Data Structures

Lecture 1: Data Types and Data Abstractions

What’s Wrong With This Program?

void f(int x[], int n){int i, y; unsigned
; label10: a = 0; i = 0; label20: if
(i>n-1)goto label30; if (x[i] <=
x[i+1])goto label40; a = 1; y = x[i];
x[i] = x[i+1]; x[i+1]=y;label40:i =
i+1; goto label20; label30: if(a==1)
gotolabel10;}

• Structured programming is about avoiding GOTOs and other bad programming habits.
• Object-Oriented programming extends this objectives.
Rules Of Structured Programming

- One statement per line
- Meaningful variable names
- Proper documentation, i.e., //Comments
- Indenting if, if..else, while, for, etc.
- Match opening and closing braces { }
- Avoiding “clever code”
- Extra space for clarity
- Avoid trivial comments
- Code should be as English-like as possible
- Line up statements on the same “level”
- Use functions wherever possible.

sort.cpp

// sort() – Classic bubble sort
void sort(int x[], int n)
{
    bool switched;
    int i, temp, pass = 0;

    // The outer loop counts passes through the
    // array. At the end of each pass one more
    // element is in the right place.
do {
    switched = false; // Nothing switched yet
    // The inner loop compared adjacent
    // elements in the array to see whether
    // they should be switched.
    for (i = 0; i < n-pass-1; i++)
        if (x[i] > x[i+1]) {
            switched = true;
            temp = x[i];
            x[i] = x[i+1];
            x[i+1] = temp;
        }
} while (switched && pass < n-1);

Why Structured Programming?

- Easier to debug
- More readable, better organized; it makes modifications easier
- Makes group effort easier
- Easier to understand by users and programmers
Interpreting Data

- Information is a little difficult to define that exactly.
- Bits (1s and 0s) are organized into groups of 8 bits (known as bytes).
- The same group of bytes can be interpreted as:
  - an integer
  - a real number
  - a computer instruction
  - a string of characters
  - a computer address

Native Data Types

- Every machine has its own native data types, which may or may not correspond to those of the programming language.
- The language’s compiler must implement the programming language’s data types using the native type available to it.
- Example:
  ```
  int x, y; Reserves 4 different locations
  float a, b; for 4 different variables
  x = x + y;  
  a = a + b; Involve different uses of +
  ```
Native Data Types - An Example

• Assume we have a native instruction:
  \text{MOVE (Source, Dest, Length)}
  which moves length bytes from \text{Source} to \text{Dest}.
  How do we use it to implement variable length strings?

\[
\begin{array}{c}
5 \text{HELLO} + 9 \text{EVERYBODY}
\end{array}
\]

should produce:

\[
\begin{array}{c}
14 \text{HELLO} \quad \text{EVERYBODY}
\end{array}
\]

Abstract Data Types

• Since a data type is a collection of values and a set of operations over these values, we can define them mathematically even before we implement them.
• Consider the abstract data type rational numbers:

\[
\begin{array}{c}
\text{// value definition} \\
\text{abstract typedef } \langle \text{int, int} \rangle \text{ RATIONAL;} \\
\text{condition } \text{RATIONAL}[1] \neq 0;
\end{array}
\]

\[
\begin{array}{c}
\text{\text{// operator definitions}} \\
\text{abstract equal (a, b)} // a == b \\
\text{RATIONAL } a, b \\
\text{postcondition equal == (a[0]*b[1] == a[1]*b[0])}
\end{array}
\]
Abstract Data Types (continued)

abstract RATIONAL makerational(a, b)  //written [a,b]
int a, b;
precondition b != 0
postcondition
    makerational[0]*b == a*makerational[1]

abstract RATIONAL add(a, b)  //written a+b
RATIONAL a, b;
postcondition
    add = (a[0]*b[1]+b[0]*a[1], a[1]*b[1])

abstract RATIONAL mult(a, b)  //written a*b
RATIONAL A, b;
postcondition mult==(a[0]*b[0], a[1]*b[1])

ADT for Variable-Length Strings

abstract typedef <<char>> STRING;
abstract length(s)
STRING s;
postcondition length = len(s)
abstract STRING concat (s1, s2)
STRINGs1, s2;
postcondition concat == s1+s2
abstract STRING substr(s1, i, j)
STRING s1;
int i, j;
precondition
    0<= i < len(s1);
    0 <= j < len(s1)-i;
postcondition substr=sub(s1, i, j);
ADT for Variable-Length Strings (continued)

abstract pos(s1, s2)
STRINGV s1, s2;
postcondition // lastpos = len(s1) - len(s2)
((pos=-1) && for (i = 0; i < lastpos; i++)
(s2 <> sub(sub(s1, i, len(s2)))))
// s2 is not within s1
|| (pos > 0) && pos <= lastpos)
&& (s2 == sub(str1, pos, len(s2))
&& (for (i = 1; i < pos ; i++)
(s2 <> sub (s1, i, len(s2))));
// s2 is within s1

