Global variables

- In general, you should avoid using global variables as much as possible!
  - they make a program harder to maintain, because they increase complexity
  - create potential for conflicts between modules
  - the only advantage of global variables is that they produce faster code
- There are two types of declarations, namely, ***definition and allusion***.
- An **allusion** looks just like a definition, but instead of allocating memory for a variable, it informs the compiler that a variable of the specified type exists but is defined elsewhere.
  - extern int j;
  - The extern keyword tells the compiler that the variables are defined elsewhere.
- Whenever you want to use global variables defined in another file you need to declare them with allusions.

The ***register*** specifier

- The ***register*** keyword enables you to help the compiler by giving it suggestions about which variables should be kept in registers (so, on CPU).
  - it is only a hint, not a directive, so ***compiler is free to ignore it!***
  - The behavior is implementation dependent.
- Since a variable declared with register might never be assigned a memory address, ***it is illegal to take address of a register variable.***
- A typical case to use register is when you use a counter in a loop.

```
int strlen ( register char *p)  {

        register int len=0;
        while(*p++)  {
                len++;
        }
        return len;
}
```

Remember that: registers are not addressable! They are not on the memory, but on the CPU

Storage classes summary

- **There are 4 storage-class specifiers**
- **auto**
  - Redundant and rarely used.
- **static**
  - In declarations within a function, static causes variables to have fixed duration. For variables declared outside a function, the static keyword gives the variable file scope.
- **extern**
  - For variables declared within a function, it signifies a global allusion. For declarations outside of a function, extern denotes a global definition.
- **register**
  - It makes the variable automatic but also passes a hint to the compiler to store the variable in a register whenever possible.

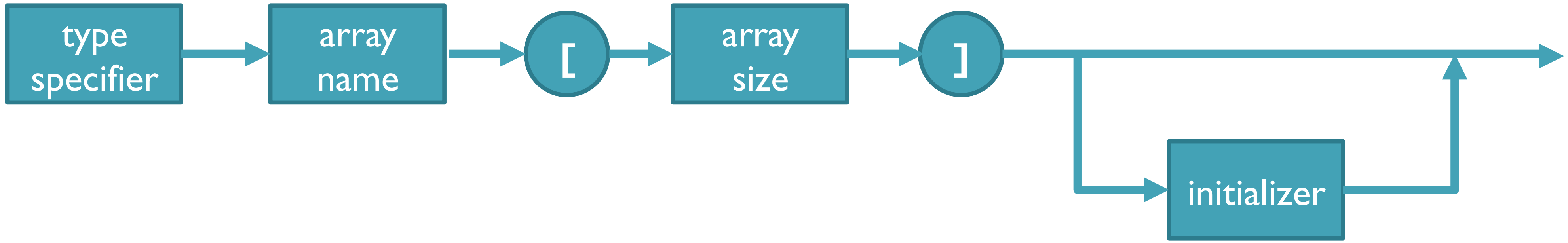
- **There are 2 storage-class modifiers**
- **const**
  - The const specifier guarantees that you can NOT change the value of the variable.
- **volatile**
  - The volatile specifier causes the compiler to turn off certain optimizations. Useful for device registers and other data segments that can change without the compiler's knowledge.
  - A variable should be declared volatile whenever its value could change unexpectedly. In practice, only 3 types of variables could change:
    - 1. Memory-mapped peripheral registers
    - 2. Global variables modified by an interrupt service routine
    - 3. Global variables accessed by multiple tasks within a multi-threaded application
  - Volatile tells the compiler not to optimize anything that has to do with the volatile variable.
  - There is only one reason to use it: When you interface with hardware. (Memory-mapped peripheral registers)
  - Another use for volatile is signal handlers.

Pointers and arrays

outline

- Declaration
- How arrays stored in memory
- Initializing arrays
- Accessing array elements through pointers
- Examples
- Strings
- Multi-dimensional arrays

declaration



int dailyTemp[365];
dailyTemp[0] = 38;
dailyTemp[1] = 43;
…

- **subscripts begin at 0, not 1 !**

```c
#include <stdio.h>
#define DAYS_IN_YEAR 365
int main () {
    int j, sum = 0;
    int daily_temp[DAYS_IN_YEAR];

    /* Assign some values to the daily_temp array here. */

    for(j=0; j<DAYS_IN_YEAR; j++)
        sum += daily_temp[j];
    printf("The average temperature for this year is: %d. \n", sum/DAYS_IN_YEAR);
    return 0;
}
```

How arrays stored in memory

```
int ar[5]; /* declaration */
ar[0] = 15;
ar[1] = 17;
ar[3] = ar[0] + ar[1];
```

- **<u>Note that ar[2] and ar[4] have undefined values!</u>**
    - the contents of these memory locations are whatever left over from the previous program execution

