

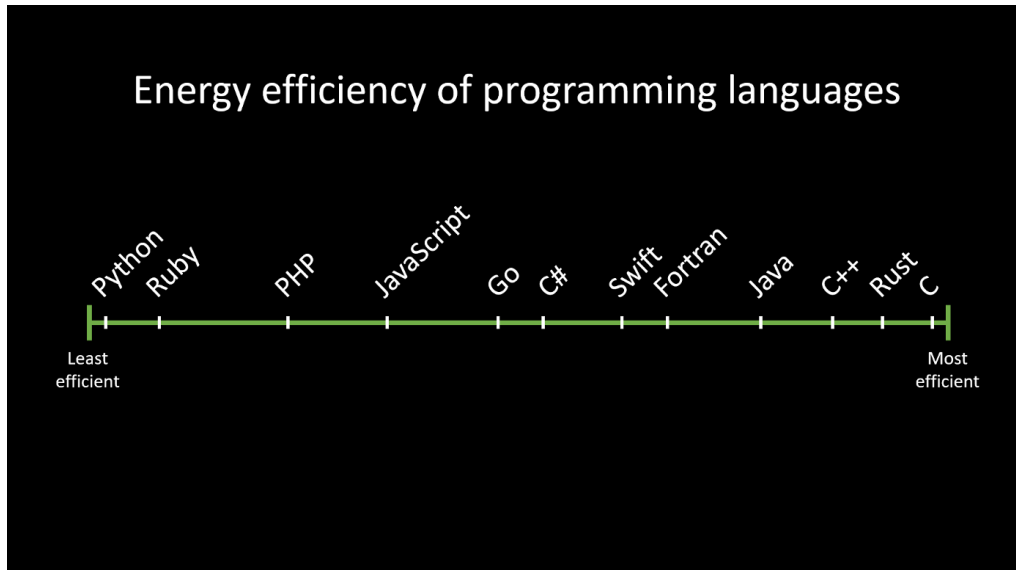
# Module 1A

## Embedded C Programming

- ❑ The slides are prepared using
  - Essential C (Summary of the basic features of the C language)
    - <http://cslibrary.stanford.edu/101/EssentialC.pdf>
  - Embedded C Tutorial
    - [https://community.nxp.com/legacyfs/online/archiveatt/Lecture\\_3\\_-\\_C\\_Intro.pdf](https://community.nxp.com/legacyfs/online/archiveatt/Lecture_3_-_C_Intro.pdf)
  - The textbook by Mazidi et al and its publicly available slides
    - [http://www.microdigitaled.com/ARM/Freescale\\_ARM\\_books.htm](http://www.microdigitaled.com/ARM/Freescale_ARM_books.htm)
  - C programming tutorial from
    - [https://www.eng.auburn.edu/~nelson/courses/elec3040\\_3050/](https://www.eng.auburn.edu/~nelson/courses/elec3040_3050/)
  - Valvano's E-book on Embedded C programming
    - <http://users.ece.utexas.edu/~valvano/embed/toc1.htm>
  - TutorialsPoint
    - <https://www.tutorialspoint.com/cprogramming>

# Why C programming?

- 😊 It is easier and less time consuming to write in C than Assembly.
- 😊 C is easier to modify and update.
- 😊 You can use code available in function libraries.
- 😊 C code is portable to other microcontrollers with little or no modification.
- 😞 Generally generates larger code
- 😞 Programmer has less control and less ability to directly interact with the hardware



- 😊 [1] Pereira, R. et al. (2017) 'Energy efficiency across programming languages: how do energy, time, and memory relate'. doi:[10.1145/3136014.3136031](https://doi.org/10.1145/3136014.3136031).



# Basic C Program Structure

```
#include "STM32L1xx.h" /* I/O port/register names/addresses for the STM32L1xx microcontrollers */
```

```
/* Global variables – accessible by all functions */
```

```
int count, bob; //global (static) variables – placed in RAM
```

```
/* Function definitions*/
```

```
int function1(char x) { //parameter x passed to the function, function returns an integer value
```

```
int i,j; //local (automatic) variables – allocated to stack or registers
```

```
-- instructions to implement the function
```

```
}
```

```
/* Main program */
```

```
void main(void) {
```

```
unsigned char sw1; //local (automatic) variable (stack or registers)
```

```
int k; //local (automatic) variable (stack or registers)
```

```
/* Initialization section */
```

```
-- instructions to initialize variables, I/O ports, devices, function registers
```

```
/* Endless loop */
```

```
while (1) { //Can also use: for(;;) {
```

```
-- instructions to be repeated
```

```
} /* repeat forever */
```

```
}
```