Data Types in C and C++

• The 4 native data types in C are:
  – int, float, char and double
• int can be qualified with:
  – long, short, or unsigned
• A C variable declaration specifies 2 things:
  – How much storage is allocated
  – How is data represented in memory
Pointers

• Pointers allow us to reference a data object location as well as its value:

  int *pi; int i; pi = &i;
  float *pf; float f; pf = &f;
  char *pc; char c; pc = &c;

• We can convert between pointer types:
  pi = (int *) pf;

• Question – What do these mean?
  *pi + 2 *(pi+2) pi[2]

Arrays

• An array is a one-dimensional (or more) structure of similar data types.
  – A one-dimensional array is a list or a vector.
  – e.g., int a[100];

• Strings (in C) are arrays of characters with a null byte at the end.
  char s[100];
  with support functions in string.h like strcat, strcpy, strlen.
Multidimensional Arrays

2-dimensional arrays

```
int t[3][5];
```

```
0 1 2 3 4
```

```
0
```

```
1
```

```
2
```

3 lists of 5 elements

A two-dimensional array like this is called a **matrix**

3-dimensional arrays

```
int class[3][4][25];
```

colleges in university

depts in college

class in dept

Implementation of Arrays in Pascal

<table>
<thead>
<tr>
<th>-3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[-3]</td>
<td></td>
</tr>
<tr>
<td>a[-2]</td>
<td></td>
</tr>
<tr>
<td>a[-1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[2]</td>
<td></td>
</tr>
<tr>
<td>a[3]</td>
<td></td>
</tr>
</tbody>
</table>

One-dimensional

```
position of x[i] = base + (i-lbound)*esize
```
Implementation of Arrays in Pascal (continued)

Two-dimensional

<table>
<thead>
<tr>
<th>x[0][0]</th>
<th>x[0][1]</th>
<th>x[0][2]</th>
<th>x[0][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[1][0]</td>
<td>x[1][1]</td>
<td>x[1][2]</td>
<td>x[1][3]</td>
</tr>
<tr>
<td>x[2][0]</td>
<td>x[2][1]</td>
<td>x[2][2]</td>
<td>x[2][3]</td>
</tr>
<tr>
<td>x[3][0]</td>
<td>x[3][1]</td>
<td>x[3][2]</td>
<td>x[3][3]</td>
</tr>
<tr>
<td>x[4][0]</td>
<td>x[4][1]</td>
<td>x[4][2]</td>
<td>x[4][3]</td>
</tr>
<tr>
<td>x[5][0]</td>
<td>x[5][1]</td>
<td>x[5][2]</td>
<td>x[5][3]</td>
</tr>
</tbody>
</table>

position of \( a[i][j] \) = base + \( [(i_1-1)\cdot r_2+i_2-1_2)] \cdot \text{esize} \)

Implementation of Arrays in C/C++
Arrays As Parameters

```c
float avg(float a[], int size)
//no size specified
{
    int i;
    float sum;
    sum = 0;

    for (i = 0;  i < size;  i++)
        sum += a[i];
    return(sum/size);
}
```

String Operations

```c
const int StrSize = 80;
char string[StrSize];

int strlength(char string[])
{
    int i;
    for (i = 0; string[i]!= '\0'; i++)
        ;
    return(i);
}
```
String Operations (continued)

```c
int strpos(char s1[], char s2[])
{
    int i, j, k;

    for(i = 0; s1[i] != '\0'; i++) {
        for(j = i, k = 0; s2[k] != '\0'
            && s1[j] == s2[k]; j++, k++)
        ;
        if(k > 0 && s2[k] == '\0')
            return i;
    }
    return -1;
}
```

String Operations (continued)

```c
void strconcat(char s1[], char s2[])
{
    int i, j;
    for (i = 0;  s1[i] != '\0'; i++)
    ;
    for (j = 0;  s2[j] != '\0';
        s1[i++] = s2[j++])
    ;
    s1[i] = '\0';
}
```
String Operations (continued)

```c
void strcopy(char s1[], char s2[])
{
    int i;
    for (i = 0;  s1[i] != '\0'; i++)
        s1[i] = s2[i];
    s1[i] = '\0';
}
```

