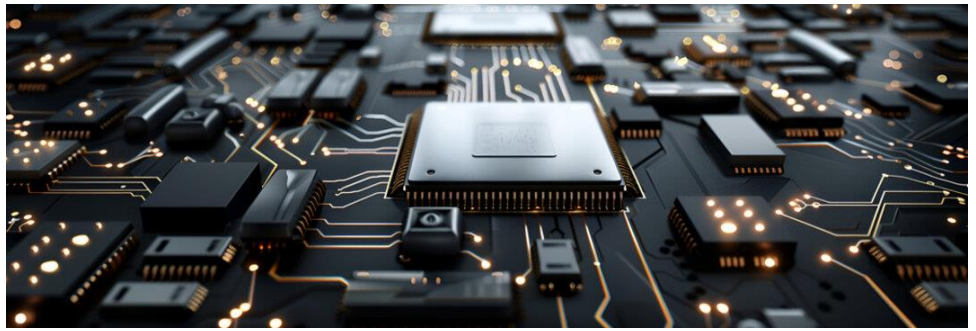
EECS 2032: Introduction to Embedded Systems



Course Instructor

Sunila Akbar

Lecture 5

Recap

- BASH - Concluded
- Programming with C
  - Introduction
  - Reading/Writing (contd.)
  - Variables and Data Types
  - Number Systems in C
  - Basic Arithmetic Operations
  - Boolean Expressions
  - Pre and Post Operators
  - `while` loop for Lab 4

Topics to be Covered

- Bitwise Operators
- Bit Shifting
- Compound Assignments
- Precedence
- Arrays
- Strings

- Control Flow
  - `if-else, else-if, switch, for and while loops, break, cotinue`
- ❑ Pointers
- ❑ Functions

Bit-wise Operators

- Used to perform operations at the bit level
- Work directly on the binary representation of data
- Typically used for low-level programming, such as working with hardware, graphics, or optimizing performance

Operator	Function	Example
`&`	Performs bitwise logical AND operation	`0101 & 0011 = 0001`
`|`	Performs bitwise logical OR operation	`0101 | 0011 = 0111`
`^`	Performs bitwise logical XOR (exclusive OR) operation	`0101 ^ 0011 = 0110`
`~`	A unary operator that inverts all bits of operand	`~0101 =1010`
`>>`	Shift right - moves the value of the left operand to the right by the number of bits that the right operand specifies	`60>>2; 00001111` (60 is 00111100 in binary)
`<<`	Shift left - moves the value of the right operand to the left by the number of bits that the right operand specifies	`60<<2; 11110000` (60 is 00111100 in binary)

Bit Shifting

- The behavior of shifting depends on whether the number is `signed` or `unsigned`
- Shift Right
  - **Unsigned:** bits that are shifted out from the least significant position are discarded, and zeros are shifted in from the left

```
int i=714; i>>1;

357 178 89 44 22 11 5 2 1 0
```

  - **Signed:** the behavior can vary depending on the implementation
    - In most **C** implementations, right shifting a signed integer is an **arithmetic shift**
    - This means that the sign bit (the most significant bit) is replicated

2's complement of -8 (16-bit): `1111 1111 1111 1000`

    - Right-shifting by 2

2's complement of -8 (16-bit): `1111 1111 1111 1110`

Bit Shifting

- Shift Left
  - **Unsigned:** any bits shifted out of the most significant position are discarded, and zeros are shifted in from the right
  - **Signed:** Can be more complex due to the two's complement representation of signed integers in C
    - If a signed integer is left shifted, it may lead to an **undefined behavior** if the sign bit is affected

Compound Assignments

- Shortcuts that combine a binary operator with an assignment

Arithmetic

Operator	Function	Example
`+=`	Adds the right operand to the left operand and assigns the result to the left operand	`x += 5;` `x = x + 5;`
`-=`	Subtracts the right operand from the left operand and assigns the result to the left operand	`x -= 3;` `x = x - 3;`
`*=`	Multiplies the left operand by the right operand and assigns the result to the left operand	`x *= 2;` `x = x * 2;`
`/=`	Divides the left operand by the right operand and assigns the result to the left operand	`x /= 4;` `x = x / 4;`
`%=`	Calculates the modulus (remainder) of the left operand divided by the right operand and assigns the result to the left operand	`x %= 7;` `x = x % 7;`

Compound Assignments

Bitwise

Operator	Function	Example
`&=`	Performs a bitwise AND on the left and right operands and assigns the result to the left operand	`x &= y;` `x = x & y;`
`|=`	Performs a bitwise OR on the left and right operands and assigns the result to the left operand	`x |= y;` `x = x | y;`
`^=`	Performs a bitwise XOR on the left and right operands and assigns the result to the left operand	`x ^= y;` `x = x ^ y;`
`<<=`	Left shifts the bits of the left operand by the number of positions specified by the right operand and assigns the result to the left operand	`x <<= y;` `x = x << y;`
`>>=`	Right shifts the bits of the left operand by the number of positions specified by the right operand and assigns the result to the left operand	`x >>= y;` `x = x >> y;`

