Arrays

int *myArray*[10];

declares an **array**:

- the **name** of the array is "*myArray*"
- the **type** of the array is "int[10]"
- the array has 10 elements
- each element is a **variable of type** int
- the elements are arranged **consecutively** in memory
- the elements have **indices** 0, …, 9
- the element with index *i* is "*myArray*[*i*]"

Array Initialisation

int *myArray*[7] = { 23, 34, 45, 56, 67, 78, 89 };
/* result: [23, 34, 45, 56, 67, 78, 89] */

int *myArray*[7] = { 23, 34, 45, 56, 67 };
/* result: [23, 34, 45, 56, 67, 0, 0] */

int *myArray*[] = { 23, 34, 45, 56, 67 };
/* result: [23, 34, 45, 56, 67] */

int *myArray*[7] = { 1 };
/* result: [1, 0, 0, 0, 0, 0, 0] */

int *myArray*[7] = { 0 };
/* result: [0, 0, 0, 0, 0, 0, 0] */

int *myArray*[7] = { };
/* Not ANSI C! */

- Automatic arrays are **not automatically initialised**!
- `static` arrays are, by default, initialised with zero values.

Character Arrays

Strings are treated as **character arrays**:

char *myString*[] = "hello";
/* result: ['h', 'e', 'l', 'l', 'o', '\0'] */

- Strings are terminated by the null character '\0'
- '\0' is the last array element of a string constant
- String generation/copying functions need **space for the terminator**

  `strdup(), fscanf(%s, … )`

  Hard to ensure for user input!
- String processing functions never read past the terminator

  `strcmp(), strncmp(), strstr()`

Passing Arguments by Value / by Reference

In general (not in C), if a variable name *v* is passed as argument to a function *f*, this can be done in two ways:

- **pass by value**: the value of *v* is passed to *f*, where it is bound to the formal parameter name.

  Usually (including in C), the formal parameter is treated as a local variable.
- **pass by reference**: *f* obtains a **reference** to *v* and turns its formal parameter name into an **alias** for *v*

C:

- in general: **pass by value**;
- **array** arguments have to be understood as **passed by reference**.
- **pass by reference** is "***faked***" using **pointers**

  — arrays are essentially treated as pointers

Pass by Reference Example 1

```
#include <stdio.h>
#define SIZE 5

void show(int ar[SIZE]) {
  int i;
  for (i = 0; i < SIZE; i++)
    printf("%7d %7d\n", i, ar[i]);
}

void update(int ar[SIZE], int i, int v) { ar[i] = v; }

int main(void) {
  int i;
  int counts[SIZE];
  for (i = 0; i < SIZE; i++) counts[i] = 10 * i;
  update (counts, 3, 333);
  show (counts);
  return 0;
}
```


Pass by Reference Example 2

```
#include <stdio.h>
#define SIZE 5

void show(int ar[], int size) {
  int i;
  for (i = 0; i < size; i++)
    printf("%7d %7d\n", i, ar[i]);
}

void swap(int ar[], int i, int j)
{ int tmp = ar[i]; ar[i] = ar[j]; ar[j] = tmp; }

int main(void) {
  int i, counts[SIZE];
  for (i = 0; i < SIZE; i++) counts[i] = 11 * i;
  swap (counts, 0, 3);
  show (counts, SIZE);
  return 0;
}
```


Arrays in `struct`s Are Passed By Value!

```
#include <stdio.h>
#define SIZE 10
struct test { int ar[SIZE]; } t;

void show(struct test t) {
  int i;
  for (i = 0; i < SIZE; i++)
    printf("%7d %7d\n", i, t.ar[i]);
}

void update(struct test t, int i, int v)
{ t.ar[i] = v; show(t); }

int main(void) {
  int i; for (i = 0; i < SIZE; i++) t.ar[i] = 10 * i;
  update(t, 2, 222);
  show(t);
  return 0;
}
```


VarArray

```
#include <stdio.h>                    /* VarArray.c */
#define SIZE 10

void printRow(int r[], int size)
{ int j; for (j=0; j<size; j++) printf("%7d ", r[j]); printf("\n"); }

void make_and_print_array(int k) {
  int ar[ k ];                /* array dimension k is a variable! */
  int j;
  for (j=0; j<k; j++) ar[j] = 101 * (j + 1);
  printRow(ar, k);
}

int main(void) { make_and_print_array(4); return 0; }
```

