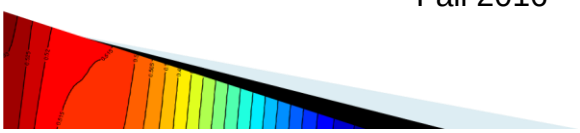
GNG 1106

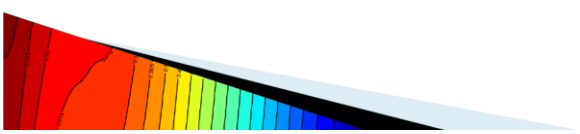
Fundamentals of Engineering Computation

Module 4 - More Arrays and Introduction to Pointers

Fall 2016



Topic 1: More on Arrays

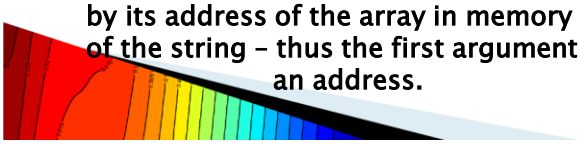


Character Strings in C

- In C, a character string in a program can be delimited by a pair of double quotes " and "
  - E.g.:

```
"This is a character String"
printf("Welcome to the University of Ottawa.\n");
```

- In C, a character string is stored in a 1D array of `char` and the **null character** is used to delimit the end of the string.
  - The null character can be obtained or assigned to a `char` using: `'\0'`
  - The name of the array used to store a string is in fact the name of the string.
  - Recall that the name of an array is also **an address** that points to the first element in the array; in this case, this is the first character in the string.
  - **In C the expression of a string enclosed in quotes is represented by its address of the array in memory that contains the characters of the string – thus the first argument passed to `printf` is an address.**



- A string may be declared and initialized in one of many ways.

E.g.:

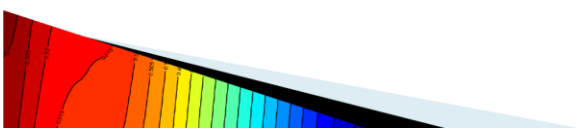
```
char hello[] = "Greetings"; /*contents */
char color[] = {'r', 'e', 'd', '\0'};
```

  - The first declaration creates an array of ten elements called `hello` and stores the characters `'G'`, `'r'`, `'e'`, `'e'`, `'t'`, `'i'`, `'n'`, `'g'`, `'s'` and `'\0'` in it.
  - The second declaration creates an array of four elements called color and stores the characters `'r'`, `'e'`, `'d'`, and `'\0'` in it.
- Note that if we use a pair of " in the assignment of a string to an array of `char`, we do not need to include the null character explicitly as C will add it to the end of the string for us.
- A string can be printed out using `printf` and the `%s` conversion specifier.

E.g.:

```
printf("%s", byePtr);
```

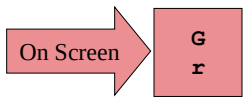
  - `printf` will stop printing out characters when it reaches the null character.
- All C standard functions for manipulating strings use the null character as the delimiter.
  - ➔ If the nul character is missing, a C standard function will continue processing the array of `char` going beyond the array limits until it hits a null character somewhere in memory!



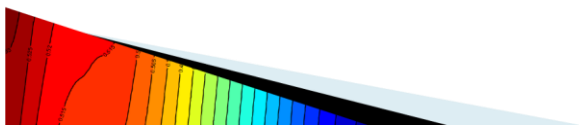
- We can read in a string from the keyboard using `scanf` and the `%s` conversion specifier.

  E.g.:
  ```
  char text_in[10];
  scanf("%s", text_in);
  ```
  - Note that there is no ampersand `&` preceding the name `text_in` in the above.
    - Recall that an array name without the `[]` is an address and in this case the address refers to the memory area where the string is to be stored.
  - `text_in` may contain at most 10 characters, including the null character; thus the user can enter at most 9 characters.
  - `scanf` will read characters until the next space, a carriage return or `EOF`.
  - `scanf` leaves the space and new line characters in the input stream.
- Individual elements of a string are of course accessed just like the elements of any array.

  E.g.:
  ```
  char hello[] = "Greetings";
  printf("%c\n", hello[0]);
  printf("%c", hello[1]);
  ```

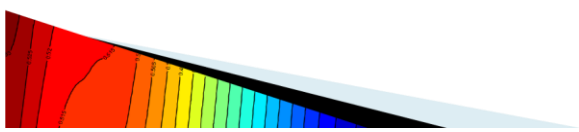


- Careful: do not confuse the nul character `'\0'` with the symbolic constant `NULL` which corresponds to the null address.

