CSCI 135
Arrays

Arrays

Recall that various composite types differ by homogeneity of element types, size characteristics, storage overhead, ordering, access, etc.

An **array** is:

- A sequence of elements that is:
  - Homogeneous (*i.e.*, each element has same type)
  - Fixed size
  - No storage overhead
  - Random access to elements (through index)
- One of two core C composite types (but not really)
- Ex: [1.4, 2.7, 1.8], [“al”, “barb”, “carol”]
- Memory for entire array allocated before program starts running; called **static memory allocation**
  $\Rightarrow$ not a good data structure if your main operations are insert/delete and you don’t need random access on a collection.

⚠ C++11 also has arrays, which are different from C arrays

Declaring An Array

Base_Type identifier[size]

where Base_Type is any type (int, float, typedef'ed type, etc.) and size is an integer literal

⚠C99 (and later) allows size to be an expression that evaluates to an int (for 1D arrays).

Examples:

```
int x1[64];                    reserves 256(?) bytes RAM
unsigned long x2[64];            for x1, and so on
string x3[256];
int y[4] = {1,5,17,6};         initializes y's elements
```

⚠Array elements are not initialized with default values

Array Usage - Example

Ex: Find lowest and 2 highest grades

```
#include <climits>
...
const int NUMSTUD=256;   need to keep track of size explicitly
int grades[NUMSTUD];
// Precond: all grades are distinct, non-negative
int min = INT_MAX;       defined in climits
int max,second = 0;
for (int i=0; i<NUMSTUD; i++)
{
   if (grades[i]<min) min = grades[i];
   if (grades[i]>max) {
      second = max;
      max = grades[i];
   };
};
```

Doesn't work if second<grades[i]<max (HW).

? Which other data sources can use essentially the same code?

Some Gotchyas

- Indices go from 0 to size-1 (NOT 1 to size)
- Good idea to have a const variable indicating array maximum size (instead of hardcoding the size constant everywhere it is used).
  Common style: all uppercase
- Assignment does not assign entire array!
- NO bounds checking by compiler!
  $\Rightarrow$ undefined results (potentially crashes program) if programmer doesn't explicitly check

? What does this program do?

```
int x[64];      indices go 0...63
int y,z;
x[65] = 7;      <- bounds error
```

Some Gotchyas

- Indices go from 0 to size-1 (NOT 1 to size)
- Good idea to have a const variable indicating array maximum size (instead of hardcoding the size constant everywhere it is used).
  Common style: all uppercase
- Assignment does not assign entire array!
- NO bounds checking by compiler!
  $\Rightarrow$ undefined results (potentially crashes program) if programmer doesn't explicitly check

❓ What does this program do?

```
int x[64];      indices go 0...63
int y,z;
x[65] = 7;      ← bounds error
```

❗ Undefined! Might crash, might assign 7 to z, might overwrite some machine code with 7, ...

$\Rightarrow$ common source of security holes

Passing Arrays to Functions

Ex: `void foo(int arr[], int arrsize) {...}`

- Only the [] (not the declared size) is in the formal
- Almost always need to pass size separately (no way for foo to iterate through arr otherwise)
- No overhead in passing by value, since only the address of the array is copied (not all elements of array).
- Since only the address is passed, foo now has access to caller's memory space $\Rightarrow$ Huge security issue!
- Can be thought of as always passed by reference (we will get more precise later); *i.e.*, think of it as implicit & between int and arr above
- Common style: size argument immediately after array argument
- Don't forget - you can still pass individual array elements (instead of the whole array) to a function.
  Ex: `age=foo(birthyears[5],2015)` where the prototype is `int foo(int yob, int current_year)`

⚠ Brackets are in function declaration, NOT in call

Example

Spec: return average of array elements

```
double mean (int data[], int size) {
  int sum;
  for (int i = 0; i<size; i++) sum += data[i];
  return (double) sum / (double) size;
};
```

`cout << mean(grades,NUMSTUD) << endl;` would print the average grade

Note: above function can be used to find average of any-sized array (not a good idea to code it assuming some hardcoded size)

Modifying and Returning Arrays in Functions

Recall: array arguments can be thought of as pass by reference
$\Rightarrow$ function can modify array

Often want to disallow that
$\Rightarrow$ Use `const` modifier before parameter
Ex: `double average (const int data[], const int size)`
(or just const size if we want to allow modifying array contents but not changing size)

Returning arrays: we will talk about later (after pointers).