- In ANSI C, dimensions must be constant expressions
- In C99, variables are allowed in local (automatic) arrays

Two-Dimensional Arrays

int *myTable*[4][10];

declares a **two-dimensional array**:

- two-dimensional array can be understood as arrays of one-dimensional arrays
- the **name** of the array is "*myTable*"
- the **type** of the array is "int[4][10]"
- the array has 4 elements
- each element is an **array of type** int[10]
- the elements are arranged **consecutively** in memory
- the elements have **indices** 0, 1, 2, 3

Initialisation of Two-Dimensional Arrays

```
int myTable[2][3] = { { 1, 2, 3}, { 4, 5, 6} };
/* result:      [ [ 1, 2, 3], [ 4, 5, 6] ] */

int myTable[2][3] = { { 1      }, { 4, 5, 6} };
/* result:      [ [ 1, 0, 0], [ 4, 5, 6] ] */

int myTable[2][3] = { { 1, 2, 3} };
/* result:      [ [ 1, 2, 3], [ 0, 0, 0] ] */
```


Accessing Rows of Two-Dimensional Arrays

```
#include <stdio.h>
void printRow(int r[10]) {
  int j;
  for (j=0; j<10; j++)
    printf("%7d %7d\n", j, r[j]);
}

int main(void) {
  int myTable[4][10];
  int i,j;
  for(i=0; i<4; i++)
    for(j=0; j<10; j++)
      myTable[i][j] = 100 * i + j;
  printRow(myTable[2]);
  return 0;
}
```


Testing Initialisation of Two-Dimensional Arrays

```
#include <stdio.h>
void printRow(int r[3]) {
  int j;
  for (j=0; j<3; j++) printf("%7d ", r[j]);
  printf("\n");
}

void printTable(int t[2][3]) {
  int j;
  for (j=0; j<2; j++) printRow(t[j]);
  printf("\n");
}

int main(void) {
  int myTable[2][3] = { { 1, 2, 3} };
  printTable(myTable);
  return 0;
}
```

What is an Array?

- More precisely: **What kind of thing is the state of an array at any given time?**
- Compare:
  - char *c*; — the state of `c` is an element of $\{0, \ldots, 255\}$
  - int *k*; — the state of `k` is an element of $\{\texttt{minint}, \ldots, \texttt{maxint}\}$
- char *a*[10];
  - the state of `a` is "the state of `a[0]` … `a[9]` together"
  - the state of `a` is a **set** of **index-value-pairs**

    — i.e., a relation of type $\texttt{int} \leftrightarrow \texttt{char}$
  - more precisely, a total function from the finite domain $\{0, \ldots, 9\}$
  - which is a partial function $\texttt{int} \nrightarrow \texttt{char}$
- An **array** is a **variable for a certain kind of partial function**.

Linear Search — Textbook (adapted)

```
/* compare key to every element of array until the location is found
   or until the end of array is reached; return subscript of element
   if key or -1 if key is not found */
int linearSearch( const int array[], int key, int size )
{
  int n; /* counter */

  /* loop through array */
  for ( n = 0; n < size; ++n ) {

    if ( array[ n ] == key )
      return n; /* return location of key */

  } /* end for */

  return –1; /* key not found */

} /* end function linearSearch */
```


Linear Search — Alternative

```
#define NOT_FOUND –1    /* this is not a legal index, so it is safe. */

int linear_search(int data[], int target, int n)
{
  /* Use parameter n as local counter variable.
   * Initial value is one larger than largest index.
   */
  while ( ––n ≥ 0 )
    if ( data[n] == target )
      break;             /* found target */

  return n;  /* If while-condition failed, this is NOT_FOUND;
              * if loop terminated by break, this is the target index.
              */
}
```


Binary Search — Exercise

Design and implement a C function

int *binary_search*(int *data*[], int *target*, int *n*)

that searches the array *data* to see if the *target* value is in the array. If the *target* value is in the array, then the function returns the array index associated with the target value. The number of entries in the array is *n*, and values in the array *data* are **assumed to be stored in ascending order**.

The function should use the *binary search algorithm*:

- Look at the value of the element in the middle of the array.
- If the middle value is not the target value, decide whether the target value is in the lower or upper half of the array.
- Take the appropriate half of the array and repeat the steps until the target is found, or until it is determined that the target value does not occur in the array.