Precedence

Operator	Operator Name	If multiple	Precedence
( )	Parentheses	L to R	1
++, --	Postincrement	L to R	2
++, --	Preincrement	R to L	3
+, -	Positive, negative	L to R	3
*, /, %	Multiplication, division	L to R	4
+, -	Addition, subtraction	L to R	5
<=, >=, >, <	Relational operator	L to R	6
==, !=	Relational operator	L to R	7
&&	Logical AND	L to R	8
||	Logical OR	L to R	9
+=, -=, *=, /=, %=	Compound assignment	R to L	10
=	Assignment	R to L	10

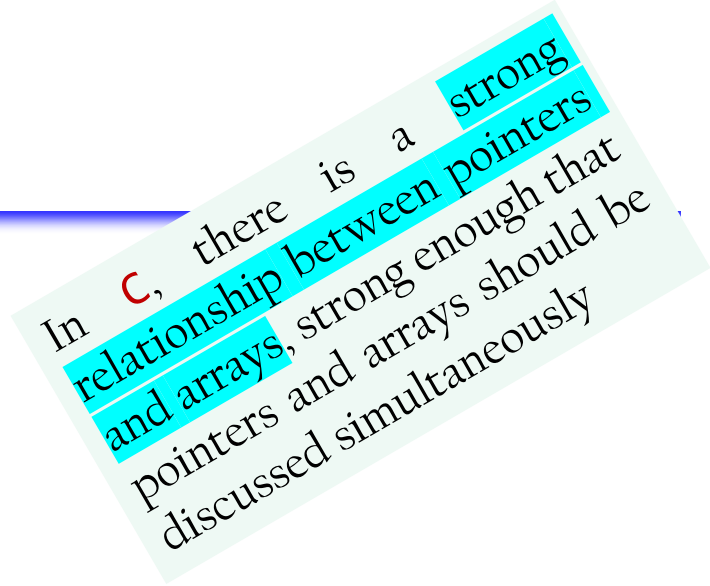
Precedence - Examples

```
abc = x+y*z;
```
// * higher than +

```
x *= y+1;
```
// addition higher than *=

```
int a=2, b=3, c=5, d=7, e=11, f=3;
f +=a/b/c;          // 2/3=0/5=0 --> f=3
d -=7+c*--d/e;      // 5 * 6 = 30; 30/11=2 ,2+7=9; d=d-9= 6-9=-3
d= 2*a%b+c+1;       // 2*2=4%3=1; 1+5+1=7
a +=b +=c +=1+2;    // c=3+5=8; b=3+8=11; a=2+11=13
```

Arrays



- An array is used to group elements of the same data type
- It is indicated by brackets (`[]`), for example `arr[ ]`
- Array size is specified as a positive integer constant
- Each element in the array is accessed via an index, where the first element has an index of `0`
  - For example, `arr[0]` accesses the first element
- Loops, like `for` loops, are commonly used to manipulate arrays (e.g., traversing or updating values)
- In `C`, the size of an `array` can be explicitly defined, implicitly determined, or left unspecified depending on the context of the declaration

Array Declaration/Initialization

- Syntax

```
type  name[value];
```

- Examples

```
int bigArray[10]; // explicitly defined

int arr[] = {1, 2, 3, 4, 5};  // Implicit - compiler will automatically determine the size

int arr[5] = {1, 2};  // arr will be {1, 2, 0, 0, 0}

double a[3];

char grade[10], oneGrade;
```

Array Declaration/Initialization

- When you declare and initialize an array, you can use curly braces {} to provide initial values

```
int a[5] = {11,22};
```

Declares array a and initializes first two elements and all remaining set to zero

```
int b[] = {1,2,8,9,5};
```

Declares array b and initializes all elements and sets the length of the array to 5

Arrays Declaration → Memory Allocation

- Declare the array → allocates memory

```
int score[5];
```

  - Declares `array` of `5` integers named "`score`"
  - Similar to declaring five variables:

```
int score[0], score[1], score[2], score[3], score[4];
```

- Individual parts are called:
  - `Indexed` or `subscripted` variables
  - "`Elements`" of the `array`
- Value in brackets are called `index` or `subscript`
  - Numbered from `0` to `(size - 1)`

Arrays Declaration - Examples

Processed by the preprocessor before the actual compilation of code begins (include header files, define constants, conditional compilation)

```
#define N_COL 200
```