Declare local variables

Initialize variables/devices

Body of the program

# ANSI C (ISO C89) Integer Data Types and Their Ranges

Data type	Size	Range Min	Range Max
<b>char</b>	1 byte	-128	127
<b>unsigned char</b>	1 byte	0	255
<b>short int</b>	2 bytes	-32,768	32,767
<b>unsigned short int</b>	2 bytes	0	65,535
<b>int</b>	4 bytes	-2,147,483,648	2,147,483,647
<b>unsigned int</b>	4 bytes	0	4,294,967,295
<b>long</b>	4 bytes	-2,147,483,648	2,147,483,647
<b>unsigned long</b>	4 bytes	0 to 4,294,967,295	
<b>long long</b>	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
<b>unsigned long long</b>	8 bytes	0	18,446,744,073,709,551,615

- Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware.
- Unfortunately, it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine.

# ISO C99 Integer Data Types and Their Ranges

Data type	Size	Range Min	Range Max
int8_t	1 byte	-128	127
uint8_t	1 byte	0 to	255
int16_t	2 bytes	-32,768	32,767
uint16_t	2 bytes	0	65,535
<b>int32_t</b>	<b>4 bytes</b>	<b>-2,147,483,648</b>	<b>2,147,483,647</b>
uint32_t	4 bytes	0	4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	8 bytes	0	18,446,744,073,709,551,615

# Constants/Literals

- ❑ Decimal is the default number format

```
int m,n; //16-bit signed numbers
m = 453; n = -25;
```

- ❑ Hexadecimal: preface value with 0x or 0X

```
m = 0xF312;
```

- ❑ Octal: preface value with zero (0)

```
m = 0453; n = -023;
```

- ❑ Don't use leading zeros on "decimal" values. They will be interpreted as octal.

- ❑ Character: character in single quotes, or ASCII value following "slash"

```
m = 'a'; //ASCII value 0x61
```

```
n = '\n';
```

```
//ASCII value 13 "return" character (escape characters) \)
```

- ❑ String (array) of characters:

- unsigned char k[7];

```
strcpy(k,"hello\n");
```

```
//k[0]='h', k[1]='e', k[2]='l', k[3]='l', k[4]='o',
```

```
//k[5]=13 or '\n' (ASCII new line character),
```

```
//k[6]=0 or '\0' (null character - end of string)
```

'A' uppercase 'A' character
'\n' newline character
'\t' tab character
'\0' the "null" character -- integer value 0 (different from the char digit '0')
'\012' the character with value 12 in octal, which is decimal 10

# Variables

- ❑ A variable is an addressable storage location to information to be used by the program
- ❑ Each variable must be declared to indicate size and type of information to be stored, plus name to be used to reference the information

```
int x,y,z; //declares 3 variables of type "int"
char a,b; //declares 2 variables of type "char"
```
- ❑ Space for variables may be allocated in registers, RAM, or ROM/Flash (for constants)
- ❑ Variables can be automatic or static

# Automatic Variables

- ❑ Declare within a function/procedure
- ❑ Variable is visible (has scope) only within that function
- ❑ Space for the variable is allocated on the system stack when the procedure is entered
- ❑ Deallocated, to be re-used, when the procedure is exited
- ❑ If only 1 or 2 variables, the compiler may allocate them to registers within that procedure, instead of allocating memory.
- ❑ Values are not retained between procedure calls



# Automatic Variable Example

```
void delay () {  
  int i,j; //automatic variables - visible only within  
  delay()  
  for (i=0; i<100; i++) { //outer loop  
    for (j=0; j<20000; j++) { //inner loop  
      } //do nothing  
    }  
  }
```

Variables must be initialized each time the procedure is entered since values are not retained when the procedure is exited.

# Static Variables

- ❑ Retained for use throughout the program in RAM locations that are not reallocated during program execution.
- ❑ Declare either within or outside of a function
- ❑ If declared outside a function:
  - the variable is global in scope, i.e. known to all functions of the program
  - Use “normal” declarations.
  - Example: `int count;`
- ❑ If declared within a function:
  - insert key word `static` before the variable definition.
  - The variable is local in scope, i.e. known only within this function.  
`static unsigned char bob;`  
`static int pressure[10];`

# Static Variable Example

```
unsigned char count; //global variable is static – allocated a fixed RAM location
                        //count can be referenced by any function

void math_op () {
    int i;              //automatic variable – allocated space on stack when function entered
    static int j;      //static variable – allocated a fixed RAM location to maintain the value
    if (count == 0)     //test value of global variable count
        j = 0;         //initialize static variable j first time math_op() entered
    i = count;          //initialize automatic variable i each time math_op() entered
    j = j + i;          //change static variable j – value kept for next function call
}                      //return & deallocate space used by automatic variable i

void main(void) {
    count = 0;          //initialize global variable count
    while (1) {
        math_op();
        count++;        //increment global variable count
    }
}
```

# Overflow

- ❑ Unlike assembly language programming, high level language programs do not provide indications when overflow occurs and the program just fails silently.
- ❑ If you use a `short int` to hold the number of seconds of a day, the second count will overflow from 32,767 to -32,768. Even if your program handles negative second count, the time will jump back to the day before.