Example: Remove Negatives

Spec: Given a sequence data of integers, return a sequence of all negatives in data. Also remove the negative elements from data.

```
// Precond: negdata dimension is >= size
// Postcond: negdata contains all negative x in data
//           negsize is number of negative elements
void removeNegs(int data[], int & size,
                int negdata[], int & negsize) {
  negsize=0;
  for (int i=0; i<size; i++)
    if (data[i]<0) {
      negsize++;
      negdata[negsize-1] = data[i];
      delArrayElem(i,data,size);
      };
  return;
  };
```

? What would happen if precondition isn't met?

HW: make precondition null (*i.e.*, make program handle all cases)

Example: Insertion Sort Step

Spec: Insert an element into a [nondecreasing] sorted array, keeping it sorted.

```
void insertElem(int data[], int maxsize,
                int &cursize, int elem)
{
  int i=0;
  while (elem > data[i]) i++;
  // assert (elem > data[i-1]) && (elem <= data[i])
  for (int j=i+1; j<=cursize; j++)
    data[j]=data[j-1];
  data[i] = elem;
  cursize++;
  return;
};
```

Example: Insertion Sort Step

Spec: Insert an element into a [nondecreasing] sorted array, keeping it sorted.

```
void insertElem(int data[], int maxsize,
                int & cursize, int elem)
{
  int i=0;
  while (elem > data[i]) i++;
  // assert (elem > data[i-1]) && (elem <= data[i])
  for (int j=i+1; j<=cursize; j++)
    data[j]=data[j-1];
  data[i] = elem;
  cursize++;
  return;
};
```

 This copies data[i] to all subsequent elements of data (erasing prior contents)

Example: Insertion Sort Step

Spec: Insert an element into a [nondecreasing] sorted array, keeping it sorted

```
void insertElem(int data[], int maxsize,
                int & cursize, int elem)
{
  int i=0;
  while (elem > data[i]) i++;
  // assert (elem > data[i-1]) && (elem <= data[i])
  for (int j=cursize; j>i; j--)
    data[j]=data[j-1];
  data[i] = elem;
  cursize++;
  return;
};
```

Example: Insertion Sort Step

Spec: Insert an element into a [nondecreasing] sorted array, keeping it sorted

```
void insertElem(int data[], int maxsize,
                int & cursize, int elem)
{
  int i=0;
  while (elem > data[i]) i++;
  // assert (elem > data[i-1]) && (elem <= data[i])
  for (int j=cursize; j>i; j--)
    data[j]=data[j-1];
  data[i] = elem;
  cursize++;
  return;
};
```

 We still aren't handling the case where elem is bigger than everything in data or if cursize==maxsize (quick homework)

Selection Sort Outline

Spec: sort an array (named data)
Idea:

- At beginning of iteration i (starting at 0), assume that first i elements of data[] are correct.
- Write loop body so that first i+1 elements are sorted by end of loop body (thus, beginning of iteration i+1).

$\Rightarrow$ after n iterations, all n elements of data[] are sorted.
A **loop invariant** is the condition that holds at beginning of each iteration in a loop (*i.e.*, an assertion positioned at beginning of loop body).

```
for (i=0; i<maxsize; i++) {
  // inv: data[0..(i-1)] is sorted   Imprecise (for now)!
  S              now we only need to write S!
  // assert: data[0..i] is sorted    Imprecise (for now)!
  }
```

Example: Selection Sort

Iteration i: S needs to find smallest element in data[i..maxsize] and swap it with data[i].

```
for (int i=0; i<maxsize; i++)
  // inv: data[0..(i-1)] contains the i
  //   smallest elements of data, in sorted order
  minIndex = i;                    index of smallest element
  for (int j=i+1; j<maxsize; j++)
    if (data[j]<data[minIndex])
      minIndex=j;                  find smallest in rest of list
  temp=data[minIndex];             ...and do the swap
  data[minIndex] = data[i];
  data[i] = temp;
  // assert: data[0..i] contains the i+1 smallest
  //   elements of data, in sorted order
  }
```

? What are the boundary cases? Does this work for all of them? Homework.