Preprocessor directive (not a variable)

```
const int N_ROW = 100;
```

`const` makes the variable immutable

```
float arr[ N_ROW ][ N_COL ];
```

2D array of floats with dimensions `N_ROW` × `N_COL`

```
scanf( "%d", &N );
```

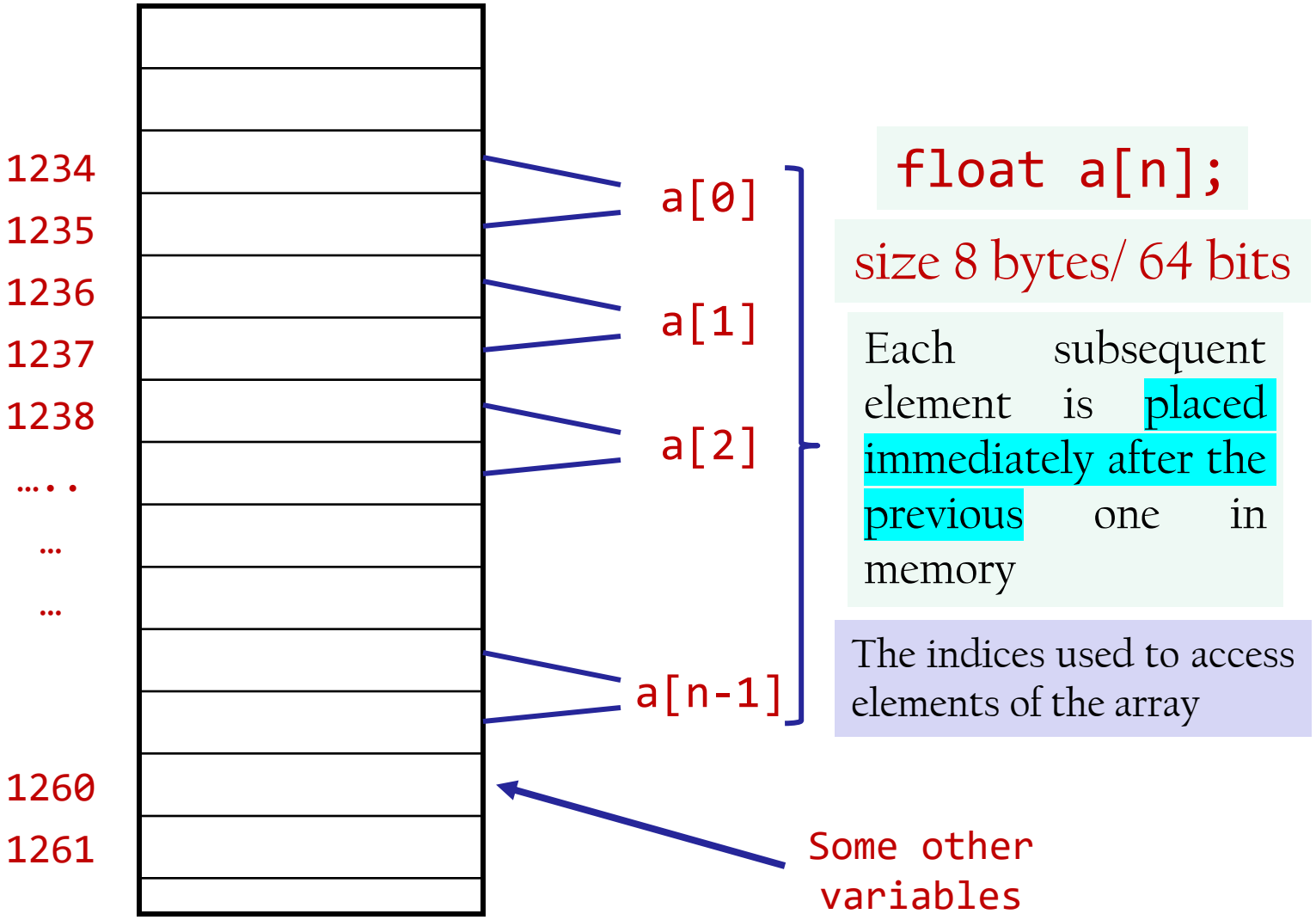
Read an integer value from the user and store it in `N`

```
double data[ N ];
```

Declare a variable-length array (VLA) of type `double` with a size of `N`

Graphical representation of an `array` in memory

32-bit memory layout




```
float a[n];
```

size 8 bytes/ 64 bits

Each subsequent element is placed immediately after the previous one in memory

The indices used to access elements of the array

Array Access

- `int ar[]={1,2,3,4,5};`
- `X = ar[2];` X=3
- `ar[3] = 2.7;` `ar[]={1,2,3,2,5}` (2.7 truncated to 2)
- What is the difference between
  - `ar[i]++`
  - `ar[i++]`
  - `ar[++i]`

Next Slide

Array Access

Expression	Accessed Value	Index After Operation
`ar[i]++`	Current value at `ar[i]`, then increment	`i` remains unchanged during the access
`ar[i++]`	Current value at `ar[i]`, then increment `i`	`i` increments after access
`ar[++i]`	Value at `ar[i + 1]` (after incrementing `i`)	`i` increments before access

```
int ar[5] = {1, 2, 3, 4, 5};
int value = ar[2]++; // value = 3, ar[2] becomes 4
```

```
int ar[5] = {1, 2, 3, 4, 5};
int i = 2;
int value = ar[i++]; // value = 3, i becomes 3
```

```
int ar[5] = {1, 2, 3, 4, 5};
int i = 2;
int value = ar[++i]; // value = 4, i becomes 3
```