# Coercion

- ❑ If you write a statement with different operand data types for a binary operation, the compiler will convert the smaller data type to the bigger data type. This implicit data type is called **coercion**. For example. 'b' + 15.
- ❑ The compiler may or may not give you warning when coercion occurs.
- ❑ If the variable is signed and the data size is increased, the new bits are filled with the sign bit (most significant bit) of the original value.
- ❑ When you assign a larger data type to a smaller data type variable, the higher order bits will be truncated.

```
char ch;  
int i;  
i = 321;  
ch = i; // truncation of an int value to fit in a char  
// ch is now 65
```

# Type Conversion (Typecasting)

```
{  
int score;  
...// suppose score gets set in the range [0,19]  
score = (score / 20) * 100; // score/20 truncates to 0
```

- ❑ Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20.
- ❑ The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100;  
// OK -- floating point division from cast
```

```
score = (score / 20.0) * 100;  
// OK -- floating point division from 20.0  
}
```

# Contd

- ❑ **int Constant Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long such as 42L.**
- ❑ **An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal; 0x10 is way of expressing the number 16.**
- ❑ **The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, char and int can be combined in arithmetic expressions such as ('b' + 5).**
  - the compiler "promotes" the smaller type (char) to be the same size as the larger type (int) before combining the values. Promotions are determined at compile time based purely on the types of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information.

# Relational and Math Operators in C

- ❑ == Equal
- ❑ != Not Equal
- ❑ > Greater Than
- ❑ < Less Than
- ❑ >= Greater or Equal
- ❑ <= Less or Equal
- ❑ ! Boolean not (unary)
- ❑ && Boolean and
- ❑ || Boolean or
- ❑ + Addition
- ❑ - Subtraction
- ❑ / Division
- ❑ \* Multiplication
- ❑ % Remainder (mod)
- ❑ ++ increment
- ❑ -- decrement



# Pre- and Post-Variations of ++ and --

```
int i = 42;  
int j;  
j = (i++ + 10);  
// i is now 43  
// j is now 52 (NOT 53)
```

```
j = (++i + 10)  
// i is now 44  
// j is now 54
```

# Example

```
unsigned char value, temp1, temp2, temp3;  
value = 0xF5; // 245 base 10  
temp1 = (value%10)+0x30;  
temp2 = value/10;  
temp3 = temp2/10+0x30;  
temp2 = temp2%10+0x30;  
  
printf(temp3"\n"); // MSB  
printf(temp2"\n");  
printf(temp1"\n"); // LSB
```

# Bit-wise Operators in C

A	B	AND (A & B)	OR (A   B)	EX-OR (A^B)	Invert ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

# Setting and Clearing (Masking) bits

- ❑ Anything ORed with a 1 results in a 1; anything ORed with a 0 results in no change.
- ❑ Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.
- ❑ Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change.

# Testing Bit with Bit-wise Operators in C

- When it is necessary to test a given bit to see if it is high or low, the unused bits are masked and then the remaining data is tested.

Example:

```
while(1)
{
    if (var1 & 0x20) /* check bit 5 (6th bit) of var1 */
        var2 = 0x55;
    /* this statement is executed if bit 5 is a 1 */
    else
        var2 = 0xAA;
    /* this statement is executed if bit 5 is a 0 */
}
```

- C does not have a distinct boolean type; int is used instead.
- The language treats integer 0 as false and all non-zero values as true.

# Bit-wise Shift Operation in C

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data >> number of bit-positions to be shifted right
Shift Left	<<	data << number of bit-positions to be shifted left

# Compound Operators

Statement	Its equivalent using compound operators
<code>a = a + 6;</code>	<code>a += 6;</code>
<code>a = a - 23;</code>	<code>a -= 23;</code>
<code>y = y * z;</code>	<code>y *= z;</code>
<code>z = z / 25;</code>	<code>z /= 25;</code>
<code>w = w   0x20;</code>	<code>w  = 0x20;</code>
<code>v = v &amp; mask;</code>	<code>v &amp;= mask;</code>
<code>m = m ^ togBits;</code>	<code>m ^= togBits;</code>

# Bit-wise Operations Using Compound Operators

- ❑ The majority of hardware access level code involves setting a bit or bits in a register, clearing a bit or bits in a register, toggling a bit or bits in a register, and monitoring the status bits. For the first three cases, the compound operators are very suitable.



# Using Shift Operator to Generate Mask

- ❑ One way to ease the generation of the mask is to use the left shift operator. To generate a mask with bit  $n$  set to 1, use the expression:  $1 \ll n$
- ❑ If more bits are to be set in the mask, they can be “or”ed together. To generate a mask with bit  $n$  and bit  $m$  set to 1, use the expression:

$(1 \ll n) \mid (1 \ll m)$