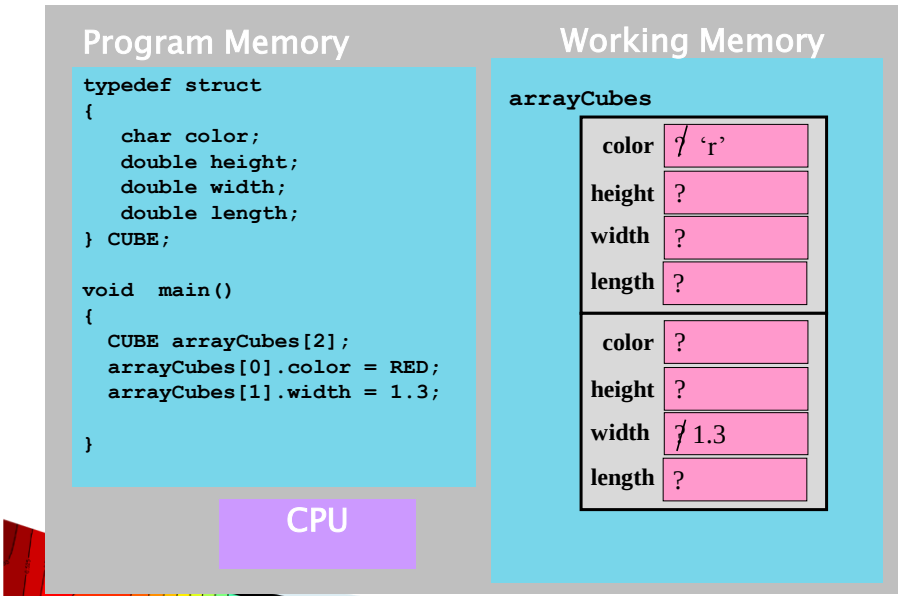


Using Arrays of Structure Variables

- Arrays of structures can also be declared (recall CUBE type):
  ```
  CUBE arrayCubes[4];  // Array of 4 cubes
  ```
- The members of an element of a structure array are also accessed using the dot operator.
  - Suppose that the structure array `arrayCubes` exists as previously declared.
    ```
    arrayCubes[0].color = RED;
    arrayCubes[1].height = 4.3;
    arrayCubes[1].width = 4.3;
    arrayCubes[1].length = 4.3;
    printf("Color of cube 0 is %c\n",
           arrayCubes[0].color);
    ```
  - Recall that the name of an array without the `[]` is the address to the first element in the array.



The Array of Structures in Memory



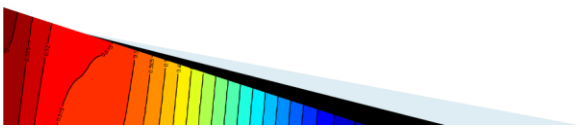
7

Topic 2: Character/String Library

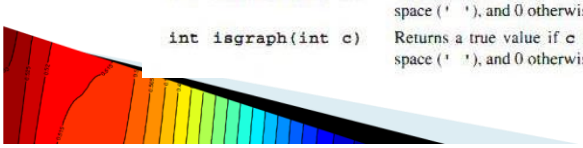
8

Standard Functions for Manipulating Characters and Strings

- There exists in C a large number of standard functions for manipulating characters and strings.
- The header file `ctype.h` contains the prototypes of a number of functions useful for manipulating or testing one character.
  - See the next slide for a summary of these functions.
- The header file `string.h` contains the prototype of functions for manipulating a string of characters.
  - These functions allow, among many, to compare strings, to search strings, and to find the length of strings.

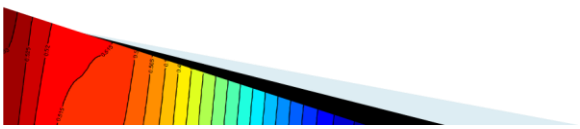


`int isdigit(int c)`	Returns a true value if `c` is a digit, and 0 (false) otherwise.
`int isalpha(int c)`	Returns a true value if `c` is a letter, and 0 otherwise.
`int isalnum(int c)`	Returns a true value if `c` is a digit or a letter, and 0 otherwise.
`int isxdigit(int c)`	Returns a true value if `c` is a hexadecimal digit character, and 0 otherwise. (See Appendix E, "Number Systems," for a detailed explanation of binary numbers, octal numbers, decimal numbers, and hexadecimal numbers.)
`int islower(int c)`	Returns a true value if `c` is a lowercase letter, and 0 otherwise.
`int isupper(int c)`	Returns a true value if `c` is an uppercase letter, and 0 otherwise.
`int tolower(int c)`	If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper(int c)`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace(int c)`	Returns a true value if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and 0 otherwise
`int iscntrl(int c)`	Returns a true value if `c` is a control character, and 0 otherwise.
`int ispunct(int c)`	Returns a true value if `c` is a printing character other than a space, a digit, or a letter, and 0 otherwise.
`int isprint(int c)`	Returns a true value if `c` is a printing character including space (`' '`), and 0 otherwise.
`int isgraph(int c)`	Returns a true value if `c` is a printing character other than space (`' '`), and 0 otherwise.



- The header file `stdlib.h` contains the prototype of functions for converting a string of digits to an integer or a real number.
  - The figure below summarizes these functions.

`double atof(const char *nPtr)`	Converts the string `nPtr` to `double`.
`int atoi(const char *nPtr)`	Converts the string `nPtr` to `int`.
`long atol(const char *nPtr)`	Converts the string `nPtr` to `long int`.
`double strtod(const char *nPtr, char **endPtr)`	Converts the string `nPtr` to `double`.
`long strtol(const char *nPtr, char **endPtr, int base)`	Converts the string `nPtr` to `long`.
`unsigned long strtoul(const char *nPtr, char **endPtr, int base)`	Converts the string `nPtr` to `unsigned long`.