String – Array of type **char**

- No `string` type in `C`
- Like we declare `int` type `array`:

```
int a[] = {1,2,3,4}
```

1	2	3	4

- Similarly, we declare `char` type `array`, which is a `string`:

```
char greetings[] = "Hello"
```

'H'	'e'	'l'	'l'	'o'	'\0'

String Functions

- `man` the following functions on Linux

Since Linux is a POSIX-compliant operating system, it includes the standard `C` library and supports all functions defined in `<string.h>` for manipulating `C`-style strings and memory

- `strcpy(s,t)` Copies the string **`t`** to the string **`s`**
- `strcmp(s,t)` Compares the character strings **`s`** and **`t`**, and returns negative, zero or positive if **`s`** is lexicographically less than, equal to, or greater than **`t`**, respectively
- `strcat(s,t)` Concatenates the string **`t`** to the end of string **`s`**. **`strcat`** assumes that there is enough space in **`s`** to hold the combination. It returns no value.
- `strlen(s)` Returns the length of its character string argument **`s`**, excluding the terminal **`'\0'`**
- `strchr(s,c)` Returns pointer to first **`c`** in **`s`**, or **`NULL`** if not present
- `strstr(s,t)` Returns a pointer to the first occurrence of the string **`t`** in the string **`s`**, or **`NULL`** if there is none

Example

```
#include <stdio.h>
#include <string.h>

// define a fixed value for USERNAME and PASSWORD
#define USERNAME "admin"
#define PASSWORD "admin@123"

int main()
{
    char un[50], pass[50];

    //input USERNAME and PASSWORD
    printf("Enter username: ");
    scanf("%s", un);
    printf("Enter password: ");
    scanf("%s", pass);

    //validate username and password
    if(strcmp(USERNAME,un)==0 && strcmp(PASSWORD, pass)==0)
        printf("Input credentials are correct\n");
    else
        printf("Input credentials are not correct\n");
    return 0;
}
```

No **&** for string in scanf

Similarly, no **&** for other data type arrays. But **&** is used for reading individual elements, that is, indexed element of an array!

```
int arr[3];

scanf("%d", arr);

scanf("%d", &arr[1]);
```

Control Flow

- ## Statements and Blocks
  - An expression such as `x = 0` or `i++` or `printf(...)` becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;
i++;
; // Null statement
printf(...);
```

  - Braces `{` and `}` are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement
    - The braces used to surround the statements of a function
    - The braces used around multiple statements after an `if`, `else`, `while`, or `for`

if-else

- Used to express decisions

```
if(expression)
        statement_1
else
        statement_2
```

`else` part is optional

- The `expression` is evaluated; if it is true (that is, if `expression` has a non-zero value), *`statement_1`* is executed.
- If it is false (`expression` is zero),

and

- If there is an `else` part, *`statement_2`* is executed instead

if-else

```
if (expression_1)
  statement_1
        if (expression_2)
              statement_2
else
  statement_3
```

else is associated with the closest if, which is the second if (expression_2)

Because the **else** is optional, it is a good idea to use braces when there are nested **ifs**

else-if

```
if (expression)
  statement
else if (expression)
  statement
else if (expression)
  statement
else if (expression)
  statement
else
  statement
```

- The expressions are evaluated in order
- If an expression is `true`, the statement associated with it is executed, and this terminates the whole chain
- The last `else` part handles the `default` case where none of the other conditions is satisfied
  - Can be omitted
  - May be used for error checking to catch an "impossible" condition

Conditional Operator

- In `C`, the ternary operator (also called the conditional operator) is a concise way to write an `if-else` statement in a single line

```
(condition) ? expr-true : expr-false;
```

```
z=(a>b)?  a:b
```

If `a > b` is true, then the value of `a` is assigned to `z`

If `a > b` is false, then the value of `b` is assigned to `z`

- Remember: False is `0`, and anything else (nonzero integers) is `1` in `C`

switch

- The `switch` statement is a multi-way decision that tests whether an `expression` matches one of a number of constant integer values or constant `expressions`, and branches accordingly
- If a `case` matches the `expression` value, execution starts at that `case`
- All `case` expressions must be different
- The `case` labeled `default` is executed if none of the other cases are satisfied.
- A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place
- The `break` statement causes an immediate `exit` from the `switch`

switch

```
switch (expression) {
case const-expr:
  statements
case const-expr:
  statements
default:
  statements
}
```

Falling through from one `case` to another is not robust, being prone to disintegration when the program is modified

```
switch (expression) {
case const-expr:
  statements
  break;
case const-expr:
  statements
  break;
default:
  statements

}
```

Loops – `while` and `do-while`

```
while (expression)
  statement
```

- The `expression` is evaluated
- If it is non-zero, `statement` is executed, and the `expression` is reevaluated
- This cycle continues until `expression` becomes zero, at which point execution resumes after `statement`

```
do
  statement
while (expression);
```

- The body is always executed at least once
- The `statement` is executed, then `expression` is evaluated
- If it is true, `statement` is executed again, and so on
- When the `expression` becomes false, the loop terminates

`for` Loop