Element	Address	Contents
	0x0FFC	
ar[0]	0x1000	15
ar[1]	0x1004	17
ar[2]	0x1008	undefined
ar[3]	0x100C	32
ar[4]	0x1010	undefined

4 bytes
Try sizeof(ar)

Initializing arrays

- It is incorrect to enter more initialization values than the number of elements in the array
- If you enter fewer initialization values than elements, the remaining elements initialized to zero.
- <u>**Note that 3.5 is converted to the integer value 3!**</u>
- When you enter initial values, you may omit the array size
  - the compiler automatically figures out how many elements are in the array…

```
int a_ar[5];
int b_ar[5] = {1, 2, 3.5, 4, 5};
int c_ar[5] = {1, 2, 3};

char d_ar[] = {'a', 'b', 'c', 'd'};
```

Examples - Bubble sort

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Examples - Bubble sort

```
#include <stdio.h>
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1]) {
              temp=arr[j];
              arr[j]=arr[j+1];
              arr[j+1]=temp;
           }
}
```

Examples - Selection sort

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

```
#include <stdio.h>
// A function to implement selection sort
void selectionSort(int arr[], int n)
{
   int i, j, min_idx, temp;
   // 1-by-1 move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
          min_idx = j;

      // Swap the found min element with the 1st element
      temp=arr[min_idx];
      arr[min_idx]=arr[i];
       arr[i]=temp;
    }
}
```

Accessing array elements through pointers

short ar[4];

short *p;

p = & ar[0]; // assigns the address of element 0 to p. (same with p=ar)

- p = ar; **<u>is same as above assignment!</u>**
- *(p+3) refers to the same memory content as ar[3]

float ar[5], *p;

```
p = ar ;          // legal
ar = p;           // illegal
&p = ar;          // illegal
ar++;             // illegal
ar[1] = *(p+3);   // legal
p++;              // legal
```

Pointer arithmetic

- C allows you to add and subtract integers to and from pointers:
  - p+3 // means: 3 objects after the object that *p* points to. This operation generates a new address value. But compiler does not simply add p with “3”, it multiples the 3 with the size of the object that *p* points to
- Suppose p points to a float var, located at the address 1000. So, p+3 would be the address 1000+3object*4byte=1012.
- What would be p+3, if p pointed to a char?
- Subtraction: &a[3]-&a[0] is 3 but &a[0]-&a[3] is -3.
- Examples:
  - long *p1, *p2; int j; char *p3;
  - p2=p1+4; //legal
  - j=p2-p1; // legal, j will be 4
  - j=p1-p2; // legal, j will be -4
  - p1=p2-2; // legal, since both of them points the same data type
  - p3=p1-1; // ILLEGAL, they point different data types.
  - j=p1-p3; // ILLEGAL, they point different data types.

Null pointer

- A null pointer is guaranteed not to point a valid object.
- A null pointer is assigned to integer value 0:
  - char *p;
  - p = 0; // makes p a null pointer. There is no need to cast the int to the pointer type, since it is 0.
- Null pointer is useful in while statements:
  - while (p){
  - …
  - // iterate until p is a null pointer (0-valued pointer will cause to get a FALSE value and break the loop, otherwise it'll be TRUE and go on iteration
  - …
  - }
- The use of null pointers is mainly in applications using arrays of pointers.

Passing pointers as function arguments

- void clear(int *p){
-   *p=0; // store a 0 at address p.
- }
- main(){
-   int s[3]={1,2,3};
-   clear(&s[1]);
-   return 0;
- }

- // s=> 1,0,3

- void clear(long *p){
-   *p=0; // store a 0 at address p.
- }
- main(){
-   short s[3]={1,2,3};
-   clear(&s[1]);
-   return 0;
- }

- // s=> 1,0,0 => why?

Passing arrays as function arguments

- main () {
-   extern float f();
-   float x, farray[5];
-   …
-   x=func(farray); // same as func(&farray[0])
-   …
- }

- Send the array with its size information:
- void foo(float farray[], int farray_size) {
-   …
- }
- Call the function from the main function:
- foo(farray, sizeof(farray)/sizeof(farray[0]));

- float func(float *ar){
-   …
- }
- Or:
- float func(float ar[]){
-   …
- }
- Or:
- float func(float ar[6]){
-   …
- }

Strings - declaring and initializing strings

- A string is an array of characters terminated by a null character.
  - **null character is a character with a numeric value of 0**
  - it is represented in C by the escape sequence '\0'

- A string constant is any series of characters enclosed in double quotes
  - it has datatype of array of char and each character in the string takes up one byte!
  - compiler automatically appends a *null character* to designate the end of the string.

- char str[] = "some text";
- char str[10] = "yes";
- **char str[3] = "four"**
- **char str[4] = "four"**

// some compilers do not include the null character within the string size, some of them include it. Both are OK for the ANSI standard.