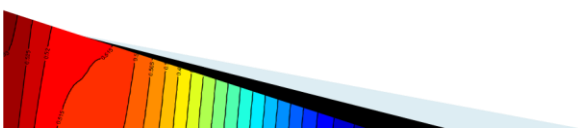


11

Finding the length of a string

`size_t strlen(const char *s);`

The `strlen` function computes the length of the string pointed to by `s`. The `strlen` function returns the number of characters that precede the terminating null character.



12

Standard Functions for Strings

```
char *strcpy(char *s1, const char *s2);
```

The `strcpy` function copies the string pointed to by `s2` (including the terminating null character) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The `strcpy` function returns the value of `s1`.

```
char *strcat(char *s1, const char *s2);
```

The `strcat` function appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined. The `strcat` function returns the value of `s1`.

```
int strcmp(const char *s1, const char *s2);
```

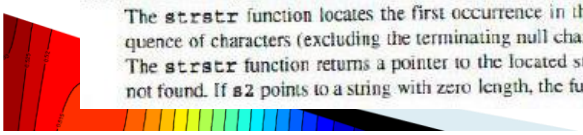
The `strcmp` function compares the string pointed to by `s1` to the string pointed to by `s2`. The `strcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

```
char *strchr(const char *s, int c);
```

The `strchr` function locates the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string. The `strchr` function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

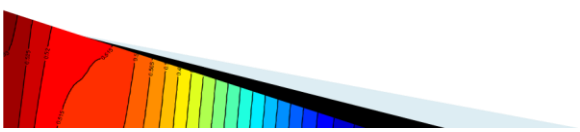
```
char *strstr(const char *s1, const char *s2);
```

The `strstr` function locates the first occurrence in the string pointed to by `s1` of the sequence of characters (excluding the terminating null character) in the string pointed to by `s2`. The `strstr` function returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length, the function returns `s1`.



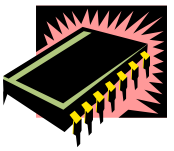
13

Topic 3: Pointers

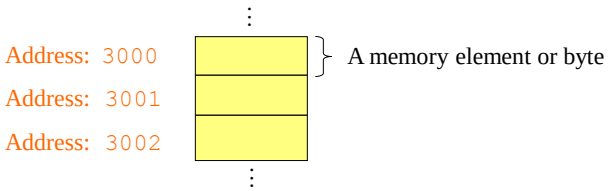


14

Addresses and Pointers



- Recall the organization of memory
- Each memory element in a computer has a unique address that specifies its position or location in memory.
  - The address of a memory element is a positive integer.
  - Contiguous memory elements have contiguous addresses:



- The value of a variable can occupy multiple bytes
  - The address of a variable is the address of the first byte that its value occupies.

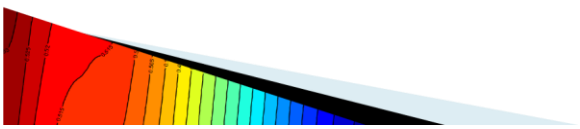
Concept of allocated memory in C

- The number of memory elements required to store **values** of variables depends on:
  - the type of the variable to be stored; i.e.: `int`, `float`..., and
  - the computer system on which the program is compiled.
- The C operator `sizeof` can be used to determine the amount of memory elements required to store a particular object.
  - The operator `sizeof` requires one argument which can be a type or variable name enclosed within parentheses.
  - The operator `sizeof` yields the number of memory elements required to store the argument.
  - If the argument is the name of an array then the total number of bytes required to store the array is obtained.

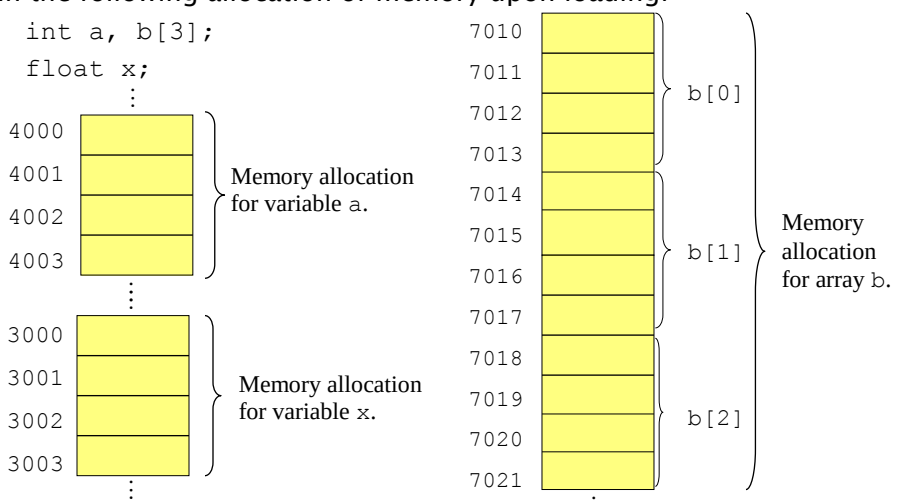
- On my PC with CodeBlocks, the following numbers are obtained from the operator `sizeof`:

E.g.:
```
int a, b[3];
double x;
sizeof(int);        → yields 4
sizeof(a);          → yields 4
sizeof(b);          → yields 12
sizeof(double);     → yields 8
sizeof(x);          → yields 8
sizeof(float);      → yields 4
```

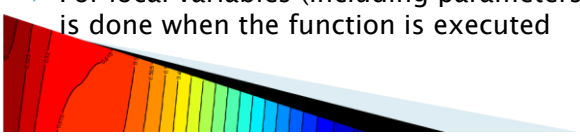
- During the compilation of a C program, the memory required to store all variables is known. At compilation time, the memory for global variables is reserved and their addresses is determined. When the program is loaded into memory prior to execution, the exact location of the global variables is known.



- E.g.: Our declarations of the following global variables could result in the following allocation of memory upon loading:

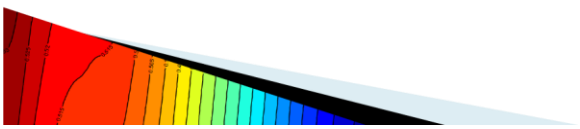


- For local variables (including parameters) the allocation of memory is done when the function is executed



Remarks

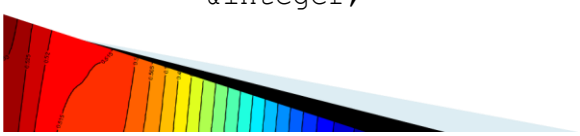
- Contiguous memory elements are always used to store a single variable.
- The elements of an array are guaranteed to be stored in contiguous memory locations.
- Variables of the same type defined in the same declaration are not necessarily stored in contiguous memory locations.
- The location of local variables (parameters) in memory cannot be known ahead of time since they are determined when the function is executed.



Concept of the Address in C

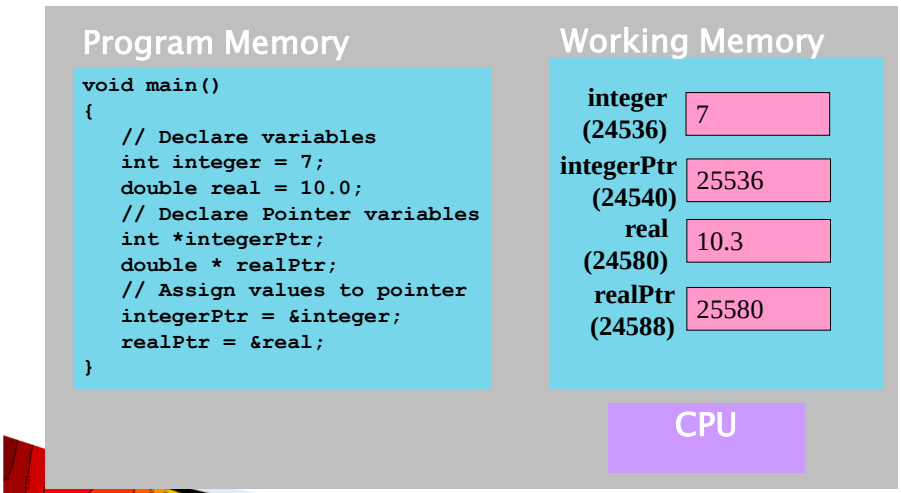
- The address of a variable in main memory is the address of the first byte of the memory used to store the variable.
- The address of a variable can be obtained in C using the address operator which is the ampersand: `&`
  - The address operator is unary and operates on the argument directly to its right.
  - The argument of the address operator must be a variable (including a structure variable)
    - The argument cannot be a symbolic constant or an expression.
  - The address operator returns the address of the first memory element used to store the argument:

    E.g.:
    ```
    int integer;
    &integer;
    ```



Pointer Variable

- The pointer variable contains an address that points to a memory location (usually the contents of a variable or array element). Notice that the pointer variable also has an address.



Pointers in C: Declaration and Initialization

- A pointer is a variable that will contain an address.
  - A *variable* contains a *value* and a **pointer** contains the **address of a variable**.
- The declaration of a pointer must specify the type of the variable to which it will point as well as the name of the pointer.
  - E.g.:

```
int integer, *integerPtr;
double real, *realPtr;
```

  - The above declarations specify that:
    `integer` is a variable of type `int`,
    `integerPtr` is a pointer to a variable of type `int`,
    `real` is a variable of type `double`,
    `realPtr` is a pointer to a variable of type `double`.
  - The asterisk `*` in a declaration signifies that the name that follows is a pointer variable.
  - The letters `Ptr` are often used in the name of a pointer to render the name more descriptive
    - It signals to the reader that the name is a pointer variable.