```
for (expr1; expr2; expr3)
  statement
```

- Most commonly, `expr1` and `expr3` are assignments or function calls and `expr2` is a relational expression
- Any of the three parts can be omitted, although the semicolons must remain
- If `expr1` or `expr3` is omitted, it is simply dropped from the expansion
- If the test, `expr2`, is not present, it is taken as permanently true

```
for (;;) {
...
}
```

- It is an "infinite" loop, presumably to be broken by other means, such as a `break` or `return`

Whether to use `while` or `for`

- Largely a matter of personal preference, for example:

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
```

- There is no initialization or re-initialization, so the `while` is most natural
- The `for` is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop

```
for (i = 0; i < n; i++)
  ...
```

```
for(i=0, j=3; i<10 && k>2; i++,j--)
  ...
```

break

- It is sometimes convenient to be able to `exit` from a loop other than by testing at the top or bottom
- The `break` statement provides an early `exit` from `for`, `while`, and `do`, just as from `switch`
- A `break` causes the innermost enclosing loop or `switch` to be exited immediately

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
int n;
for (n = strlen(s)-1; n >= 0; n--){
   if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n’)
          break;
   }
   s[n+1] = ‘\0’;
   return n;
}
```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters!

continue

- The `continue` causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin
- In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step
- The `continue` statement applies only to loops, not to `switch`
- A `continue` inside a `switch/if` inside a loop causes the next loop iteration

```
/* processes only the non-negative elements in the array a */
for (i = 0; i < n; i++)
   if (a[i] < 0) /* skip negative elements */
         continue;
   ... /* do positive elements */
```

Pointers

- Contain the "**address**" of a variable
- Lead to more compact and efficient code
- Pointers and arrays are closely related
- General form

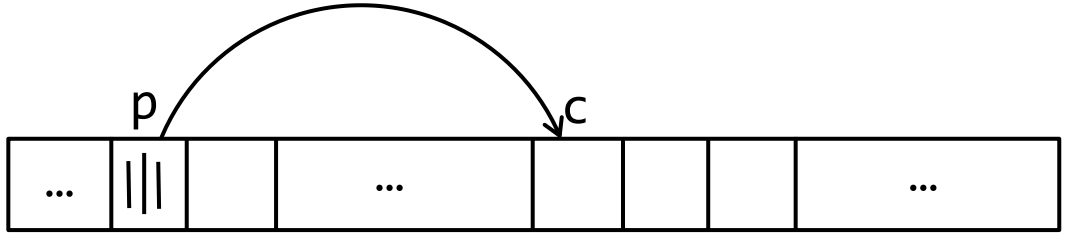
```
pointer_variable = &ordinary_variable
```

Name of the pointer

Name of ordinary variable

Pointers

- A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups



- A `pointer` is a group of cells that can hold an (memory) address
- So, if `c` is a `char` and `p` is a `pointer` that points to it, we could represent the situation as shown
- The unary operator & gives the address of an object

```
p = &c;
```

// assigns the address of c to the variable p, and p is said to "point to" c

Pointers - Declaration

- Declared with data type, * and identifier

```
type *pointer_var1, *pointer_var2, …
```

- Example.

```
double *p;

int *p1, *p2;
```

- There has to be a * before EACH of the pointer variables

Pointers – Accessing Value

- Can be used to access value
- Example

```
int *p1, v1;
v1 = 0;
p1 = &v1;
*p1 = 42; /*accesses the object the pointer points to
(v1) and make it 42 */
printf("%d\n",v1);
printf("%d\n,*p1);
```

Output:
42
42

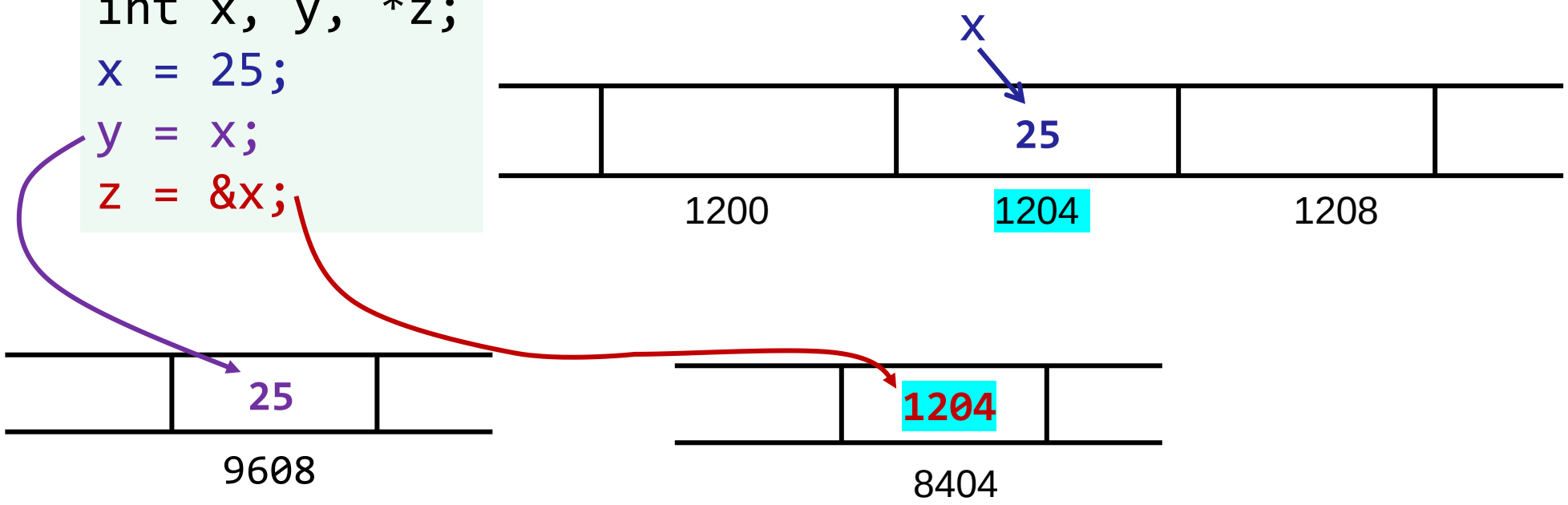
Pointers

- The unary operator * is the indirection or dereferencing operator
- When applied to a pointer, it accesses the object the pointer points to

