Distributed and Parallel Technology

C Revision (Part II)

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

[0]No proprietary software has been used in producing these slides

Where we are

So far we covered:

- C Program Layout
- C Types:
  - Basic C Types
  - Arrays
  - Structures
  - Pointers
  - Type Coercions

Union Types

- Union types can take different types of value at different times
- To declare a union type, use

  `union identifier { type1 id1; ... typeN idN };`

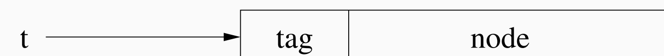
- `union identifier` is the name of a new type
- can be *either* `type1`, accessed through `id1`, or `type2` accessed through `id2` etc
- space is allocated for the *largest* `typeI`
- other types are aligned within this space
- *NB:* no way to tell at run time which type is intended (use an explicit tag)
- To declare a variable of union type, use

  `union identifier1 identifier2;`

- `identifier2` is a variable of type `union identifier1`

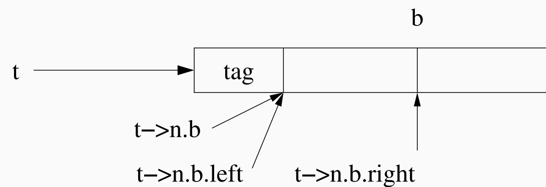
A Tree Structure

```
/* enumeration of possible tags of type int  */
typedef enum { Leaf = 0, Branch = 1 } Tag;
/* define composition of a branch */
struct branch { struct tree *left, *right; };
/* define alternatives of a node */
union node { struct branch b; int value; };
/* define a tree as a tagged node */
struct tree { Tag tag; union node n; };
...
/* allocate a pointer to a tree */
struct tree *t;
/* allocate a tree structure */
t = (struct tree *) malloc(sizeof(struct tree))
```



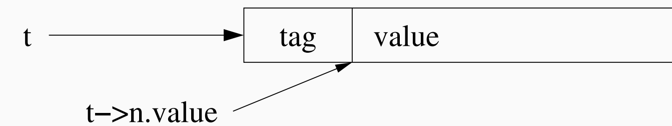
A Tree Structure

```
/* enumeration of possible tags of type int */
typedef enum { Leaf = 0, Branch = 1 } Tag;
/* define composition of a branch */
struct branch { struct tree *left, *right; };
/* define alternatives of a node */
union node { struct branch b; int value; };
/* define a tree as a tagged node */
struct tree { Tag tag; union node n; };
...
/* allocate a pointer to a tree */
struct tree *t;
/* allocate a tree structure */
t = (struct tree *) malloc(sizeof(struct tree))
```



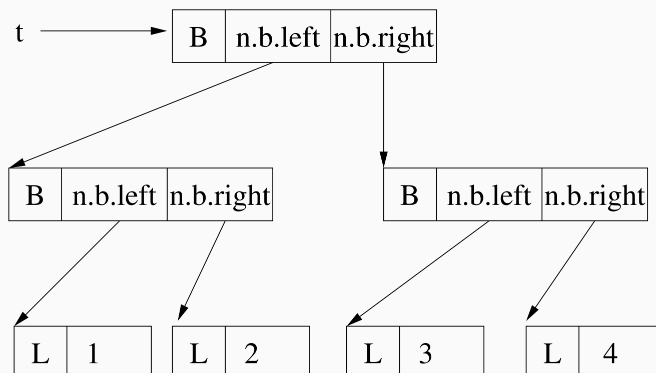
A Tree Structure

```
/* enumeration of possible tags of type int */
typedef enum { Leaf = 0, Branch = 1 } Tag;
/* define composition of a branch */
struct branch { struct tree *left, *right; };
/* define alternatives of a node */
union node { struct branch b; int value; };
/* define a tree as a tagged node */
struct tree { Tag tag; union node n; };
...
/* allocate a pointer to a tree */
struct tree *t;
/* allocate a tree structure */
t = (struct tree *) malloc(sizeof(struct tree))
```



Example Tree

```
int a[4] = { 1, 2, 3, 4 };
t = mkTree(0,3,a);
```



Example Code: Building a Tree

```
struct tree *
mkTree(int from, int to, int *arr) {
  if (from>to) {
    return (struct tree *)NULL;
  } else if (from==to) {
    struct tree * t;
    t = (struct tree *) malloc(sizeof(struct tree));
    t->tag = Leaf;
    t->n.value = arr[from];
    return t;
  } else {
    struct tree * t, *left, *right;
    int mid = (from + to) / 2;
    left  = mkTree(from,mid,arr);
    right = mkTree(mid+1,to,arr);
    t->tag = Branch;
    t->n.b.left = left;
    t->n.b.right = right;
    return t;
  }
}
```

Exercise

Useful exercises, involving this tree representation, are at the end of this set of slides.