- A pointer to a variable of one type cannot be used to point to a variable of another type.
  - E.g.: A pointer to a variable of type `int` cannot be used to point to a variable of type `double`.
- A pointer points to nowhere in particular until it is initialized.
  - In our previous examples, `integerPtr` and `realPtr` do not yet point to the variables `integer` and `real` even if the variable and its pointer appear in the same declaration.
- To initialize a pointer, we must assign it an address, and the address is that of the variable to which we want it to point.
  - E.g.:

```
int integer, *integerPtr;
double real, *realPtr;
integer = 7;
real = 10.0;
integerPtr = &integer;
realPtr = &real;
```

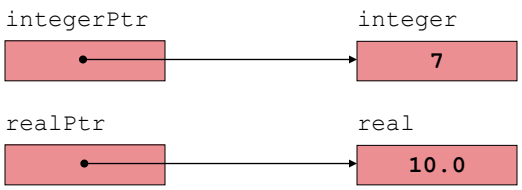
Declarations.

Variable initializations.

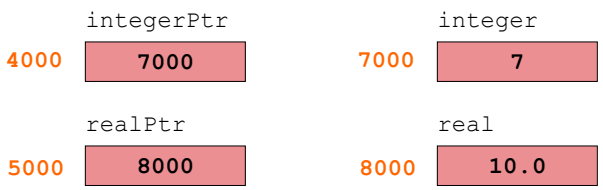
Pointer initializations.

- A variable can have more than one pointer pointing to it.

- After the previous initializations, we can visualize the memory contents as shown below:



- Assuming addresses, the organization in memory and content of our variables and pointers would be more like:



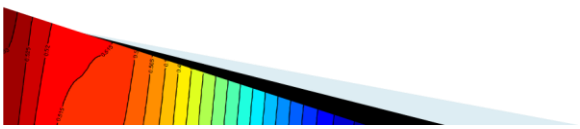
  - Thus if `integer` and `real` were located at addresses `7000` and `8000`, respectively, then our pointers `integerPtr` and `realPtr` would contain these addresses after initialization.
  - Note that pointers also have addresses since they are themselves variables.
  - Remember: it is only after allocation that actual addresses become known.

- A pointer can also be assigned the values `0` or `NULL`.
  - `NULL` is a symbolic constant (often called the null address) defined in the header file `stdio.h`
  - A pointer that contains `NULL` points to nothing.
  - E.g.:

```
integerPtr = NULL;
realPtr = 0;
```

`integerPtr` points to nothing.

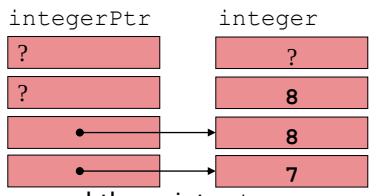
  - If the program tries to use a "NULL pointer" a fatal error occurs and the program crashes.



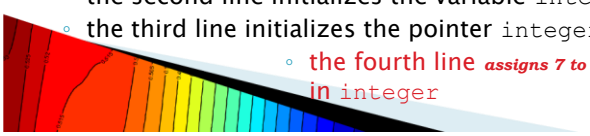
Pointers: Indirection in C

- An initialized variable contains a value.
  - We can access this value by specifying the name of the variable.
- An initialized pointer contains the address of a variable which contains a value.
  - The value in the variable can be accessed through the variable's pointer using the indirection or dereferencing operator.
- The indirection or dereferencing operator is the asterisk `*`. It is a unary operator which operates directly on the argument to its right.
  - E.g.:

```
int integer, *integerPtr;
integer = 8;
integerPtr = &integer;
*integerPtr = 7;
```

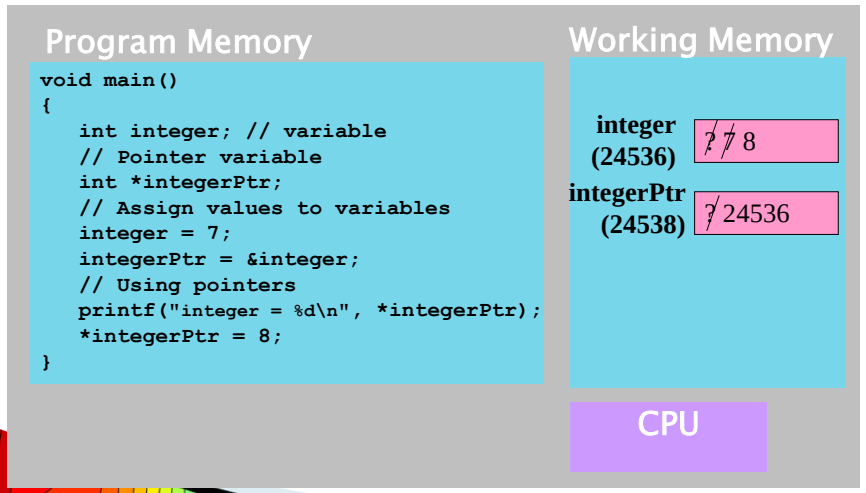


  - The first line declares the variable `integer` and the pointer `integerPtr`,
  - the second line initializes the variable `integer`,
  - the third line initializes the pointer `integerPtr`,
  - the fourth line *assigns 7 to the content pointed by integerPtr*; i.e.: in `integer`



Indirection and dereferencing

- The pointer variable contains an address that points to a memory location (usually the contents of a variable or array element).
  - Note * is used in the declaration and as the indirection operator to access the value referenced by the pointer.