- char *ptr = "more text" ;

string assignments

```
main ()  {
    char array[10];
    char *ptr1="10 spaces";
    char *ptr2;
    array = "not OK";   // can NOT assign to an address !
    array[5] = 'A';     // OK
    ptr1[5] = 'B';      // OK in old compilers BUT, ptr1 is a string literal and
                        we can't  change its value, so, this will kill the program.
        you can't modify a string literal. You're going to have to return a new string.
    ptr1="OK";          // same with above explanation
    ptr1[5]='C';        // questionable due to the prior assignment
    *ptr2 = "not OK";   // type mismatch ERROR
    ptr2="OK";
}
```

strings vs. chars

char ch = ‘a’; // one byte is allocated for ‘a’

*p = ‘a’; // OK

p = ‘a’; // Illegal, p is a pointer, not a char

char *p = “a”; // two bytes allocated for “a” (the 2nd one is ‘/0’)

*p = “a”; // INCORRECT

p = “a”; // OK

*p = “string”; // INCORRECT

p = “string”; // OK

reading & writing strings

```c
#include <stdio.h>

#define MAX_CHAR 80

int main ( void )  {

        char str[MAX_CHAR];
        int i;

        printf("Enter a string :");
        scanf("%s", str);

        for(i=0;i<10;i++)  {
                printf("%s\n", str);
        }
        return 0;
}
```

- You can read strings with ***scanf()*** function.
  - the data argument should be a pointer to an array of characters ***that is long enough to store*** the input string.
  - after reading input characters scanf() automatically appends a null character to make it a proper string
- You can write strings with ***printf()*** function.
  - the data argument should be a pointer to a null terminated array of characters

string length function

- We test each element of array, one by one, until we reach the null character.
  - null char has a value of zero, making the while condition **false**
  - any other value of str[i] makes the while condition **true**
  - once the null character is reached, we exit the while loop and return *i*, which is the last subscript value

```
int strlen ( char *str)  {

        int i=0;

        while(str[i])  {
                i++;
        }

        return i;
}
```

string copy function

```
void strcpy ( char s1[], char s2[])  {

        int i;

        for(i=0; s1[i]; ++i)
                s2[i] = s[i];
        s2[++i] = '\0';
}
```

```
void strcpy ( char *s1, char *s2)  {

        int i;

        for(i=0; *(s+i); ++i)
                *(s2+i) = *(s1+i);
        s2[++i]= '\0';
}
```

```
void strcpy ( char *s1, char *s2)  {

        while(*s++ = *s1++);

}
```

strcpy()	Copies a string to an array.
strncpy()	Copies a portion of a string to an array.
strcat()	Appends one string to another.
strncat()	Copies a portion of one string to another.
strcmp()	Compares two strings.
strncmp()	Compares two strings up to a specified number of characters.
strchr()	Finds the first occurrence of a specified character in a string.
strcoll()	Compares two strings based on an implementation-defined collating sequence.
strcspn()	Computes the length of a string that does not contain specified characters.
strerror()	Maps an error number with a textual error message.
strlen()	Computes the length of a string.
strpbrk()	Finds the first occurrence of any specified characters in a string.
strrchr()	Finds the last occurrence of any specified characters in a string.
strspn()	Computes the length of a string that contains only specified characters.
strstr()	Finds the first occurrence of one string embedded in another.
strtok()	Breaks a string into a sequence of tokens.
strxfrm()	Transforms a string so that it is a suitable as an argument to *strcmp()*.

Table 6-1. String Functions in the Standard Library. See Appendix A for a more complete description of these routines.

other string functions

Pattern matching example

- Write a program that
  - gets two strings from the user
  - search the first string for an occurrence of the second string
  - if it is successful
    - return byte position of the occurrence
  - otherwise
    - return -1

- Return the position of str2 in str1; if not found then return -1.

```
#include <stdio.h>

/* Return the position of str2 in str1; -1 if not
 * found.
 */
int pat_match( str1, str2 )
char str1[], str2[];
{
  int j, k;

  for (j=0; j < strlen(str1); ++j)
  {
/*  test str1[j] with each character in str2[].  If
 *  equal, get next char in str1[].  Exit loop if we
 *  get to end of str1[], or if chars are equal.
 */
    for (k=0; (k < strlen(str2) && (str2[k] ==
         str1[k+j])); k++);

/*  Check to see if loop ended because we arrived at
 *  end of str2.  If so, strings must be equal.
 */
    if (k == strlen( str2 ))
      return j;
  }
  return -1;
}
```

multi-dimensional arrays

- In the following, ar is a 3-element array of 5-element arrays

  int ar[3][5];

- In the following, x is a 3-element array of 4-elemet arrays of 5-element arrays

  char x[3][4][5];

- the array reference

  ar[1][2]

- is interpreted as

  *(ar[1]+2)