```
int x = 1, y = 2, z[10];

int *ip; // ip is a pointer to int

ip = &x; // ip now points to x

y = *ip; // y is now 1 as ip points to (*) x

*ip = 0; // x is now 0

ip = &z[0]; // ip now points to z[0]
```

Pointers – More Explanation

```
int x, y, *z;
x = 25;
y = x;
z = &x;
```



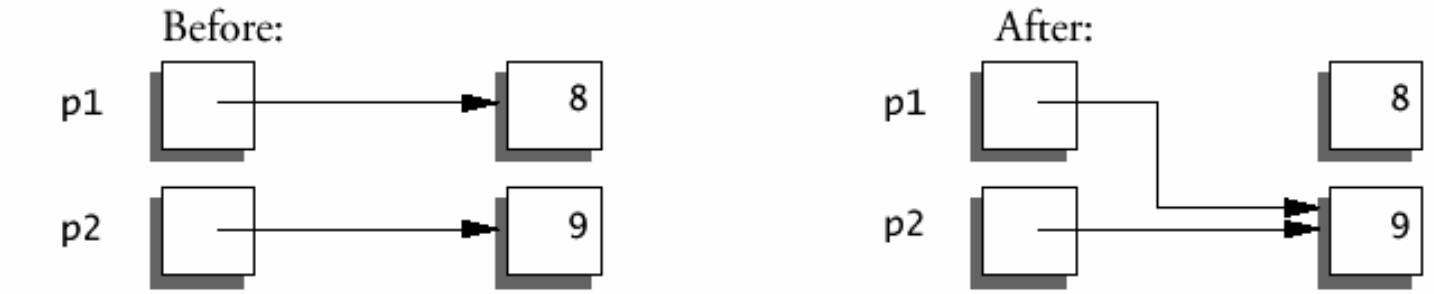
- To access a variable via address

```
a = 0xABCD0000; //BAD IDEA
a = &i;
```

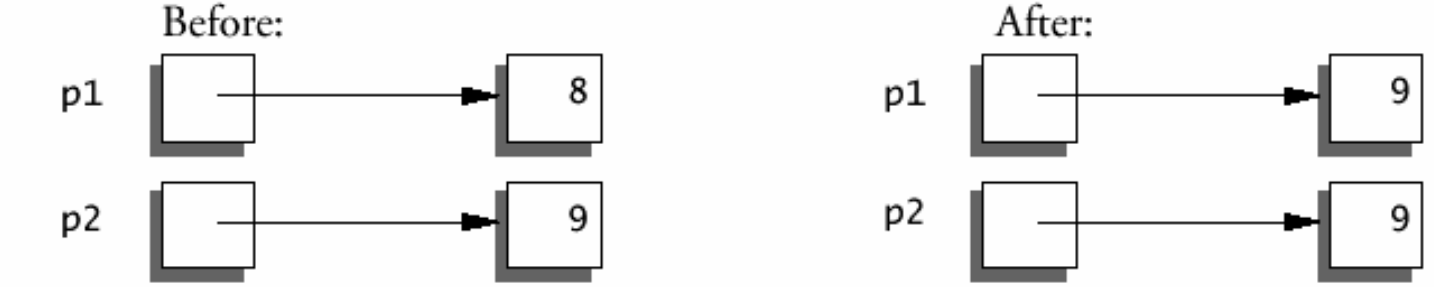
a is address here

Pointers – More Explanation

```
p1 = p2;
```




```
*p1 = *p2;
```



Pointer to Pointers

```
int x=5, *y, **z;

y = &x; // pointer y is assigned the address of variable x

z = &y; // pointer z is assigned the address of variable y, which
        // itself points to x

printf("%d", x); // Outputs the value of x

printf("%d", *y); // Outputs the value of x (because y points to x)

printf("%d", **z); // Outputs the value of x (because z points
                   // to y, and y points to x)
```

Pointer Arithmetic

- int *x, *y
- int z;
- Can do
  - z=x-y;
  - x=NULL;
  - if(x==NULL)
  - void *