String Operations (continued)

```c
void substring(char s1[], int i, int j, char s2[])
{
    int k, m;
    for (k = i, m = 0;  m < j;
        s2[m++] = s1[k++])
    {
        s2[m] = '\0';
    }
```
Structures

- While an array is a *homogeneous* collection of data, a *structure* is a *heterogeneous* collection of data, i.e., a collection of fields that may be quite different.

```
struct
{
    char first[10];
    char midinitial;
    char last[20];

} sname, ename;
```

Structures (continued)

- If we wish to define such a type:

```
struct nametype
{
    char first[10];
    char midinitial;
    char last[20];

};

struct nametype  sname, ename;
```
Structures (continued)

• We could define such a type using a **typedef**:

```c
typedef struct nametype {
    char first[10];
    char midinitial;
    char last[20];
} NameType;
```

NameType **sname, ename**;

Unions

• Unions allow a variable to be interpreted in several different ways:

```c
union utype {
    int i;
    float x;
    char c;
};
```

• If we say:

```c
u.c = 'a';
```

```c
putchar(u.c);
```

```c
u.x = 0.5;
```

we have changed **u.c**’s value as well – they share the same memory
Unions (continued)

• We can use unions to create variant records:
  
  ```c
  typedef enum insuretype {Life, Auto, Home};
  typedef struct {
    char street[50];
    char city[10];
    char state[2];
    char zip[5];
  } addr;

  typedef struct {
    int month;
    int date;
    int year;
  } date;

  typedef struct {
    int polnumber;
    char name[30];
    addr address;
    int amount;
    float premium;
    int kind; //Life, Home or Auto
  } policy;
  ```
union {
  struct {
    char beneficiary[30];
    date birthday;
  } Life;
  struct {
    int autodeduct;
    char license[10];
    char state[2];
    char model[15];
    int year;
  } Auto;
  struct {
    int homededuct;
    int yearbuilt;
  } Home;
} policyinfo;
} policy;

Printing Variant Records

• We can print the policy information:
  policy p;
  ...
  ...
  if (p.kind == Life)
    printf("\n%s %2d/%2d/%4d",
           p.policyinfo.Life.beneficiary,
           p.policyinfo.Life.birthday.month,
           p.policyinfo.Life.birthday.date,
           p.policyinfo.Life.birthday.year);
  else if (p.kind == Auto)
    printf("\n%d %s %s %s %d",
           p.policyinfo.Auto.autodeduct,
           p.policyinfo.Auto.license,
           p.policyinfo.Auto.state,
           p.policyinfo.Auto.model,
           p.policyinfo.Auto.year);
else if (p.kind == Home)
    printf("\n%\d %d",
    p.policyinfo.Home.homededuct,
    p.policyinfo.Home.yearbuilt);
else
    printf("\nbad type %\d in kind", p.kind);
• We could declare an array of structures by writing:

policy a[100];

---

Structures As Parameters

• Traditionally in C, we pass the structure’s address (not the whole structure) to save memory. A function using a structure might look like this:

  //Prints name in a neat format
  int writename(struct nametype *name)
  {
      int count, i;
      printf("\n");
      count = 0;
      for (i = 0;  (i<10) && name-> first[i] != '\0'; i++)
          putchar(name->first[i]);
      count++;
  }
putchar(' ');
count++;
if (name -> midinitial != ' ') {
    printf("%c%s", name -> midinitial, ". ");
count += 3;
}
for (i = 0; (i<20) && (name->last[i] !='\0'); i++) {
    putchar(name->last[i];
count++;
}
}

---

**Four features of Object-Oriented Programming**

- **Data abstraction** – using data in a program as we conceive it in the real world.
- **Encapsulation** – manipulate private data only within the class itself or within “friend” classes.
- **Polymorphism** – Overloading functions and operators, allowing them to be used in multiple ways differing only in parameter (or operand) types.
- **Inheritance** – designing new classes based on previously defined classes.
An Example in C++ - A **Rational** Class

class Rational {

public:
    Rational operator +(Rational);
    Rational operator +(long);
    Rational operator *(Rational);
    Rational operator /(Rational);
    int operator ==(Rational);
    void print(void);
    void setrational(long, long);

private:
    long numerator;
    long denominator;
    void reduce(void);

};

Rational Rational::operator +(Rational r)
{
    int k, denom, num;
    Rational rnl;

    //First reduce both rationals to lowest terms
    reduce();
    r.reduce();

    //implement the line k=rden(b, d)
    rnl.setrational(denominator, r.denominator);
    rnl.reduce();
    k = rnl.denominator;

    //Compute the result's denominator
    denom = denominator*k;
}
//Compute the result's numerator
num = numerator\*k +
    rnl.numerator*(denom/rnl.denominator);