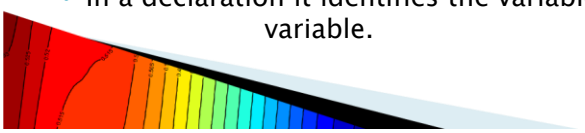
- Careful with illegal assignments:
  - E.g.:

```
int a = 7, *aPtr = &a;          Correct.
float b = 10.0, *bPtr = &b;     Correct.
&a = bPtr;                      Attempts to change the address of a.
bPtr = a;                       Attempts to store a non-address value into a pointer.
*aPtr = bPtr;                   Attempts to store an address into a variable of type int.
*aPtr = &a;                     Attempts to store an address into a variable of type int.
aPtr = *bPtr;                   Attempts to store a non-address value into a pointer.
```

- The following table summarizes in order of precedence the associativity of all the C operators that we have seen to date.

`() [] .`	left to right	
`- ! (type) * & sizeof`	right to left	unary
`* / %`	left to right	binary
`+ -`	left to right	binary
`< <= > >=`	left to right	binary
`== !=`	left to right	binary
`&&`	left to right	binary
`||`	left to right	binary
`=`	right to left	binary

- Careful with the three distinct roles that the asterisk `*` performs in C.
  - As a binary operator it is the multiplication operator.
  - As a unary operator it is the dereferencing operator (in red).
  - In a declaration it identifies the variable name as being a pointer variable.

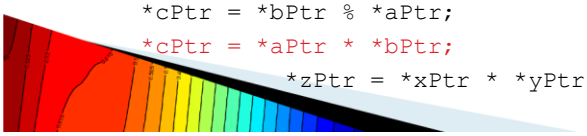


29

Arithmetic Expressions With Dereferenced Pointers

- Using the indirection operator, we can write arithmetic expressions that are similar to those written using variables (see Module 2 of the course notes).
  - E.g.:

```
int a=3, b=4, c, *aPtr=&a, *bPtr=&b, *cPtr=&c;
Double x=1.0, y=2.5, z, *xPtr=&x, *yPtr=&y, *zPtr=&z;
*cPtr = *aPtr + *bPtr;                  → c  7
*zPtr = *xPtr + *yPtr;                  → z  3.5
*zPtr = *aPtr + *bPtr;                  → z  7.0
*zPtr = *xPtr + *aPtr;                  → z  4.0
*cPtr = *xPtr + *yPtr;                  → c  3      Loss of info!
*cPtr = *xPtr + *aPtr;                  → c  4
*zPtr = *aPtr / *bPtr;                  → z  0.0
*zPtr = (float) *aPtr / *bPtr;          → z  0.75
*cPtr = *aPtr % *bPtr;                  → c  3
*cPtr = *bPtr % *aPtr;                  → c  1
*cPtr = *aPtr * *bPtr;                  → c  12
*zPtr = *xPtr * *yPtr;                  → z  2.5
```



30

Pointers and Arrays

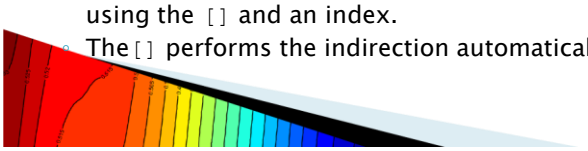
- Pointers and arrays are intimately related in C.
  - Any element in an array may be accessed via a pointer with indexing.
- The name of an array without the `[]` is in fact the address to the first element in the array.
  - E.g.:

```
int b[5], *bPtr;
bPtr = b;   // Note that the & is not used here
bPtr = &b[0];
```

  - The above assignments are equivalent: they both assign the address of the first element of array `b` to the pointer `bPtr`.
  - Note that the assignment such as `b = &a;` is not allowed. Why?
- It is possible to use indexing with a pointer to access individual array elements.
  - Based on the previous example:

```
bPtr[2] = 1;                    //Stores 1 in b[2]
```

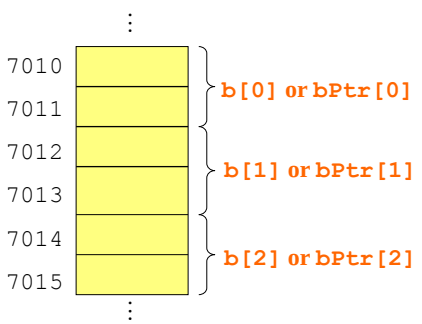
  - We do not use the indirection operator `*` when accessing array elements using the `[]` and an index.
  - The `[]` performs the indirection automatically.

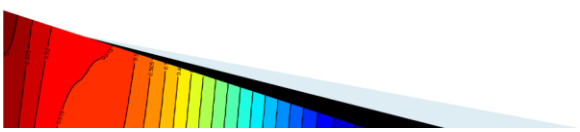


- E.g.:
  - Assume that the following declaration results in array `b` being stored from address `7010` onwards:

```
int b[3], *bPtr;
bPtr = &b[0];
```

  - The array name `b` or the pointer `bPtr` followed by an integer index placed between `[]` can be used to access an element of the array.