```
register |= (1 << 6) | (1 << 1);
```

# Setting the Value in a Multi-bit Field

```
register |= 1 << 30; // set bit 30
register &= ~(1 << 29); // clear bit 29
register |= 1 << 28; // set bit 28
register &= ~(7 << 28); // clear bits 28,29,30
register |= 5 << 28; // make three bits 101
register = register & ~(7 << 28) | (5 << 28);
```

Precedence	Operator	Associativity
1	~(Bitwise negation)	Right to left
2	<<(Bitwise LeftShift), >>(Bitwise RightShift)	Left to Right
3	& (Bitwise AND)	Left to Right
4	^(Bitwise XOR)	Left to Right
5	(Bitwise Or)	Left to Right

# Control Structures - If

- Both an if and an if-else are available in C. The `<expression>` can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

```
if (<expression>) <statement> // simple form with no {}'s or else
```

```
if (<expression>) { // simple form with {}'s to group statements
<statement>
<statement>
}
```

```
if (<expression>)
{
<statement>
}
else {
<statement>
}
```

**Example:**

```
if (x < y) {
min = x;
}
else {
min = y;
}
```

# Example

```
if (num <= 10 || eli==7)
{
// do something
}
else if (num >= 20)
{
// do something
}
// as many 'else if's as you want
else
{
// default case
}
```

# While Loops

- The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {  
  <statement>  
}
```

- The do-while loop is like a while, but with the test condition at the bottom of the loop.
- Always executed at least once

```
do {  
  <statement>  
} while (<expression>)
```

# For Loops

- ❑ The for loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and an action.

```
for (<initialization>; <continuation>; <action>) {  
    <statement>  
}
```

- ❑ The initialization is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true (like a while). After every execution of the loop, the action is executed.
- ❑ The following example executes 10 times by counting 0..9.

```
for (i = 0; i < 10; i++) {  
    <statement>  
}
```

# Structures

- ❑ C has the usual facilities for grouping things together to form composite types – arrays and records (which are called "structures").