Control Constructs

- Strong syntactic similarity to C# or Java
- *BUT:* without object-orientation, ie. semantic differences
- Assignment

  `expression1 = expression2;`
- set address from `expression1` to value from `expression2`
- block statement

  `{ declarations; statements }`
- `declarations` are optional
- declared values are *local* to `statements`

Conditional Constructs: `if`

```
if (expression)
 statement1;
else
 statement2;
```

- A (single-selection) conditional statement
- *NB:* in C there are no true or false values
  - 0 is interpreted as false
  - every other value is interpreted as true
- the `else` branch is optional

Conditional Constructs: `switch`

```
switch(expression)
{  case constant1: statements1;
    ...
    default: statements;
}
```

- A (multi-selection) conditional statement
- `statements` can be one or more statement
- a `break` leaves the `switch` statement
- *NB:* it's possible to "fall-through" to the next block, if there is no `break` (unlike in C#)
- the `default` branch applies, if none of the previous cases matches

Loops: while

```
while (expression)
 statement;
```

- A While loop: execute statements, while `expression` is true
- `statement` is optional
- to combine statements, use the blocking statement {...}

Loops: for

```
for (statement1;expression;statement2)
 statement3;
```

- A For loop
- `statement1` performs initialisation (once)
- `expression` checks for termination (start of each iteration)
- `statement2` prepares for the next iteration, eg. increases the counter (end of each iteration)
- `statement3` is the loop body (optional)
- `break` can be used in `while` or `for` loops to terminate the loop;
- `continue` can be used to jump to next iteration

Functions

To define a function, use

```
type identifier(id1,id2...)
type1 id1; type2 id2;
...
{  declarations;
   statements; }
```

- The return `type` is optional
  - by default the type is `int`
  - if the function does not return a value, the type is `void`
- `declarations` is optional (they declare local variables)
- `id1, id2` ... are the formal parameters
- can also be defined inline as:

  `type identifier(type1 id1,type2 id2,...)`

Parameter Passing

- Parameter passing is *call by value*, i.e. value of the actual parameter is copied to the space for the formal parameter
- thus, a change to formal does not change actual
- use address operators for *call by reference*
  - call function with `&actual`
  - in function manipulate `*formal` to change actual
- *NB:* arrays effectively passed by reference

Input/Output

- Files are character sequences
- Access files via a `FILE *`
- To open a file, use
  `FILE * fopen(char *string, char mode)`
- `string` must contain the path to a file
- `mode` can be `r` for read, `w` for write or `a` for append
- the return value is a file pointer or NULL if no file
- To close a file, use
  `fclose(FILE *fd)`
- for input/output, C handles characters as ints i.e. 16 bit values
- To read the next value, use
  `int getc(FILE *fd)`
- This returns `EOF` at the end of the file.
- To write the next value, use
  `int putc(int c,FILE *fd)`
- This returns `EOF` on failure

Input/Output

- To read *formatted* data, use
  `int fscanf(FILE *fd, char *string, lval1, lval2...)`
- string specifies format of input, using these conversion specifications
  - %c for character
  - %d for integer
  - %f for double
- `lval` is *address* of where to put converted byte sequence
- must be one conversion spec for each lval
- To write *formatted* data, use
  `int fprintf(FILE *fd, char *string, exp1, exp2...)`
- `string` is format for ouput
- mix text with conversion specs
- must be one conversion spec for each `exp`
- conversion spec as above
- additionally: %s for string
- can precede format character with width/precision info eg. %4.2f for double, at least 4 chars wide with 2 chars after decimal point

