KU LEUVEN

C: an introduction

Arrays - basics

Why arrays?

```c
/*
average_grade_bruteforce.c
Averaging ten grades - storing values the hard way
taken from I. Horton: Beginning C 5th Ed
*/

#include <stdio.h>
int main(void)
{
int grade0 = 0, grade1 = 0, grade2 = 0, grade3 = 0, grade4 = 0;
int grade5 = 0, grade6 = 0, grade7 = 0, grade8 = 0, grade9 = 0;
long sum = 0L; // Sum of the grades
float average = 0.0f; // Average of the grades

// Read the ten grades to be averaged
printf("Enter the first five grades,\n");
printf("use a space or press Enter between each number.\n");
scanf("%d%d%d%d%d", & grade0, & grade1, & grade2, & grade3, & grade4);
printf("Enter the last five numbers in the same manner.\n");
scanf("%d%d%d%d%d", & grade5, & grade6, & grade7, & grade8, & grade9);

// Calculate the average
sum = grade0 + grade1 + grade2 + grade3 + grade4 +
grade5 + grade6 + grade7 + grade8 + grade9;
average = (float)sum/10.0f;
printf("\nAverage of the ten grades entered is: %.2f\n", average);
return 0;
}
```

```c
/*
average_grade_array.c
Averaging ten grades - - storing the values the easy way
taken from I. Horton: Beginning C 5th Ed
*/

#include <stdio.h>

int
main (void)
{
  int grades[10];   // Array storing 10 values
  unsigned int count = 10;  // Number of values to be read
  long sum = 0L;    // Sum of the numbers
  float average = 0.0f;   // Average of the numbers

  printf ("\nEnter the 10 grades:\n");  // Prompt for the input
// Read the ten numbers to be averaged
  for (unsigned int i = 0; i < count; ++i)
    {
      printf ("%2u> ", i + 1);
      scanf ("%d", &grades[i]); // Read a grade
      sum += grades[i];   // Add it to sum
    }

// calculate the average
  average = (float) sum / count;  // Calculate the average
  printf ("\nAverage of the ten grades entered is: %.2f\n", average);
  return 0;
}
```

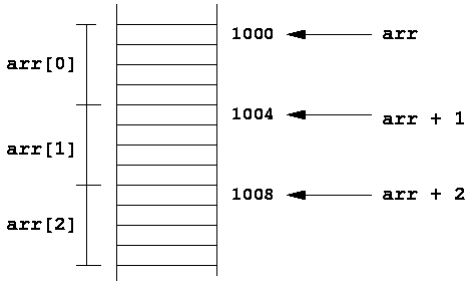
Array basics

- Array: a block of memory that holds one or more objects of a given type.
- Array: store multiple data with common characteristics
  - Same name
  - Same type
  - Accessed by specifying subscript(s) (indexing starts at index 0!)

Array basics

- Example

  int arr[10] ;



Array basics

- Declare an array:
  - Declare the **type** of elements
  - Declare the **maximum** number of elements.

```
double empty[0]; /* Error: cannot be empty */
int an_array[10]; /* allocate for 10 ints. */

a = an_array[0]; /* first element */
b = an_array[9]; /* last element */
c = an_array[10]; /* Error: but will compile */
```

- Elements of an array are stored at consecutive locations in memory (continuous memory)
  - Easy access
  - Difficult for large arrays
- Access to arrays is performed **without bounds checking**. Bounds checks must be applied explicitly by the programmer.

Array Initialisation

- An array is not initialised by default.
- Can explicitly initialise an array using an initialiser list enclosed in braces {}.

```
int days[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

- If the number of elements in the initialiser list is less than the size of the array, the remainder of an array is initialised to zero.