```
int arr[5];
int *x = &arr[3];  // x points to the 4th element
int *y = &arr[1];  // y points to the 2nd element
int z = x - y;     // z will be 2 because there
                   // are 2 elements between
                   // arr[1] and arr[3]
```

Example

A NULL pointer is a special pointer that points to nothing

This check is used to ensure that the pointer does not point to any valid memory before trying to dereference it

Generic pointer in C, meaning it can point to any data type. Unlike other pointers (e.g., int *, float *), a void * does not have a specific type associated with it

Pointer Arithmetic

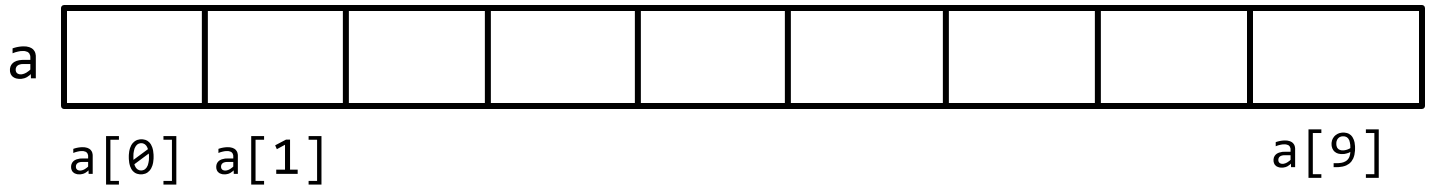
- What is the unit of `x++` or `x+4`?
  - The unit of `x++` or `x+4` is determined by the data type that the pointer `x` points to
  - `x++`: This increments the pointer `x` to point to the next element in the array. If `x` points to an `int`, and an `int` is `4 bytes`, then `x++` moves the pointer by `4 bytes`
  - `x + 4`: This moves the pointer `x` forward by `4` elements in the array. If `x` points to an `int`, the total distance moved in memory is `4 * sizeof(int)`. If `sizeof(int)` is `4 bytes`, `x+4` would move the pointer forward by `16 bytes (4 elements)`

Pointer Arithmetic

- Legal:
  - Adding/Subtracting a pointer and an integer
  - Subtracting two pointers (only if they point to the elements in the same array)
  - Comparing two pointers to members of the same array: (`p1 < p2`, `p1 > p2`, etc. If `p1` points to a memory location that comes before `p2`, then `p1 < p2` is true
- Illegal:
  - Adding Two Pointers: Adding two memory addresses doesn't result in a meaningful memory location
  - Multiplying Two Pointers: Pointer multiplication has no meaningful use in memory addressing
  - Dividing: There is no meaningful way to divide memory addresses
  - Assigning one pointer type to another without a cast: Exception: `void *`

Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously
- Any operation that can be achieved by array subscripting can also be done with pointers
- The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand
- Recap: Arrays



Pointers – Pointing to Array elements

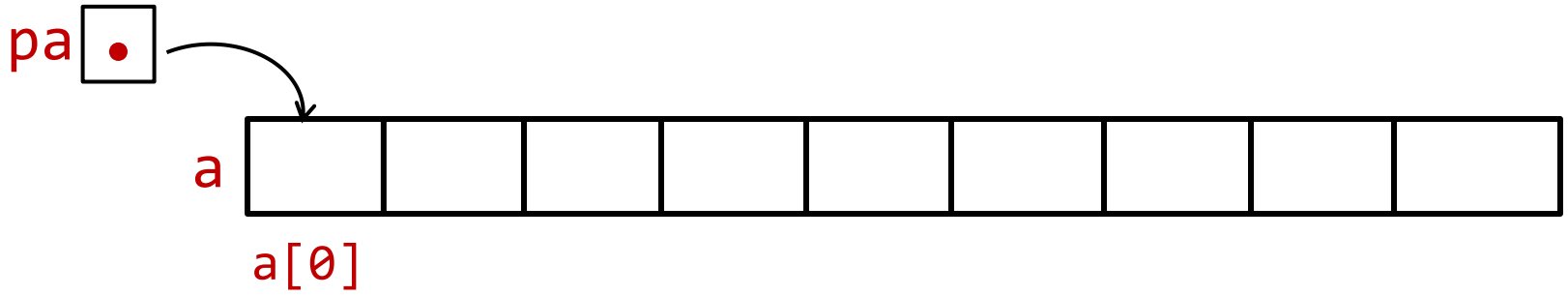
- If `pa` is a pointer to an integer, declared as:

  ```
  int *pa;
  ```

  then the assignment

  ```
  pa = &a[0];
  ```

  sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`



Pointers – Pointing to Array elements

- Now the assignment