Multidimensional Arrays

Base_Type identifier[dim1Size][dim2Size]...[dimNSize];

where each Size is an integer literal

Ex: store a 256 X 256 array of pixels (gray-scale)

```
typedef unsigned int Pixel;
const unsigned IMAGE_DIM = 256;
Pixel image[IMAGE_DIM][IMAGE_DIM];

for (int row = 0; row<IMAGE_DIM; row++)
  for (int col = 0; col<IMAGE_DIM; col++)
    image[row][col] = 0;          Make entire image black
...
```

⚠ Not image[row,col]

Example – Pascal's Triangle

Spec: Create the first `num` rows of Pascal's Triangle

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Observation: pas[row][col] = pas[row-1][col-1] + pas[row-1][col]

Example – Pascal's Triangle

Spec: Create the first `num` rows of Pascal's Triangle

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Observation: pas[row][col] = pas[row-1][col-1] + pas[row-1][col]

```
for (row=0; row<DIM; row++)
  for (col=0; col<DIM; col++)
    pas[row][col]=0;
for (row=0; row<DIM; row++) pas[row][0]=1;
// assert: pas is all 0's except for left column of 1's

for (row=1; row<num; row++)
  for (col=1; col<row+1; col++)
    pas[row][col] = pas[row-1][col-1] + pas[row-1][col];
```

Example – Pascal's Triangle

Spec: Create the first `num` rows of Pascal's Triangle

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Observation: pas[row][col] = pas[row-1][col-1] + pas[row-1][col]

```
for (row=0; row<DIM; row++)
  for (col=0; col<DIM; col++)
    pas[row][col]=0;
for (row=0; row<DIM; row++) pas[row][0]=1;
// assert: pas is all 0's except for left column of 1's

for (row=1; row<num; row++)
  for (col=1; col<row+1; col++)
    pas[row][col] = pas[row-1][col-1] + pas[row-1][col];
```

- Row n column k stores $\binom{n}{k}$

Passing Multidimensional Arrays To Functions

- Similar to 1-D case, but 2nd (and on) dimension sizes must be given
- Not always simple to write functions that handle arbitrary sized 2nd/3rd/... dimensions

Ex: `int foo(int image[][256], int dim1) {...}`

Example – Tic Tac Toe

Spec: Given a 3X3 array of x's and o's determine if it has a winning line for player p.

```
enum CellType {x, o, blank};      in its own scope
CellType board[3][3];             static default

bool checkBoard(CellType board[][3], cellType player) {
  for (int row = 0; row<3; row++)
    if ((board[row][0]==board[row][1]) &&
        (board[row][0]==board[row][2]) &&
        (board[row][0]==player))
      return true;
   repeat above for columns (not shown)
  if ((board[0][0]==board[1][1]) &&
      (board[1][1]==board[2][2]) &&
      (board[2][2]==player))
    return true;
   repeat above for other diagonal (not shown)
  return false;
}
```

Exercises

- Do all the 'homework problems' in this set of slides.
- Image Window operator: Given a 32-bit gray scale image, replace each pixel with the average of itself and its 8 neighbors (rounded).
- Game: Given a 8 X 8 chessboard where each position is either 'blank' or one of the chesspieces, and the row-column position of one knight, print all possible moves for that knight.
- Insertion Sort: Write the main loop for insertion sort. Don't forget to specify the invariant.
- Sorting: Given two sorted arrays, each of size SIZE, merge them into one sorted array.
- Sorting: Given an array of size SIZE where each half of the array is sorted, merge two half of arrays to create one sorted array. Do not use any extra composite types for temporary storage.
  Ex: If array is [5,7,9,11,8,10,12,14], update array to [5,7,8,9,10,11,12,14].