```
int local_array[50] = {0};
```

- If the number of elements is greater, it is an error.
- An array with an initialiser list may be sized automatically by the compiler.

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```c
#include <stdio.h>
/*
array_init.c
based on http://gribblelab.org/cbootcamp/6_Complex_Data_Types.html
*/

int main ()
{
  int grades[5] = {11, 9, 14, 15, 13};
//  int grades[5] = {11, 9, 14, 15, 13, 12};  // error?
  int grades2[5] = {[0]=1, [2]=3, [4]=5};
  int local_arr[10] = {-1};
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

  int i;

// what are the initial values?
  for (i=0; i<5; i++) {
    printf("grades[%d]=%d\n", i, grades[i]);
  }

// what are the initial values?
  for (i=0; i<5; i++) {
    printf("grades2[%d]=%d\n", i, grades2[i]);
  }

// out of the boundary?
  printf("grades[5]=%d\n", grades[5]);
  printf("grades[100]=%d\n", grades[100]);


// what are the initial values?
  for (i=0; i<10; i++) {
    printf("local_arr[%d]=%d\n", i, local_arr[i]);
  }

// what are the initial values?
  for (i=0; i<12; i++) {
    printf("days[%d]=%d\n", i, days[i]);
  }
  return 0;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_init.c -o array_init
frankvp@CRD-L-08004:.../Arrays$ ./array_init
grades[0]=11
grades[1]=9
grades[2]=14
grades[3]=15
grades[4]=13
grades2[0]=1
grades2[1]=0
grades2[2]=3
grades2[3]=0
grades2[4]=5
grades[5]=0
grades[100]=1269202072
local_arr[0]=-1
local_arr[1]=0
local_arr[2]=0
local_arr[3]=0
local_arr[4]=0
local_arr[5]=0
local_arr[6]=0
local_arr[7]=0
local_arr[8]=0
local_arr[9]=0
days[0]=31
days[1]=28
days[2]=31
days[3]=30
days[4]=31
days[5]=30
days[6]=31
days[7]=31
days[8]=30
days[9]=31
days[10]=30
days[11]=31
```

```c
#include <stdio.h>
/*
array_bounds.c
based on http://gribblelab.org/cbootcamp/6_Complex_Data_Types.html
*/

int main ()
{
  int grades[5];
  int i;

// what are the initial values?
  for (i=0; i<5; i++) {
    printf("grades[%d]=%d\n", i, grades[i]);
  }

// out of the boundary?
  printf("grades[5]=%d\n", grades[5]);
  printf("grades[10]=%d\n", grades[10]);

// assign a value
  for (i=0; i<5; i++) {
    grades[i]=i;
  }
  for (i=0; i<5; i++) {
    printf("grades[%d]=%d\n", i, grades[i]);
  }

  return 0;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_bounds.c -o array_bounds
frankvp@CRD-L-08004:.../Arrays$ ./array_bounds
grades[0]=0
grades[1]=0
grades[2]=1074737280
grades[3]=21967
grades[4]=669659408
grades[5]=32765
grades[10]=1617907891
grades[0]=0
grades[1]=1
grades[2]=2
grades[3]=3
grades[4]=4
frankvp@CRD-L-08004:.../Arrays$
```

Assigning / Getting values

- Assignment of values:

```
array_x[1] = 61;  /* 2nd element gets 61*/
array_y[2] = 1.14;
```

- Accessing values from arrays:

```
val1 = array_x[2];
val2 = array_y[0];
```

- Accessing variable array_x[n] = (n+1)th element!
- Tip: for-loops are ideal for processing array elements

KU LEUVEN

Arrays and pointers

- Arrays and Pointers are strongly related.
- Whenever an array name appears in an expression, it is automatically converted to a pointer to its first element.

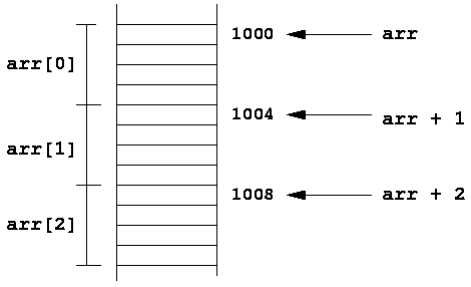
```c
#include <stdio.h>
/*
array_pointer_1.c
based on Computer programming in C for beginners
*/

int main ()
{
  int grades[5]={10, 12, 11, 16, 7};

   printf("The address contained in grades is %p \n", grades);
   printf("The address of grades[0] is %p \n", &grades[0]);

  return 0;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_pointer_1.c -o array_pointer_1
frankvp@CRD-L-08004:.../Arrays$ ./array_pointer_1
The address contained in grades is 0x7ffea01920c0
The address of grades[0] is 0x7ffea01920c0
frankvp@CRD-L-08004:.../Arrays$
```


KU LEUVEN

Arrays and pointers

- Array can be treated as a constant pointer that points to the first element in the array
- int arr[10] ;



KU LEUVEN

Pointers and Arrays are Different

- An array name is not a variable – its value cannot be changed.

```
int a1[10], a2[10];
int *pa = a1;
a1 = a2; /* Error: won't compile. */
a1++;    /* Error: won't compile. */
```

- An array name always refers to the beginning of a segment of allocated memory.
  A pointer may point anywhere (e.g., to allocated memory, to NULL, to free memory, to invalid locations).
- The size of an array is equal to the number of characters of memory allocated. The size of a pointer is just the size of the pointer-type.
- Pointers and array names may be used interchangeably for array indexing operations.