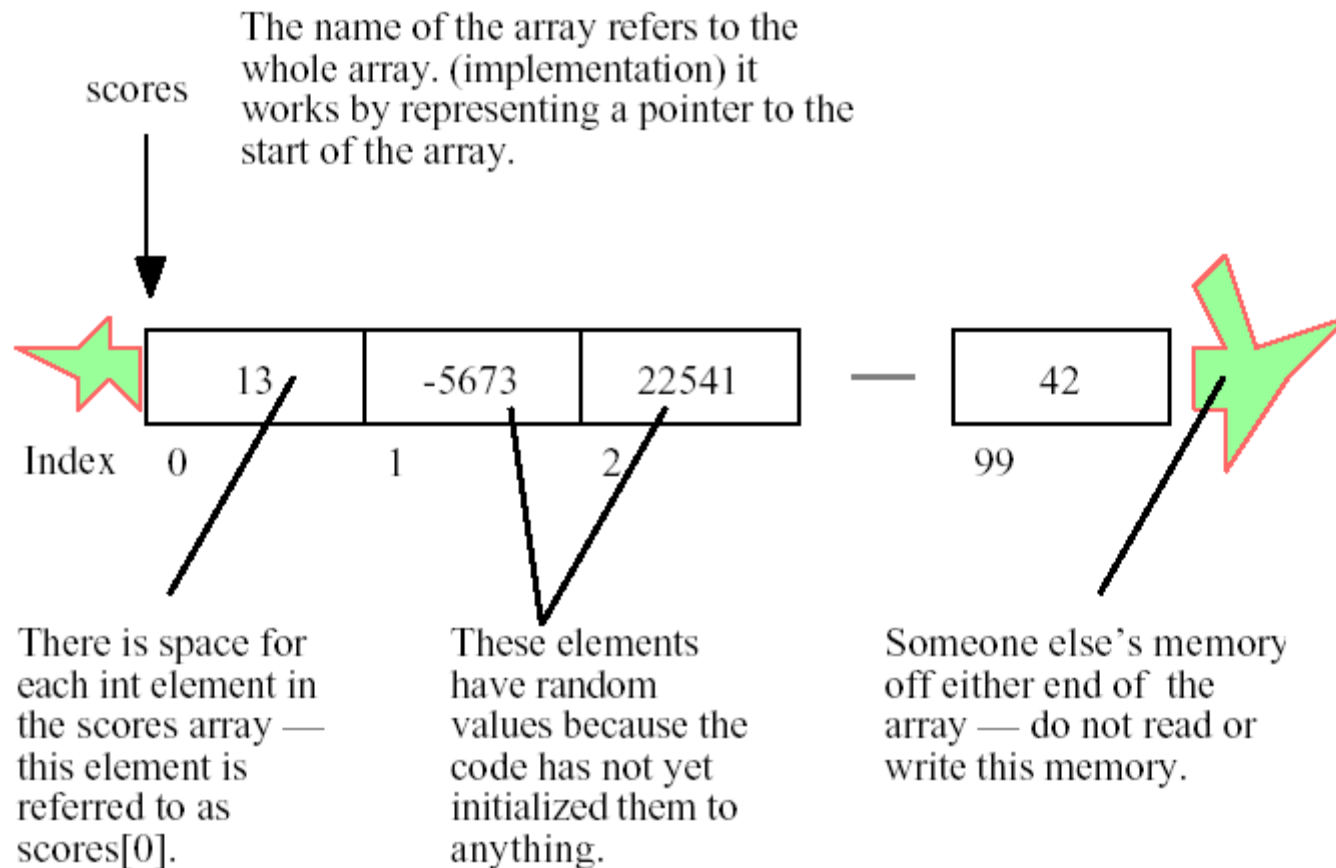
```
struct fraction {  
    int numerator;  
    int denominator;  
}; // Don't forget the semicolon!
```

- ❑ C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, however == does not work on structs.

```
struct fraction f1, f2; // declare two fractions  
f1.numerator = 22;  
f1.denominator = 7;  
f2 = f1; // this copies over the whole struct
```

# Arrays

```
int scores[100]; // array defined  
scores[0] = 13; // set first element  
scores[99] = 42; // set last element
```





# More

- ❑ Initialization

```
int num[] = {1,2};
```

- ❑ Multidimensional Arrays

```
int board [10][10];
```

```
board[9][9] = 13;
```

```
board[0][0] = 13;
```

- ❑ Array of structures

```
struct fraction numbers[1000];
```

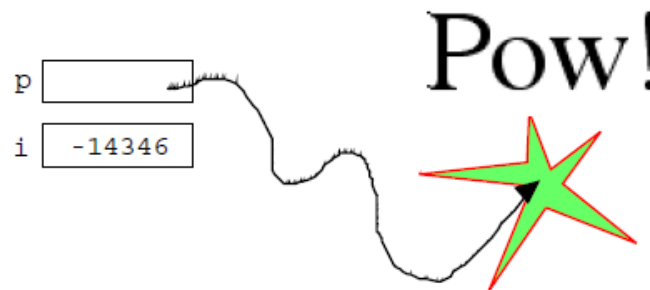
```
numbers[0].numerator = 22; // set the 0th
```

```
numbers[0].denominator = 7;
```

# Pointers

- ❑ When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee".
- ❑ There are three things which must be done for a pointer/pointee relationship to work...
  - (1) The pointer must be declared and allocated
  - (2) The pointee must be declared and allocated
  - (3) The pointer (1) must be initialized so that it points to the pointee (2).

```
{  
    int* p;  
  
    *p = 13;    // NO NO NO p does not point to an int yet  
               // this just overwrites a random area in memory  
}
```