Binary Search — Exercise — Implementation

```
#define NOT_FOUND –1     /* this is not a legal index, so it is safe. */
int binary_search(int data[], int target, int n)
{
  int low = 0, high = n – 1;    /* boundaries of current search space */
  int middle;                   /* candidate index */
  /* terminate the search when either we have narrowed to a single record,
   * or we have found the target
   */
  while (low ≤ high)
  {
    middle = (low + high) / 2;
    if      (target < data[middle])  /* three-way comparison */
            high = middle –1;        /* strictly smaller than original high */
    else if (target > data[middle])
            low = middle +1;         /* strictly greater than original low */
    else    return middle;           /* search was successful */
  }
  return NOT_FOUND;   /* search was unsuccessful */
}
```


Exercise 3.1: ASCII Art — Ribbons

(a) Implement a C function *printCharArray* that prints the contents of a two-dimensional character array to the screen, each row on a separate line.

(b) Implement a C function *putRibbon* that, given a two-dimensional character array, a start height *h* and a character *c*, will place a "ribbon" of *c* values into the array that starts at height *h* and then wind **upwards** diagonally around the array.
Below, the first box contains the result of putting a ribbon of asterisks from start height 2 into a 7×30 array filled with space characters; the second box contains the result of additionally inserting a ribbon of plus characters.

```
  *      *      *      *
 *      *      *      *      *
*      *      *      *      *
      *      *      *      *
     *      *      *      *
    *      *      *      *
   *      *      *      *
```

```
  *+     *+     *+     *+
 *+     *+     *+     *+     *
*+     *+     *+     *+     *+
+     *+     *+     *+     *+
     *+     *+     *+     *+
    *+     *+     *+     *+
   *+     *+     *+     *+
```


(c) Implement a C function *putSlantedRibbon* that in addition to the arguments of *putRibbon* also accepts an integral *slant* value that indicates the steepness of the ribbon's slant as it winds around the array. This allows one to produce the contents of the box to the right with a single call to *putSlantedRibbon* with a 7×30 array filled with space characters.

```
      A      A      A      A
   A      A      A      A
A      A      A      A      A
    A      A      A      A
 A      A      A      A      A
     A      A      A      A
  A      A      A      A
```

As an additional feature, this function **must not override non-space characters** in the array. Use an auxiliary function `squeeze` to squeeze a character into its target position, pushing right all non-space content encountered at the target position and at consecutive positions — the first space character encountered will be consumed.

The same call as for the previous box, when applied after the second box of (b), produces the box to the right — observe how the "`A`" characters sometimes push only a "+" to the right, sometimes the combination "`*+`"; at the end of the third line, a "+" has been "pushed off the board".

```
  *+  A  *+  A  *+  A  *+  A
 *+A    *+A    *+A    *+A    *
A*+    A*+    A*+    A*+    A*
+   A *+   A *+   A *+   A *+
 A   *+ A   *+ A   *+ A   *+ A
    *A+    *A+    *A+    *A+
  A*+    A*+    A*+    A*+
```


printCharArray

Throughout this question, the second dimension of the two-dimensional arrays will be some fixed WIDTH; in the examples this is 30.

Implement a C function *printCharArray* that prints the contents of a two-dimensional character array to the screen, each row on a separate line.

Array width is given, and demanded to be constant by ANSI C (not by C99). Array height should be flexible, and must be supplied separately.

```
void printCharArray(char ar[][WIDTH], int height) {
  int i,j;
  for ( i = 0 ; i < height ; i++ ) {
    for ( j = 0 ; j < WIDTH ; j++ )
      printf ( "%c", ar[i][j] );
    printf( "\n" );
  }
}
```

putRibbon

Implement a C function *putRibbon* that, given a two-dimensional character array, a start height *h* and a character *c*, will place a "ribbon" of *c* values into the array that starts at height *h* and then wind **upwards** diagonally around the array. Below, the first box contains the result of putting a ribbon of asterisks from start height 2 into a 7×30 array filled with space characters; the second box contains the result of additionally inserting a ribbon of plus characters.

```
  *      *      *      *      
 *      *      *      *      *
*      *      *      *      * 
      *      *      *      *  
     *      *      *      *   
    *      *      *      *    
   *      *      *      *     
```

```
  *+     *+     *+     *+     
 *+     *+     *+     *+     *
*+     *+     *+     *+     *+
+     *+     *+     *+     *+ 
     *+     *+     *+     *+  
    *+     *+     *+     *+   
   *+     *+     *+     *+    
```

```
void putRibbon(char ar[][WIDTH], int height, int startHeight, char c)
{ int j; for ( j = 0 ; j < WIDTH ; j++ ) ar[ rem (startHeight – j, height) ][ j ] = c; }

int rem(int i, int j) /* positive remainder of integer division i / j */
{ int m = i % j; return (m < 0) ? (m + j) : m; }
```