Using Pointers to Structure Variables

- It is possible to define pointers to structure variables as follows:

```
CUBE cube1;
CUBE *cubePtr;     // pointer to a cube structure var.
cubePtr = &cube1;
```

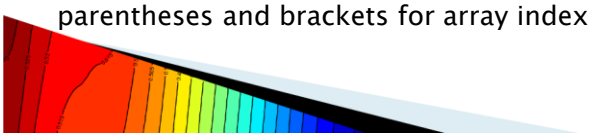
- The structure pointer operator, also called the arrow operator (->), is used with a pointer to a structure variable in order to access the structure variable's members.
  - The arrow operator is constructed from the minus sign - followed by the greater than sign > with no intervening spaces

```
cubePtr->color = RED;
cubePtr->height = 1.2;
cubePtr->width = 2.5;
cubePtr->length = 4.2;
```

Initializes the structure variable cube1

```
printf("The cube has a width of %f\n",
       cubePtr->width);
```

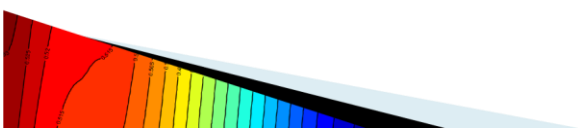
- The dot and arrow operators have the same precedence as parentheses and brackets for array indexing.



33

- The following table summarizes in order of precedence the associativity of all the C operators that we have seen to date.

`() [] -> .`	left to right	
`- ! ++ -- (type) * & sizeof`	right to left	unary
`* / %`	left to right	binary
`+ -`	left to right	binary
`< <= > >=`	left to right	binary
`== !=`	left to right	binary
`&&`	left to right	binary
`||`	left to right	binary
`= += -= *= \= %=`	right to left	binary



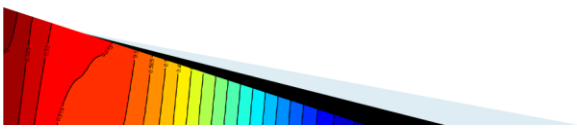
34

Review of the Passage of Arguments to a Function

- We have seen that C uses pass by value in passing argument values to function parameters:
  - In the case of simple variables and structure variables, the content (i.e. value) of the variable in the argument of the function call is copied to the called function parameter (the parameter is a local variable created in the called function's working memory).
- What about adding the address operator & to an argument as seen with `scanf`?
  - It is the address of the variable that is passed as the argument value – this can be viewed as a pass by reference.

    What kind of parameter in the called function is used to store this address?
  - This reference gives the called function access the local variable in the calling function,

    Recall that `scanf` copies the value read from the keyboard into the local variable of the calling function.



Passing structures to functions

Program Memory

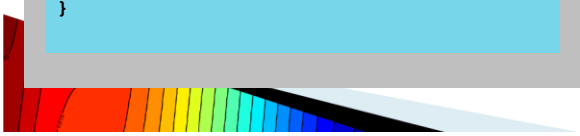
```
typedef struct
{
   char[20] color;
   double height;
   double width;
   double length;
} CUBE;

void  main()
{
  CUBE cube1;
  scanf("%s", cube1.color);
  fflush(stdin);
  scanf("%f", &cube1.height);
  scanf("%f", &cube1.width);
  scanf("%f", &cube1.length);
  print1Cube(cube1);
  print2Cube(&cube1);
}
```

Working Memory

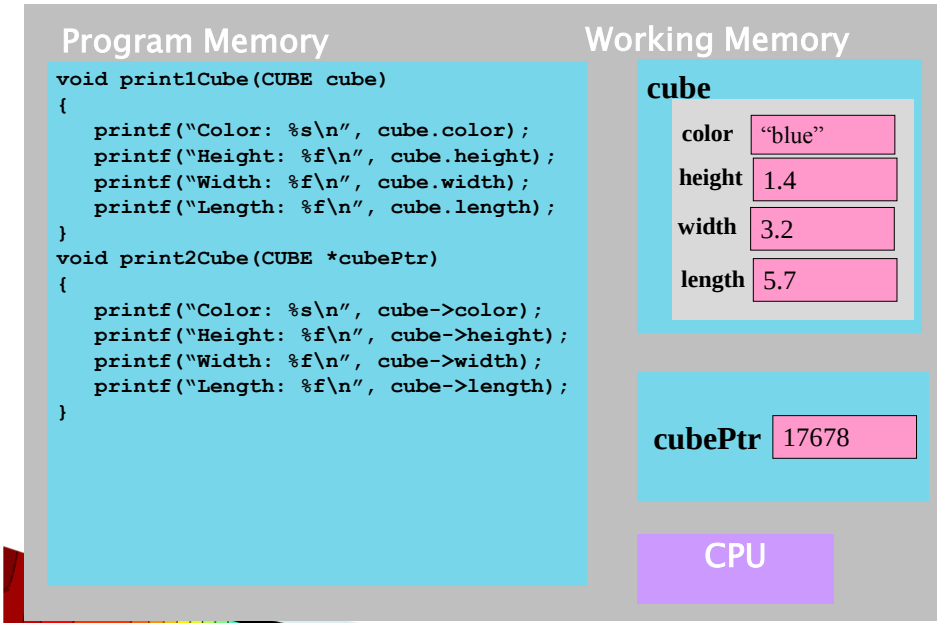
cube1 (17678)

color	"blue"
height	1.4
width	3.2
length	5.7

CPU



Passing structures to functions – continued



Passing structures to functions – continued

- As we have seen that a structure variable is passed to a function by value just like a simple variable
  - See the function print1Cube in the previous example
- Structure variables can also be passed to functions by reference
  - The address operator, &, is added to the structure variable name in the called function argument so that its address is passed.
  - The called function parameter that receives the address is a pointer variable and contains an address to a structure variable
- In the example shown in the previous slides, which method would be preferable to have a function fill in a structure: pass by value or pass by reference?
  - Passing a reference to a structure is also a practical way to allow a called function to return multiple values to a calling function since the called function can modify all members of the structure variable that exists in the working memory of the calling function.