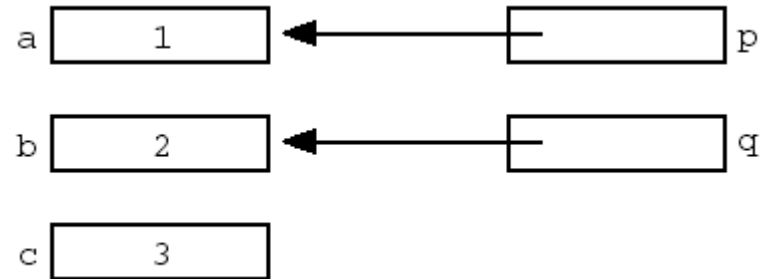
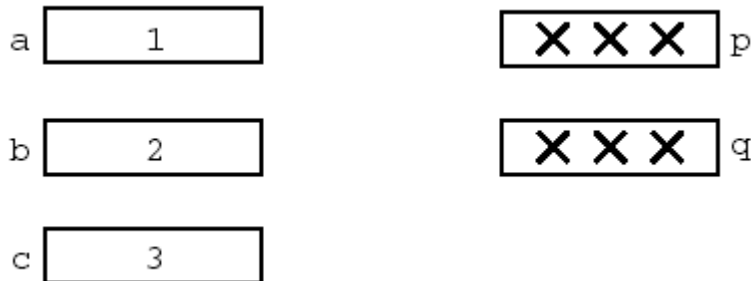


# Pointer Example

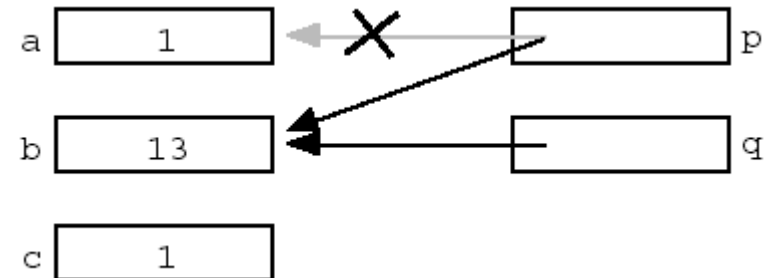
```
p = &a; // set p to refer to a  
q = &b; // set q to refer to b
```

```
void PointerTest() {  
    // allocate three integers  
    and two pointers
```

```
    int a = 1;  
    int b = 2;  
    int c = 3;  
    int* p;  
    int* q;
```

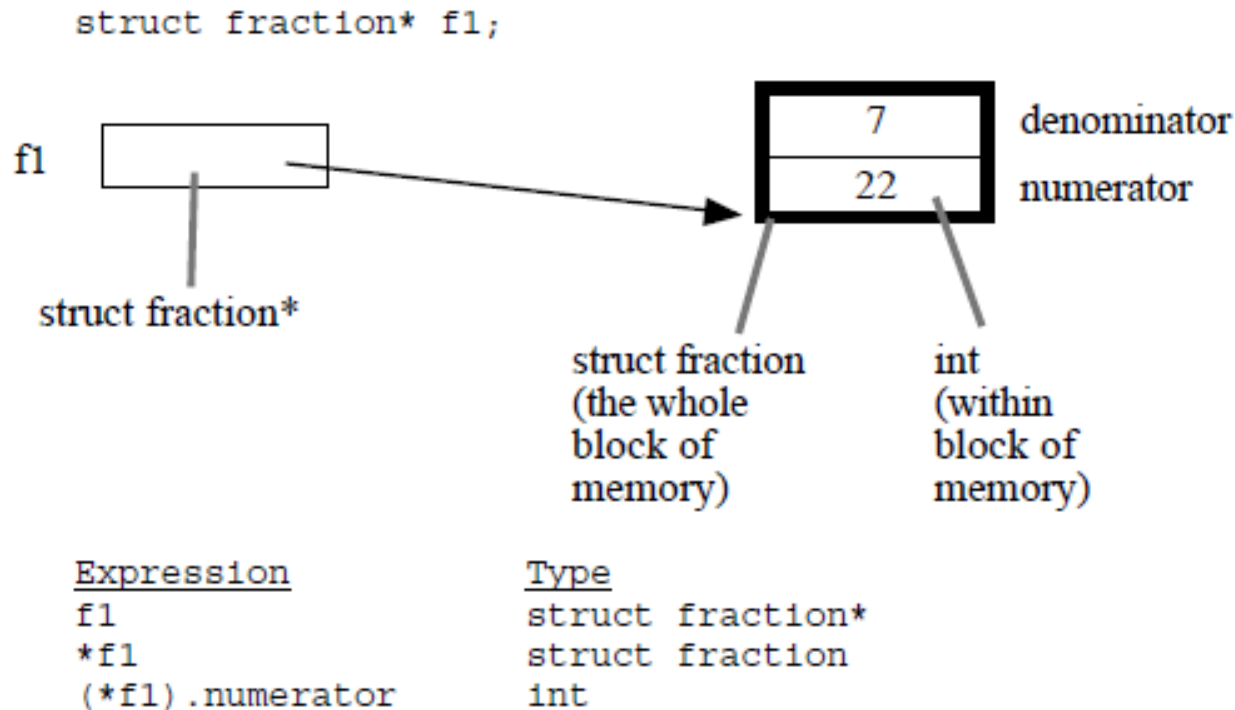


```
    c = *p;  
    p = q;  
    *p = 13;
```



```
    }
```