```
x = *pa; // copy the contents of a[0] into x
```

- If `pa` points to a particular element of an array, then by definition `pa+1` points to the next element
- `pa+i` points to `i` elements after `pa`, and `pa-i` points to `i` elements before `pa`
- Thus, if `pa` points to `a[0]`

```
*(pa+1); // refers to the contents of a[1]
```

- Similarly, `pa+i` is the address of `a[i]`, and `*(pa+i)` is the content of `a[i]`

True regardless of the type or size of the variables in the array `a`

Referencing Array elements via Pointer and Array name

- We know that:

```
pa = &a[0]; // pa and a have identical values
```

- Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

- A reference to `a[i]` can also be written as `*(a+i)`
- In evaluating `a[i]`, `C` converts it to `*(a+i)` immediately; the two forms are equivalent
- Also `a+i = &a[i]`

Pointers (variables) and Arrays (identifiers)

- There is one difference between an array name and a pointer
- A pointer is a variable, so `pa=a` and `pa++` are legal
- But an array name is not a variable, rather an identifier; constructions like `a=pa` and `a++` are illegal
- Identifier of an array is equivalent to the address of its first element

Example

```
int numbers[20];
int * p;
p = numbers;     // Valid
numbers = p;     // Invalid
```

- `p` and `numbers` are equivalent, and they have the same properties
- Only difference is that we could assign another value to the pointer `p` whereas `numbers` will always point to the first of the 20 integer numbers of type `int`

Pointers and Functions

- A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation
- A function is a small program that may receives some data, perform some computations, and may return a value
- Before the use, functions must be declared:

```
return-type function-name(parameter declarations, if any);
```

Functions

```
return-type function-name(parameter declarations, if any)
{
declarations
statements
}
```

Format of Called Function

- A function may return a value using `return`
- Returning a value from a function that returns `void` is an error
- Not returning a value from a function that returns a value is unpredictable

Functions - Example

```
#include <stdio.h>
int power(int m, int n);
/* test power function */
main()
{

int i;  // local to the function
for (i = 0; i < 10; ++i)
printf("%d %d %d\n", i,
power(2,i), power(-3,i));
return 0;
}
```

Calling Function

Passed to the called function

```
/* power: raise base to n-th
power; n >= 0 */
int power(int base, int n)
{
int i, p; // local to the function

p = 1;  // local to the function
for (i = 1; i <= n; ++i)
      p = p * base;
return p;
}
```

Called

pow(x,y) – standard library function

Functions – Scope of `variables`

- Variables do exist within their own function block
- Each *local variable* in a function comes into existence only when the function is called, and disappears when the function is exited *[Automatic Variables]*
- As an alternative to *automatic variables*, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function
- The *external variables* are globally accessible, they can be used instead of argument lists to communicate data between functions
- The *external variables* retain their values even after the functions that set them have returned

External Variables

- Usually, an *external variable* must be declared outside of any function that uses it

```
char line[MAXLINE];
```

- The variable must also be declared in each function that wants to access it; this states the type of the variable

```
extern char line[];
```

- If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an `extern` declaration in the function

External Variables

- If the program is in several source files, and a variable is defined in *file1* and used in *file2* and *file3*, then `extern` declarations are needed in *file2* and *file3* to connect the occurrences of the variable

- The usual practice is to collect `extern` declarations of variables and functions in a separate file, historically called a header, that is included by `#include` at the front of each source file

- The suffix `.h` is conventional for header names. The functions of the standard library, for example, are declared in headers like `<stdio.h>`

Static Variables

- If a variable is declared `static` outside of `main`, it is not visible to other files

```
static int bufp = 0;
```

- `static` can also be used internal to functions
- If a variable in a function is declared `static`, the variables does not vanish after the function returns, it stays in the memory so the next call to the function will find the old value
- This means that internal `static` variables provide private, permanent storage within a single function
- `static` variables are initialized to `0`

Functions – argument passed "by value"

- In C, all function arguments are passed "by value"
- This means that the called function is given the values of its arguments in temporary variables rather than the originals
- Parameters can be treated as conveniently initialized local variables in the called routine
- Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function

Functions – To Modify a `variable` in Calling Routine

- When necessary, it is possible to arrange for a function `to modify a variable in a calling routine`
- The caller must provide the address of the variable to be set (`technically a pointer to the variable`), and the called function must declare the `parameter to be a pointer` and access the variable indirectly through it

Check Example in the next slide

Example – Functions and Pointers

```
#include<stdio.h>
void swap(int *x, int *y);
void main( ) {
  int a, b;
  scanf("%d %d", &a, &b);
  swap(&a, &b);
  printf("%d %d\n", a, b);
}
```

```
void swap(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

Functions, Arrays, and Pointers

- For arrays, the story is different with respect to the other identifiers (pass-by-value)
- But it is similar with respect to the pointers (pass-by-address)

```
strcpy(char dest[], char src[])
```

- When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array
- By subscripting this value, the function can access and alter any argument of the array *[There is no copying of array elements]*
- This means within the called function, an array name parameter is a pointer, that is, a variable containing an address

Remember that the name of the array is a pointer to its first element

Thank You!

Happy Learning