Passing Arrays as Arguments to C Functions

Passing One Dimensional Arrays to Functions

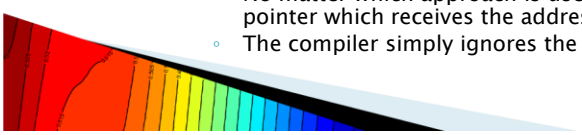
- The one dimensional array is passed to a function simply by citing the name of the array in the call to the function.
  - The array name is cited without `[]`.
  - The number of elements can be included in the call if the size of the array is required by the function.
  - The name of the array is a reference to the array (its address).

E.g.:
```
int temp[5]={0};
        ⋮
sum_array_1D(5, temp);
```
← Function call in `main()`.

- The definition of a function that receives the reference to a 1D array must specify a pointer in its parameter list to receive the reference.
  - The passage of an array can also be indicated by adding `[]` after the parameter name used to access the array in the called function.
  - The brackets can be empty or may contain the number of elements in the array.

E.g.:
```
void sum_array_1D(int dim, int array_1D[])
```
or:
```
void sum_array_1D(int dim, int array_1D[5])
```
or:
```
void sum_array_1D(int dim, int *array_1D)
```

  - No matter which approach is used, the parameter, array_1D, is a pointer which receives the address of the array.
  - The compiler simply ignores the value in the brackets if provided.



- The prototype of the function must also specify that an array will be received through its parameter list; this is stated in a manner similar to that found in the function header.

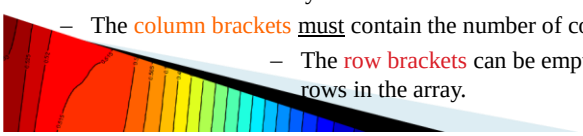
E.g.:
```
void sum_array_1D(int, int []);
```
or:
```
void sum_array_1D(int, int [5]);
```
or:
```
void sum_array_1D(int, int *);
```

Passing Two Dimensional Arrays to Functions

- A reference to a two dimensional array is passed to a function simply by citing the name of the array in the call to the function.
  - The array name is cited without `[][]`.
  - The number of rows and columns can be included in the call if required by the function.
  - The name of the matrix is a reference to the matrix.

E.g.:
```
int a[3][4]={0};
      ⋮
sum_array_2D(3, 4, a);
```

- The definition of a function that receives a two dimensional array must specify that a two dimensional array will be received through its parameter list.
  - The passage of a two dimensional array is indicated by adding `[][4]` after the name used to access the array in the function.
  - The column brackets must contain the number of columns in the array.
    - The row brackets can be empty or may contain the number of rows in the array.



E.g.: `void sum_array_2D(int rows, int cols, int array_2D[][4])`

or: `void sum_array_2D(int rows, int cols, int array_2D[3][4])`

or: `void sum_array_2D(int rows, int cols, int array_2D[][cols])`

– Note that the third option allows the 2D array referenced to be of any dimension.

- The prototype of the function must also specify that a two dimensional array will be received through its parameter list; this is stated in a manner similar to the function definition. Note the third case where the column dimension is defined by parameter.

E.g.: `void sum_array_2D(int, int, int [][4]);`

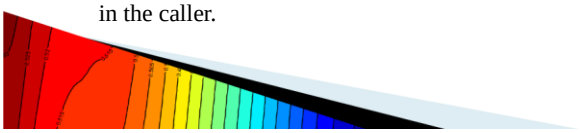
or: `void sum_array_2D(int, int, int [3][4],);`

or: `void sum_array_2D(int, int cols, int [][cols]);`

On the Passage of Array Arguments to a Function

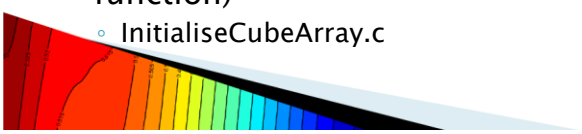
- In C, an array is passed by reference to a function and not by value as is done with variables.
  - ➔ An array that is modified in a function will appear modified in the caller as well.

- In C, an array element is passed by value to a function as though it were a variable.
  - ➔ The array element can be modified in the function and this modification will not appear in the caller.



Examples

- Initializing arrays using a function.
  - initialise1DArray.c
  - initialise2DArray.c
- Working with strings
  - stringDemo.c
- Initializing a structure using a function
  - InitialiseCubeType.c
- Working with array of structures (passing to a function)
  - InitialiseCubeArray.c



Next Module

- Topic 1: Decision Instructions
- Topic 2: Loop Instructions