KU LEUVEN

```c
#include <stdio.h>
/*
array_pointer_2.c
based on Computer programming in C for beginners
*/

int main ()
{
  int grades[5]={10, 12, 11, 16, 7};
  int points[5];
  int * pa, * pb;
  int sg, spa;

   printf("The address contained in grades is %p \n", grades);
   printf("The address of grades[0] is %p \n", &grades[0]);

   pa = grades;
   printf("The address contained in pa is %p \n", pa);

// points = grades;  /* will this compile? */

   grades[4] = 6; /* Equivalent indexes. */
   printf("grades[4] updated %d \n", grades[4]);
   pa[4] = 6;
   printf("grades[4] updated %d \n", pa[4]);
   *(grades + 4) = 6;
   printf("grades[4] updated %d \n", *(grades+4));
   *(pa + 4) = 6;
   printf("grades[4] updated %d \n", *(pa+4));

   pb = &grades[1]; /* Equivalent addresses. */
   pb = &pa[1];
   pb = grades + 1;
   pb = pa + 1;


   sg = sizeof(grades);
   spa = sizeof(pa);
   printf("The size of grades is %d \n", sg);
   printf("The size of pa is %d \n", spa);

  return 0;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_pointer_2.c -o array_pointer_2
frankvp@CRD-L-08004:.../Arrays$ ./array_pointer_2
The address contained in grades is 0x7ffe5727f330
The address of grades[0] is 0x7ffe5727f330
The address contained in pa is 0x7ffe5727f330
grades[4] updated 6
grades[4] updated 6
grades[4] updated 6
grades[4] updated 6
The size of grades is 20
The size of pa is 8
frankvp@CRD-L-08004:.../Arrays$
```


Pointer arithmetic

- The variable name of an array is also a pointer to its first element.
  - `a == &a[0]`
  - `a[0] == *a`
- The pointer advances/retreats by that number of elements (of the type being pointed to)
  - `a+i == &a[i]`
  - `a[i] == *(a+i)`

Passing arrays to functions

- Array names are in fact pointers!
- Actually passing an array by reference, rather than by value
  - Passing the array name only to the called function (without the brackets).
  - Pass the size of the array, the calling function knows the size of the array
- The two prototypes below are exactly equivalent.

```
int count_days(int days[]);
int count_days(int *days);
```


Passing arrays to functions

```c
#include <stdio.h>
#include <stdlib.h>
/*
array_passing_1.c
based on Computer programming in C for beginners
*/
void double_it(int [], int); // prototype
int main()
{
int arr[10] = {0}, n;
for(n=0; n<10; n++)
   printf("The content of cell %d is initially %d \n", n, arr[n]);
double_it(arr, 10);
printf("\n\n");
for(n=0; n<10; n++)
   printf("The content of cell %d is now %d \n", n, arr[n]);
return 0;
}

void double_it(int a[], int i)
{
int k = 0;
a[0] = 1;
for(k=1; k<i; k++)
   a[k] = a[k-1] * 2;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_passing_1.c -o array_passing_1
frankvp@CRD-L-08004:.../Arrays$ ./array_passing_1
The content of cell 0 is initially 0
The content of cell 1 is initially 0
The content of cell 2 is initially 0
The content of cell 3 is initially 0
The content of cell 4 is initially 0
The content of cell 5 is initially 0
The content of cell 6 is initially 0
The content of cell 7 is initially 0
The content of cell 8 is initially 0
The content of cell 9 is initially 0


The content of cell 0 is now 1
The content of cell 1 is now 2
The content of cell 2 is now 4
The content of cell 3 is now 8
The content of cell 4 is now 16
The content of cell 5 is now 32
The content of cell 6 is now 64
The content of cell 7 is now 128
The content of cell 8 is now 256
The content of cell 9 is now 512
frankvp@CRD-L-08004:.../Arrays$
```

```c
/*
generate_array.c
test function to generate an array
*/

#include<stdio.h>

void generate_array(int size, int dummy[], int x);

int main() {

int i;
int dummy1[10];
int dummy2[5];

generate_array(10, dummy1, 100);
for (i=0; i<10; i++)
  printf("dummy1 - %d = %d \n", i, dummy1[i]);

generate_array(5, dummy2, 33);
for (i=0; i<5; i++)
  printf("dummy1 - %d = %d \n", i, dummy2[i]);

return 0;
}

void generate_array(int size, int dummy[], int x)
{
int i;

for (i=0; i<size; i++){
   dummy[i] = i + x;
  }
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc generate_array.c -o generate_array
frankvp@CRD-L-08004:.../Arrays$ ./generate_array
dummy1 - 0 = 100
dummy1 - 1 = 101
dummy1 - 2 = 102
dummy1 - 3 = 103
dummy1 - 4 = 104
dummy1 - 5 = 105
dummy1 - 6 = 106
dummy1 - 7 = 107
dummy1 - 8 = 108
dummy1 - 9 = 109
dummy1 - 0 = 33
dummy1 - 1 = 34
dummy1 - 2 = 35
dummy1 - 3 = 36
dummy1 - 4 = 37
frankvp@CRD-L-08004:.../Arrays$
```