putSlantedRibbon

Implement a C function *putSlantedRibbon* that in addition to the arguments of *putRibbon* also accepts an integral *slant* value that indicates the steepness of the ribbon's slant as it winds around the array. This allows one to produce the contents of the box to the right with a single call to *putSlantedRibbon* with a 7×30 array filled with space characters.

```
      A      A      A      A  
   A      A      A      A     
A      A      A      A      A 
    A      A      A      A    
 A      A      A      A      A
     A      A      A      A   
  A      A      A      A      
```

```
void putSlantedRibbon(char ar[][WIDTH], int height, int h, int slant, char c)
{ int j;
  for ( j = 0 ; j < WIDTH ; j++ ) {
    startHeight = rem (h, height);
    squeeze( ar[ h ], WIDTH, j, c );
    h += slant;
  }
}
```


squeeze

As an additional feature, this function **must not override non-space characters** in the array. Use an auxiliary function *squeeze* to squeeze a character into its target position, pushing right all non-space content encountered at the target position and at consecutive positions — the first space character encountered will be consumed.

The same call as for the previous box, when applied after the second box of (b), produces the box to the right — observe how the "`A`" characters sometimes push only a "+" to the right, sometimes the combination "`*+`"; at the end of the third line, a "+" has been "pushed off the board".

```
  *+  A  *+  A  *+  A  *+  A  
 *+A    *+A    *+A    *+A    *
A*+    A*+    A*+    A*+    A*
+   A *+   A *+   A *+   A *+ 
 A   *+ A   *+ A   *+ A   *+ A
    *A+    *A+    *A+    *A+  
  A*+    A*+    A*+    A*+    
```

```
/* squeeze char c into position k, pushing right all non-blank content
 * encountered at position k and consecutive positions.
 */
void squeeze(char row[], int width, int k, char c)
```


squeeze

```
/* squeeze char c into position k, pushing right all non-blank content
 * encountered at position k and consecutive positions.
 */
void squeeze(char row[], int width, int k, char c)
{
  char tmp;
  while ( k < width && row[k] ≠ ' ' ) {
    tmp = row[k];              /* swap c and row[k] */
    row[k] = c;
    c = tmp;
    k++;                       /* increment k */
  }
  if ( k < width )             /* found a blank --- insert c */
    row[k] = c;
}
```

squeeze — Recursive

```
/* squeeze char c into position k, pushing right all non-blank content
 * encountered at position k and consecutive positions.
 */
void squeeze(char row[], int width, int k, char c)
{
  if ( k ≥ width ) return;
  if (row[k] ≠ ' ' )                /* need to push right */
    squeeze(row, width, k+1, row[k]);
  row[k] = c;                        /* now free --- insert c */
}
```


`main()`

Write a `main` program that uses the above (**and other**) functions to produce as screen output the contents of the four example boxes above **in the same sequence as above**, using a **single** array of size 7×30.

```
void initCharArray(
   char ar[][WIDTH],
   int height,
   char c
 )
{
  int i,j;
  for(i = 0; i < height; i++)
    for(j = 0; j < WIDTH; j++)
      ar[i][j] = c;
}
```

```
void main() {
  char ar[ HEIGHT ][ WIDTH ];
  initCharArray(    ar, HEIGHT, ' ');
  putRibbon(        ar, HEIGHT, 2, '*');
  printCharArray(   ar, HEIGHT);
  putRibbon(        ar, HEIGHT, 3, '+');
  printCharArray(   ar, HEIGHT);
  initCharArray(    ar, HEIGHT, ' ');
  putSlantedRibbon( ar, HEIGHT, 2, 2, 'A');
  printCharArray(   ar, HEIGHT);
  initCharArray(    ar, HEIGHT, ' ');
  putRibbon(        ar, HEIGHT, 2, '*');
  putRibbon(        ar, HEIGHT, 3, '+');
  putSlantedRibbon( ar, HEIGHT, 2, 2, 'A');
  printCharArray(   ar, HEIGHT);
}
```