//Form a Rational from the result and reduce
rnl.setrational(num, denom);
rnl.reduce();
return(rnl);
}

void Rational::print(void)
{
    cout << numerator << "/" << denominator
         << endl;
}

void Rational::setrational(long n, long d)
{
    if (d == 0)   {
        cerr << "Error: denominator may not "
             "be zero" << endl;
        exit(1);
    }
    numerator = n;
    denominator = d;
    reduce();
}
Rational Rational:: operator *(Rational r)
{
    Rational rnl, rnl1, rnl2;
    int num, denom;

    // reduce both inputs to lowest terms
    reduce();
    rnl.reduce();

    // switch numerators and denominators and
    // reduce
    rnl1.setrational(numerator, r.denominator);
    rnl1.reduce();
    rnl2.setrational(r.numerator, denominator);
    rnl2.reduce();

    // compute result
    num = rnl1.numerator * rnl2.numerator;
    denom = rnl1.denominator * rnl2.denominator;
    rnl.setrational(num, denom);
    rnl.print();
    return(rnl);
}
Rational Rational::operator /(Rational r)
{
    Rational rnl1, rnl2, rnl3;

    // compute the reciprocal of r
    rnl1.setrational (numerator, denominator);
    rnl2.setrational(r.denominator, r.numerator);

    // Multiply by the reciprocal
    return(rnl1*rnl2);
}

int Rational::operator ==(Rational r)
{
    reduce();
    r.reduce();
    if (numerator == r.numerator
        && denominator == r.denominator)
        return (1);
    else
        return(0);
}
void Rational::reduce(void)
{
    int a, b, rem, sign;

    if (numerator == 0)
    {
        denominator = 1;
        sign = 1; // assume positive
    }

    // check if any negatives
    if (numerator < 0 && denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }

    if (numerator < 0) {
        numerator = -numerator;
        sign = -1;
    }

    if (denominator < 0) {
        denominator = -denominator;
        sign = -1;
    }

    if (numerator > denominator) {
        a = numerator;
        b = denominator;
    }
}
else {
    a = denominator;
    b = numerator;
}

while (b != 0) {
    rem = a % b;
    a = b;
    b = rem;
}
numerator = sign * numerator/a;
denominator = denominator / a;

---

Function Overloading – An Example

Rational Rational::operator +(long i)
{
    Rational r, r2;

    r.setrational(i, 1);
    r2.setrational(numerator, denominator);
    return(r + r2);
}
Using the *Rational* Class

```c
#include "rational.h"
#include <string.h>

void main(void)
{
    int readtoken(char **);
    void error(char *);

    char *optr, *token1, *token2, *token3;
    int    int1, int2;
    Rational opnd1, opnd2, result;

    while (readtoken(&optr) != EOF) {
        // read the operator
        readtoken(&token1);
        // read the first integer's
        // character string
        int1 = atol(token1);
        // convert the first token to an
        // integer
        readtoken(&token2);
```
if (strcmp(token2, "/") != 0)
    // convert the integer operand
    // to a Rational
    opnd1.setrational(int1, 1);
else {
    // get the denominator of the
    readtoken(&token3);
    int2 = atol(token3);
    // convert the numerator and
    // denominator to a Rational
    opnd1.setrational(int1, int2);
    readtoken(&token2);
}

// get the second operand
int1 = atol(token2);
readtoken(&token2);
if (strcmp(token2, "/") != 0)
    // convert the operand to Rational
    opnd2.setrational(int1, 1);
else {
    // get the operand’s denominator
    readtoken(&token3);
    int2 = atol(token3);
    // convert the numerator and
    // denominator to a Rational
    opnd2.setrational(int1, int2);
    readtoken(&token2);
}
if (strcmp(token2, ";") != 0) {
    cout << "ERROR! ; expected, not" << " found" << endl;
    exit(1);
}

// apply the operator to the operands
if (*optr == '+')
    result = opnd1 + opnd2;
else if (*optr == '*')
    result = opnd1 * opnd2;
else {
    cout << "ERROR: illegal operator;" << " must be * or +" << endl;
    exit(1);
}
result.print();

Constructors

- A constructor performs necessary initialization work when an object of this class is first defined.
- Constructors can be used for:
  - initializing private data
  - converting input values into the object’s class
  - allocating necessary storage
Rational Constructors

Rational::Rational(void)
{
    // assume that the rational is 0
    numerator = 0;
    denominator = 1;
}

Rational::Rational(long i)
{
    numerator = i;
    denominator = 1;
}

Rational::Rational(long num, long denom)
{
    numerator = num;
    denominator = denom;
}