Input/Output

- For keyboard input, use
  `int getchar()`
- same as getc(stdin)
- or
  `scanf(char *string, lval1, lval2...)`
- same as fscanf(stdin, string, lval1, lval2...)
- For screen output, use
  `int putchar(int n)`
- same as putc(int,stdout)
- or
  `printf(char *string, exp1, exp2...)`
- fprintf(stdout,string,exp1,exp2...)
- For command line input, use
  `main(argc,argv)`
  `int argc; char ** argv;`
- argc is count of number of command line arguments
- argv are the command line arguments (including the name of the executable at position 0)

Example: Matrix Multiplication

We want to define

- integer matrices,
- held in file as:

```
matrix1: row 1/col 1 ... matrix1: row 1/col n
...
matrix1: row m/col 1 ... matrix1: row m/col n
```

- using 2D array representation
- with fixed maximum sizes

Example: Matrix Multiplication

```
#include <stdio.h>
#include <stdlib.h>
#define MAXR 5
#define MAXC 5

/* read a matrix from a stream */
readMatrix(fin,M,m,n)
FILE * fin;     /* input stream */
int M[][MAXC]; /* matrix to store values in */
int m,n;        /* number of rows and columns*/
{  int i,j;

   for(i=0;i<m;i++)    /* iterate over all rows */
    for(j=0;j<n;j++)   /* iterate over all columns */
     fscanf(fin,"%d",&(M[i][j]));   /* read data */
}
```

- must specify size of column for 2D array parameter
- use `&(M[i][j])` to pass address of element i/j to fscanf

Example: Matrix Multiplication

Similarly, for writing a matrix:

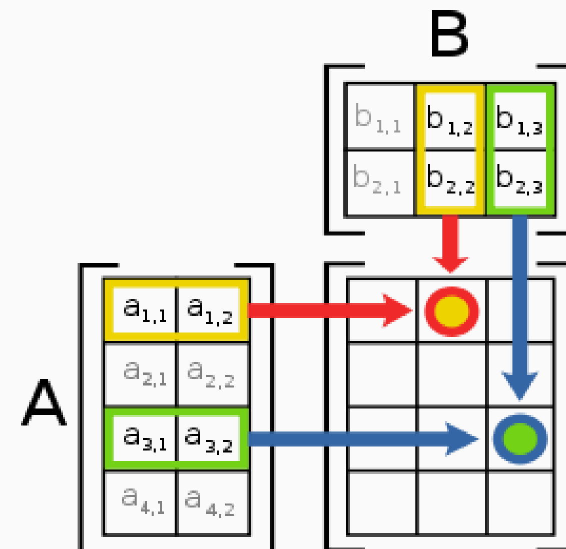
```
/* write a matrix to a stream */
writeMatrix(fout,M,m,n)
FILE * fout;    /* output stream */
int M[][MAXC]; /* matrix to read values from */
int m,n;        /* number of rows and columns */
{  int i,j;

   for(i=0;i<m;i++)      /* iterate over all rows */
   {  for(j=0;j<n;j++) /* iterate over all columns */
       fprintf(fout,"%d ",M[i][j]); /* write data */
      putc('\n',fout);
   }
}
```

Example: Matrix Multiplication

Now, the classical matrix multiplication function, as 3 nested loops:

```
 /* Input: integer matrices M1 (m row, n columns), M2 (n row, m colum
 /* Output: integer matrices M3 (m row, m columns), s.t. M3 = M1 * M3
matrixProd(M1,M2,M3,m,n)
int M1[][MAXC]; /* 1st input matrix */
int M2[][MAXC]; /* 2nd input matrix */
int M3[][MAXC]; /* Output matrix */
int m,n;         /* Number of rows and columns of M1 */
{  int i,j,k;

   for(i=0;i<m;i++)     /* iterate over all rows */
    for(j=0;j<m;j++)    /* iterate over all columns */
    {  M3[i][j]=0;
       for(k=0;k<n;k++) /* compute dot product */
        M3[i][j] =
         M3[i][j]+M1[i][k]*M2[k][j];
    }
}
```

Matrix Multiplication



[0]Picture from http://en.wikipedia.org/wiki/Matrix_multiplication

File Format

```
m n
matrix1: row 1/col 1 ... matrix1: row 1/col n
...
matrix1: row m/col 1 ... matrix1: row m/col n
matrix 2: row 1/col 1 ... matrix2: row 1/col m
...
matrix 2: row n/col 1 ... matrix2: row n/col m
```