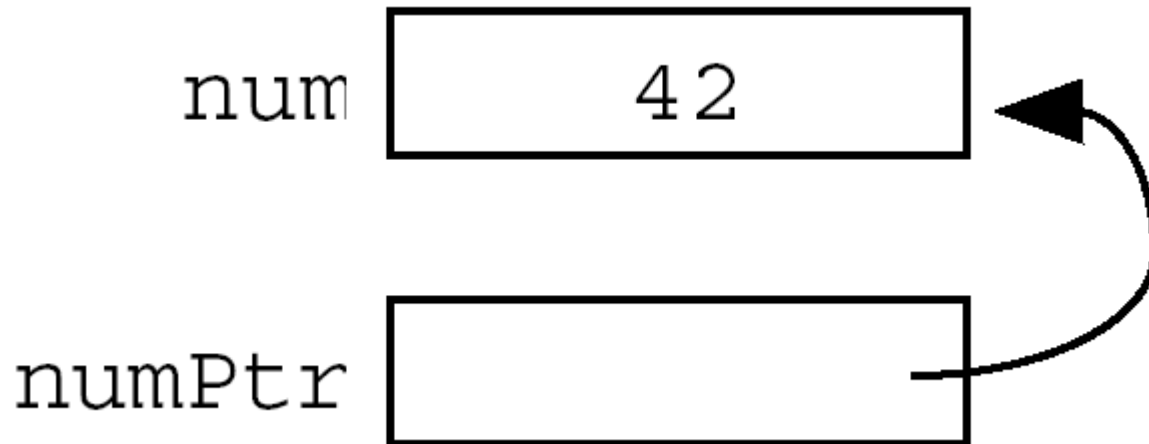
# Pointer to Structures



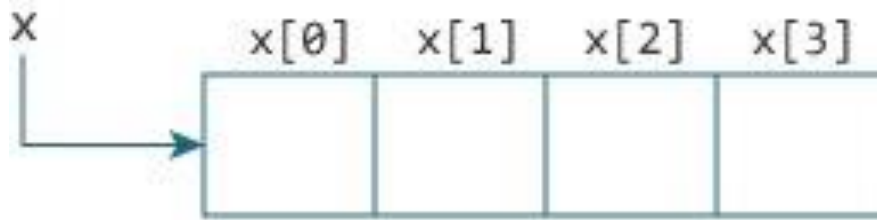
- ❑ There's an alternate, more readable syntax available for dereferencing a pointer to a struct.
- ❑ A `"->"` at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written `f1->numerator`.

# Pointers

```
void NumPtrExample() {  
    int num;  
    int* numPtr; numPtr is a pointer to int  
    num = 42;  
    numPtr = &num;  
    // Compute a reference to "num", and store it in numPtr  
    // At this point, memory looks like drawing below  
}
```



# Pointers and Arrays



- ❑ There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).
- ❑ Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.
- ❑ `x[0]` is equivalent to `*x`.
- ❑ `&x[j]` is equivalent to `x+j` and `x[j]` is equivalent to `*(x+j)`.

# Example

```
#include <stdio.h>
int main() {
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;
    // ptr is assigned the address of the third element
    ptr = &x[2];
    printf("*ptr = %d \n", *ptr);    // 3
    printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
    printf("*(ptr-1) = %d", *(ptr-1)); // 2
    return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

# Example

```
{
char string[1000]; // string is a local 1000 char array
int len;
strcpy(string, "binky");
len = strlen(string);
/*
    Reverse the chars in the string:
    i starts at the beginning and goes up
    j starts at the end and goes down
    i/j exchange their chars as they go until they meet
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
temp = string[i];
string[i] = string[j];
string[j] = temp;
}
// at this point the local string should be "yknib"
}
```



# Pointer to a Pointer

```
int main () {  
    int  var;  
    int  *ptr;  
    int  **pptr;  
    var = 3000;  
    ptr = &var;  
  
    pptr = &ptr;  
    printf("Value of var = %d\n", var );  
    printf("Value available at *ptr = %d\n", *ptr );  
    printf("Value available at **pptr = %d\n", **pptr);  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result -

