComplexMult2(Complex2,Complex2);
```

```
Implementation file complex2.c
#include "complex2.h"
#include <stdlib.h>
struct complex { float Re; float Im; };
Complex2 ComplexInit2(float Re, float Im) {
  Complex2 t = malloc(sizeof *t); t->Re = Re; t->Im = Im;return t;
}
float Re2(Complex2 z) { return z->Re; }
float Im2(Complex2 z) { return z->Im; }
Complex2 ComplexMult2(Complex2 a, Complex2 b) {
  return ComplexInit2(Re2(a) * Re2(b) - Im2(a) * Im2(b),
                      Re2(a) * Im2(b) + Im2(a) * Re2(b));
}
```

ADT Example 3 – Polynomial Evaluation

- We consider an ADT to evaluate polynomial expressions such as the following

$$\left(1-x+\frac{x^2}{2}-\frac{x^3}{6}\right)\left(1+x+x^2+x^3\right)=1+\frac{x^2}{2}+\frac{x^3}{3}-\frac{2x^4}{3}+\frac{x^5}{3}-\frac{x^6}{6}$$

ADT Example 3 – Polynomial Evaluation

- ADT should be capable of handling multiplication and addition of polynomials
- In the example above, we want to be able to arrive at the following expression

$$\left(1-x+\frac{x^2}{2}-\frac{x^3}{6}\right)\left(1+x+x^2+x^3\right):$$

$$
\begin{array}{l}
1-x+\dfrac{x^2}{2}+\dfrac{x^3}{6} \\
\quad +x-x^2+\dfrac{x^3}{2}\quad -\dfrac{x^4}{6} \\
\qquad +x^2-x^3\quad +\dfrac{x^4}{2}-\dfrac{x^5}{6} \\
\qquad\qquad x^3\quad -x^4+\dfrac{x^5}{2}-\dfrac{x^6}{6} \\
\hline
1\quad +\dfrac{x^2}{2}+\dfrac{x^3}{3}-\dfrac{2\mathrm{x}^4}{3}+\dfrac{x^5}{3}-\dfrac{x^6}{6}
\end{array}
$$

ADT Example 3 – Interface

```
Interface file poly.h
typedef struct poly *Poly;
void showPoly(Poly);
Poly PolyTerm(int, int);
Poly PolyAdd(Poly, Poly);
Poly PolyMult(Poly, Poly);
float PolyEval(Poly, float);
```

ADT Example 3 – Implementation

Implementation file poly.c

```
#include <stdio.h>
#include <stdlib.h>
#include "poly.h"
struct poly {int N;int *a;};
void showPoly(Poly p) {
  int i;
  printf("f(x) = ");
  for (i = p->N -1; i >= 0; i--)
   if (p->a[i] != 0) {
    if ((i<p->N-1) && (p->a[i]>0))
     printf("+");
    if (p->a[i]!=1) printf("%d",p->a[i]);
     if (i > 0) {
      printf("%c",'x');
      if (i > 1) printf("%c%d",'^',i);
     }
     else    printf("%d",p->a[i]);
   }
   printf("\n");
}
Poly PolyTerm(int coeff, int exp) {
   int i;
   Poly t = malloc(sizeof *t);
   t->a = malloc((exp+1) * sizeof(int));
   t->N = exp + 1; t->a[exp] = coeff;
   for (i=0; i<exp;i++)  t->a[i]=0;
   return t;
}
```

poly.c contd.

```
Poly PolyAdd(Poly p, Poly q) {
  int i;  Poly t;
  if (p->N < q->N) {
     t = p; p = q; q = t;
  }
  for (i = 0; i < q->N; i++)
     p->a[i] += q->a[i];
  return p;
}
Poly PolyMult(Poly p, Poly q) {
  int i, j;
  Poly t=PolyTerm(0,(p->N-1)+
                (q->N-1));
  for (i = 0; i < p->N; i++)
    for (j = 0; j < q->N; j++)
      t->a[i + j] += p->a[i] *
                    q->a[j];
  return t;
}
float PolyEval(Poly p, float x) {
  int i; double t = 0.0;
  for (i = p->N - 1; i >= 0; i--)
     t = t*x + p->a[i];
  return t;
}
```

Notes on ADT Example 3 – Implementation

- Polynomial **poly**
  - Structure with two fields
    - int field **N** for the degree of the polynomial,
    - int array **a** representing the coefficients of the terms
    - Note: space required by array will be allocated dynamically at the time the polynomial gets known
    - User-defined type (Poly) points to `struct poly`
  - Function PolyTerm
    - Used to represent a polynomial term
    - Represented as a pointer to `struct poly`
    - Array field of the structure is an array of size M, with the first M-1 cells set to zero, and the last cell set to the coefficient of the term
      - Polynomials can be added together by adding corresponding array indexes (i.e., coefficients)
    - Function reserves space for `struct poly` (the first `malloc`), and then space for the array field (the second `malloc`)
      - Array size is one greater than the exponent, to take account of constant term

Notes on ADT Example 3 – Implementation

- Function PolyAdd
  - Adds two polynomials
  - First determines which of the polynomials has the higher degree, then adds corresponding array terms and stores the result in the larger array
- Function PolyMult
  - More complicated than PolyAdd
  - Creates a new polynomial whose degree is the sum of the degrees of the two polynomials to be multiplied
    - Initializes all the coefficients to zero
    - Multiplies each term of the first polynomial to each term of the second, and adds the result to the corresponding term of the newly created polynomial
  - Function PolyEval
    - Evaluates a polynomial for a given value of x
    - Uses Horner's algorithm, which is based on parenthesization such as:

$$a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

ADT Example 3 - Client

- Sample client program to make use of the polynomial ADT
- Program performs symbolic operations corresponding to the following polynomial equations:

$$
\begin{aligned}
(x+1)^2 &= x^2+2x+1,\\
(x+1)^3 &= x^3+3x^2+3x+1,\\
(x+1)^4 &= x^4+4x^3+6x^2+4x+1,\\
(x+1)^5 &= x^5+5x^4+10x^3+10x^2+5x+1
\end{aligned}
$$

ADT Example 3 - Client

```
Interface file polyclient.c
#include <stdio.h>
#include <stdlib.h>
#include "poly.h"
int main(int argc, char *argv[]) {
  int N = atoi(argv[1]);
  float p = atof(argv[2]);
  Poly t, x;
  int i, j;
  t = PolyAdd(PolyTerm(1,1), PolyTerm(1,0));
  for (i= 1, x = t; i < N; i++)
    x = PolyMult(t, x);
  printf("exponent = %d\n", N);
  showPoly(x);
  printf("x = %.2f, f(x) = %.2f\n",p, PolyEval(x,p));
}
```

ADT Example 3 - Client

```
Input: 1 2
exponent = 1
f(x) = x+1
x = 2.00, f(x) = 3.00
Input: 2 2
exponent = 2
f(x) = x^2+2x+1
x = 2.00, f(x) = 9.00
Input: 3 2
exponent = 3
f(x) = x^3+3x^2+3x+1
x = 2.00, f(x) = 27.00
Input: 10 3
exponent = 10
f(x) = x^10+10x^9+45x^8+120x^7+210x^6+252x^5+210x^4+120x^3+45x^2+10x+1
x = 3.00, f(x) = 1048576.00
```