Pointer passing revisited - `const`

- Declaring function parameters `const` indicates that the function promises not to change these values.

```c
#include <stdio.h>
#include <stdlib.h>
/*
array_passing_const.c

based on https://azrael.digipen.edu/~mmead/www/Courses/CS170/Const-1.html#ARGUMENTS
*/
int find_largest2(const int a[], int size);

int
main ()
{
  int arr[10] = { 0, 1, 2, 3, 4, 4, 4, 1, 1, 1 };
  int n;
  int max_array;
  const int *parr;

  max_array = find_largest2(arr,10);
  printf("largest value in array = %d \n", max_array);
  return 0;
}


int find_largest2(const int a[], int size)
{
  int i;
  int max = a[0];
//  a[0] = 0;   /* ILLEGAL: elements are const, compiler prevents it */
  for (i = 1; i < size; i++)
  {
    if (a[i] > max)
      max = a[i];
//    a[i] = 0; /* ILLEGAL: elements are const, compiler prevents it */
  }
  return max;
}
```

Passing arrays to functions

```c
/*
array_passing_2.c
http://www.cs.yale.edu/homes/aspnes/classes/223/examples/pointers/sumArray.c
*/
void double_it(int [], int); // prototype
int sumArray(int n, const int *a);

int main()
{
int arr[10] = {0};
int n;
int sum_array;
const int * parr;
// put values
parr = arr;
for(n=0; n<10; n++){
   arr[n] = n;
   printf("The content of cell %d is %d \n", n, arr[n]);}

// calculate the sum
sum_array = sumArray(10, parr);

printf("\n\n");
printf("The sum of the array elements is %d \n", sum_array);
return 0;
}

/* compute the sum of the first n elements of array a */
int
sumArray(int n, const int *a)
{
    int i;
    int sum;

    sum = 0;
    for(i = 0; i < n; i++) {
        sum += a[i];
    }

    return sum;
}
```

```
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Arrays$ gcc array_passing_2.c -Wall -o array_passing_2
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Arrays$ ./array_passing_2
The content of cell 0 is 0
The content of cell 1 is 1
The content of cell 2 is 2
The content of cell 3 is 3
The content of cell 4 is 4
The content of cell 5 is 5
The content of cell 6 is 6
The content of cell 7 is 7
The content of cell 8 is 8
The content of cell 9 is 9


The sum of the array elements is 45
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Arrays$
```

- Note the use of `const` to promise that sumArray won't modify the contents of a.

KU LEUVEN

2D arrays

- a[i][j], *not* a[i, j]
- Initiliaze:
  - **Row dominant**
  - Use {}

```
int a[3][4]={
    {0, 1, 2, 3},
    {4, 5, 6, 7},
   {8, 9, 10, 11}};
```

or

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm

KU LEUVEN

```c
/*
array_2dim.c
taken from http://gribblelab.org/cbootcamp/6_Complex_Data_Types.html
*/

#include <stdio.h>

int main ()
{
  int grades[2][2] = {1,2,3,4};  // C is row dominant!
  int i,j;
  for (i=0; i<2; i++) {
    for (j=0; j<2; j++) {
      printf("grades[%d][%d]=%d\n", i, j, grades[i][j]);
    }
  }
  return 0;
}
```

```
frankvp@CRD-L-08004:.../Arrays$ gcc array_2dim.c -o array_2dim
frankvp@CRD-L-08004:.../Arrays$ ./array_2dim
grades[0][0]=1
grades[0][1]=2
grades[1][0]=3
grades[1][1]=4
frankvp@CRD-L-08004:.../Arrays$
```


2D arrays and double pointers

- Int A[n][m]
- Consider it as equivalent to *pointers to row*
  - A[0] address of row 0
    - A[0] is an int*
  - A[1] address of row 1
- `A[i] == *(A+i)`
- `A[i][j] = *(A[i]+j) = *(*(A+i)+j)`
- A is a 2D int array, consider it as a pointer to a pointer to an integer: `int**`
  - A dereference of A[0] : *A[0]
  - the first element of row 0 or A[0][0]
  - **A = A[0][0] is an int

Matrix calculations

- multidimensional arrays to represent matrices
- better to make use of one of the pre-existing APIs for matrix algebra, rather than coding up this yourself.
- Common choices:
  - The GNU Scientific Library GSL Vectors and Matrices
  - LAPACK (and BLAS) libraries