Value of var = 3000

Value available at \*ptr = 3000

Value available at \*\*pptr = 3000

# Switch

```
switch (menu) /* select the type of calculation */
{
    case 1: total = numb1 + numb2;
    calType = '+'; /* assign a char to
symbolise calculation type */
    break;
    case 2: total = numb1 - numb2;
    calType = '-';
    break;
    case 3: total = numb1 * numb2;
    calType = '*';
    break;
    case 4: total = numb1 / numb2;
    calType = '/';
    break;
    default: printf("Invalid option selected\n");
}
```

# Ternary Operator

`<expression1> ? <expression2> : <expression3>`

- ❑ This is an expression, not a statement, so it represents a value.
- ❑ **An expression is something that returns a value, whereas a statement does not.**
- ❑ The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2
- ❑ Otherwise, it evaluates and returns expression3.
- ❑ Example:

```
min = (x < y) ? x : y;
```

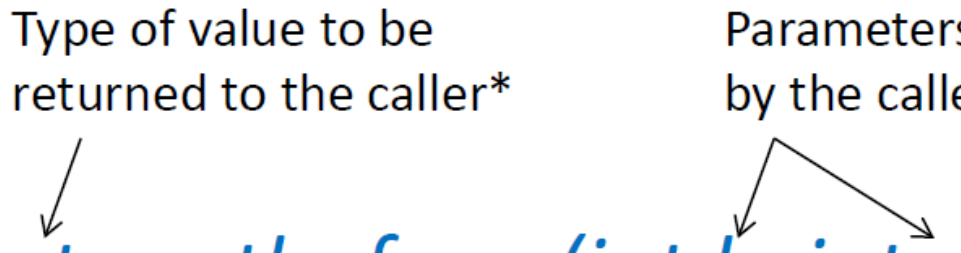
# C Functions

- ❑ Functions partition large programs into a set of smaller tasks
- ❑ Helps manage program complexity
- ❑ Smaller tasks are easier to design and debug
- ❑ Functions can often be reused instead of starting over
- ❑ Can use “libraries” of functions developed by 3rd parties as opposed to designing your own
- ❑ A function is “called” by another program to perform a task
  - The function may return a result to the caller
  - One or more arguments may be passed to the function/procedure

# Function Definition

Type of value to be  
returned to the caller\*

Parameters passed  
by the caller



```
int math_func (int k; int n)  
{  
    int j;           //local variable  
    j = n + k - 5;   //function body  
    return(j);       //return the result  
}
```


\* If no return value, specify "void"

# Function Arguments

- ❑ Calling program can pass information to a function in two ways
- ❑ By value: pass a constant or a variable value
  - function can use, but not modify the value
- ❑ By reference: pass the address of the variable
  - function can both read and update the variable
- ❑ Values/addresses are typically passed to the function by pushing them onto the system stack
- ❑ Function retrieves the information from the stack

# Example – Pass by Value

```
/* Function to calculate x2 */  
int square ( int x ) { //passed value is type int, return an int value  
    int y;             //local variable – scope limited to square  
    y = x * x;          //use the passed value  
    return(y);          //return the result  
}  
  
void main {  
    int k,n;            //local variables – scope limited to main  
    n = 5;  
    k = square(n);      //pass value of n, assign n-squared to k  
    n = square(5);      // pass value 5, assign 5-squared to n  
}
```



# Example

```
void func(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n",
a, b);
}
int main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    func(x, y);
    printf("In main, x = %d y = %d\n",
x, y);
    return 0;
}
```

Output:

In func, a = 12 b = 7

In main, x = 5 y = 7



# Example – Pass by Reference

```
/* Function to calculate x2 */  
void square ( int x, int *y ) { //value of x, address of y  
    *y = x * x; //write result to location whose address is y  
}  
  
void main {  
    int k,n; //local variables – scope limited to main  
    n = 5;  
    square(n, &k); //calculate n-squared and put result in k  
    square(5, &n); // calculate 5-squared and put result in n  
}
```

In the above, *main* tells *square* the location of its local variable, so that *square* can write the result to that variable.

# Example

## Call By Reference

```
void main()
{
    int n = 10;
    func(&n);
    printf( "%d", n );
}
```

```
void func ( int *add )
{
    *add = 20;
}
```

n  
10 20  
#2008

add  
2008  
#4016

Actual value of n  
changes to 20



# Find Maximum Value in An Array

```
#include <stdio.h>
#include <conio.h>
max(int [],int);
void main()
{
int a[]={10,5,45,12,19};
int n=5,m;
clrscr();
m=max(a,n);
printf("\n MAXIMUM NUMBER
IS %d",m);
getch();
}
```

```
int max(int x[],int k)
{
int t,i;
t=x[0];
for(i=1;i<k;i++)
{
if(x[i]>t)
t=x[i];
}
return(t);
}
```