- which is further expanded to

  ((ar+1)+2)

initialization of multi-dimensional arrays

```
int exap[5][3] = { {  1, 2, 3 },
                      { 4 },
                      { 5, 6, 7 } };
```

1	2	3
4	0	0
5	6	7
0	0	0
0	0	0

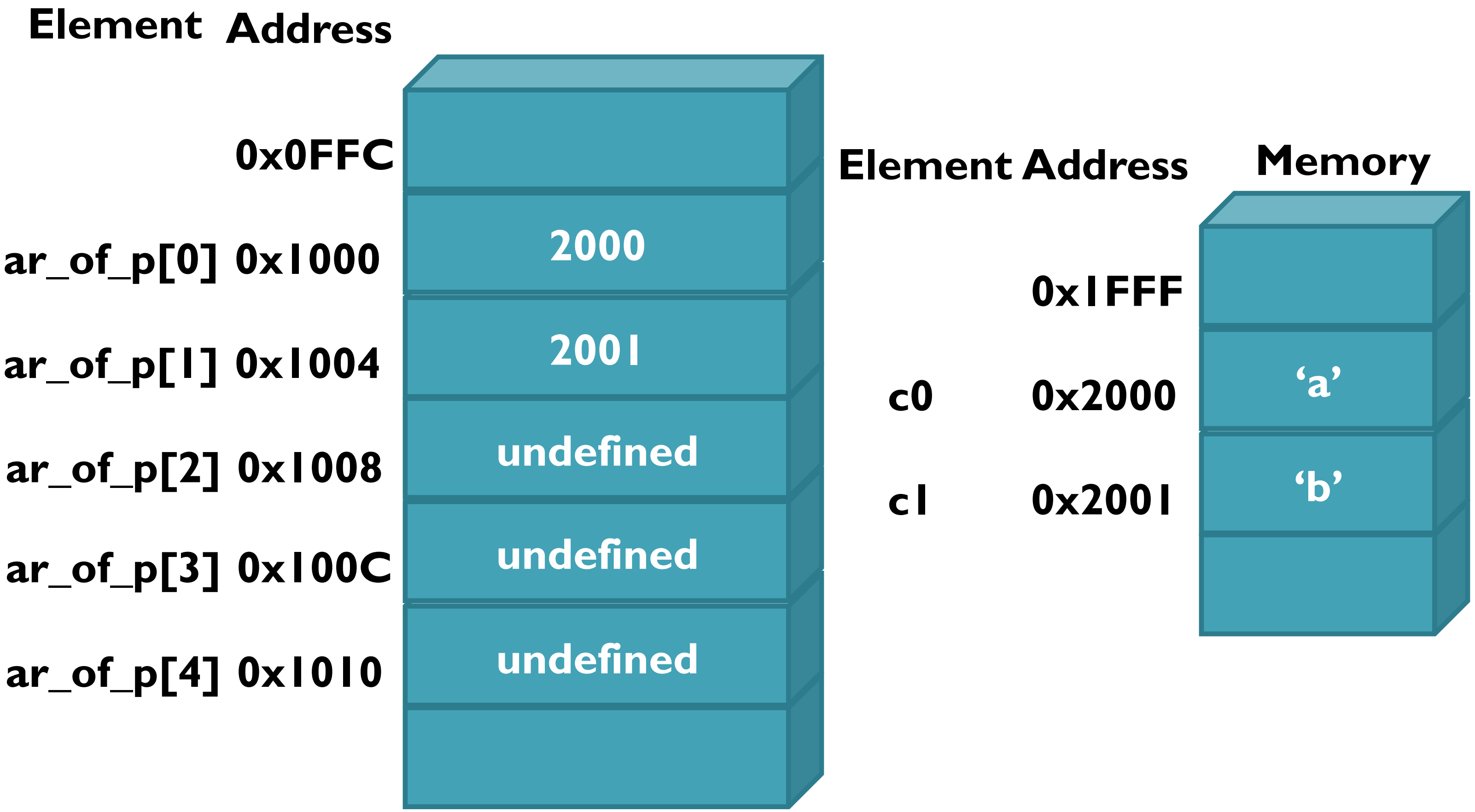
```
int exap[5][3] = {  1, 2, 3,
                    4,
                    5, 6, 7 };
```

1	2	3
4	5	6
7	0	0
0	0	0
0	0	0

array of pointers

```
char *ar_of_p[5];
char c0 = 'a';
char c1 = 'b';

ar_of_p[0] = &c0;
ar_of_p[1] = &c1;
```



pointers to pointers

int r = 5;	declares ***r*** to be an int
int *q = &r;	declares ***q*** to be a pointer to an int
int **p = &q;	declares ***p*** to be a pointer to a pointer to an int
r = 10;	Direct assignment
*q = 10;	Assignment with one indirection
**p = 10;	Assignment with two indirections