Main Function

```
main(argc,argv)
int argc; char ** argv;
{  FILE * fin;
   int m1[MAXR][MAXC];
   int m2[MAXR][MAXC];
   int m3[MAXR][MAXC];
   int m,n;

   fin = fopen(argv[1],"r");
   fscanf(fin,"%d %d",&m,&n);
   readMatrix(fin,m1,m,n);
   readMatrix(fin,m2,n,m);
   fclose(fin);
   writeMatrix(stdout,m1,m,n);
   putchar('\n');
   writeMatrix(stdout,m2,n,m);
   putchar('\n');
   matrixProd(m1,m2,m3,m,n);
   writeMatrix(stdout,m3,m,m); }
```

Main Function

Discussion

- open file in read mode using argv[1] as file name
- pass addresses for int variables `m` and `n` to `fscanf`
- stdout as file pointer for `writeMatrix/fprintf`

Example: Matrix Multiplication 2

Disadvantages of 2D array representation:

- have to allocate worst case space
- poor representation for parallelisation
- may want to locate parallelism in row/column manipulation

Example: Matrix Multiplication 2

Instead, use an array of arrays

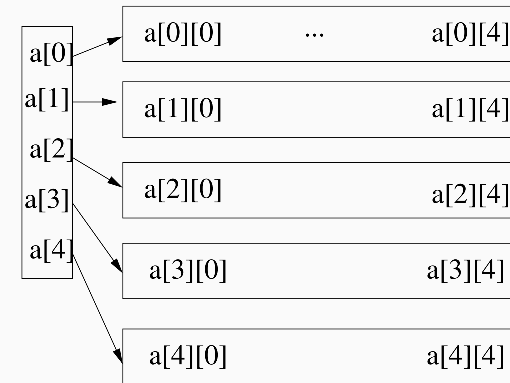
```
int * allocVector(int n)
{
  return (int *)malloc(n*sizeof(int));
}

int ** allocMatrix(int m, int n)
{
  int ** newM = (int **)malloc(m*sizeof(int *));
  int i;
  for (i=0; i<m; i++)
    newM[i] = allocVector(n);
  return newM;
}
```

Discussion

- `allocMatrix` returns array of m arrays of n ints
- in other functions, only need to change matrix type from [][] to

Array data layout

Layout of a matrix generated by `allocMatrix`:



Example: Matrix Multiplication 2

```
readMatrix(fin,M,m,n)
FILE * fin;
int ** M;
...

writeMatrix(fout,M,m,n)
FILE * fout;
int ** M;
...

matrixProd(M1,M2,M3,m,n)
int **M1,**M2,**M3;
...
```

Example: Matrix Multiplication 2

```
main(argc,argv)
int argc;
char ** argv;
{  FILE * fin;
   int ** m1;
   int ** m2;
   int ** m3;
   int m,n;

   ...
   m1 = allocMatrix(m,n);
   m2 = allocMatrix(n,m);
   m3 = allocMatrix(m,m);
   ...
}
```

Example: Optimised Matrix Multiplication

```
matrixProd(M1,M2,M3,m,n,z)
int **M1,**M2,**M3;
int m,n,z;
{ int i,j,k,ii,jj,kk,temp;
  int *pa, *pb;

  for (jj=0; jj<m; jj=jj+z)
    for (kk=0; kk<n; kk=kk+z)
      for (i=0; i<m; i++)
        for (j=jj; j < jj+z; j++) {
          pa = &M1[i][kk]; pb = &M2[kk][j];
          temp = (*pa++)*(*pb);
          for (k=kk+1; k < kk+z; k++) {
            pb = pb+m;
            temp += (*pa++)*(*pb);
         }
         M3[i][j] += temp;
        }
}
```

Exercise

- Write a function that checks, whether a tree is balanced.
- Write a function `struct node *readTree(FILE *fin, int n)` that reads `n` integer values from a file and builds a balanced tree.
- Write a function `void insert(struct node *t, int n)` that inserts a value `n` into a tree, whose leaves are sorted in ascending order.
- For an input array of length n, how much space does such a tree consume?
- Change the tree data structure, to use a more efficient representation of the `tag`.