Introduction to C Programming

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

ELSEVIER

C Chapter :: Topics

- Introduction to C
- Why C?
- Example Program
- Compiling and running a C Program
- Variables and Data types
- Operators
- Control Statements
- Programming the PIC32

Introduction to C

- Developed by Dennis Ritchie and Brian Kernighan at Bell labs in 1972.
- Motivation: rewriting UNIX (used to be in assembly language).
- Many languages derived from C: C++, C#, Objective C.
- By many measures, C is the most widely used language in existence.

Why C?

- Availability for variety of platforms (supercomputers down to embedded microcontrollers)
- Relative ease of use
- Huge user base
- Ability to directly interact with hardware
- Allows us to write at a high-level but still retain the power of assembly language (i.e., the programmer still has a good idea of how the code will be executed)

C Code Example 1

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C Code Example 1

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}
```

Header: header file needed for using printf function

Main: the main function must be included in all C programs

Body: prints "Hello world!" to the screen

Compiling and Running a C Program

- **Many C compilers exist**
  - cc (C compiler)
  - gcc (GNU C compiler)
  - PIC32 C compiler (will use in lab)
- **Create text file (i.e., "hello.c")**
- **At command prompt, type: gcc hello.c**
- **Executable is created: a.out (a.exe on Windows)**
- **At command prompt, type: ./a.out (or ./a.exe)**

Programmer support

- **comments: `//` and `/*  */`**
  - Organization
  - Readability
  - Usability (by yourself and others)
- **`#define`:** enables organization, ease of modification, and the avoidance of "magic numbers"
- **`#include`:** gives access to common functions

Comments

- **Single-line comments: //**

```
// this is an example of a one-line comment.
```

- **Multi-line comments: /* */**

```
/* this is an example
   of a multi-line comment */
```

#define

- **Format:**

```
#define identifier replacement
```

- **All instances of `identifier` will be replaced by `replacement` before compilation**

- **Example:**

```
#define MAXGUESSES 3
...
if (num < MAXGUESSES)
   ...
```

#include

- **Gives access to common functions:**
  - **provided by built-in libraries**
  - **written by you or others**

- **Examples:**

```
#include <stdio.h>
#include “mypath/myfile.h”
```

Variables

- **Each variable has:**
  - Type
  - Name
  - Value
  - Memory Location

- **Note:** variable names are case sensitive and cannot begin with a number or contain special characters (except underscore _).

Variables: Example

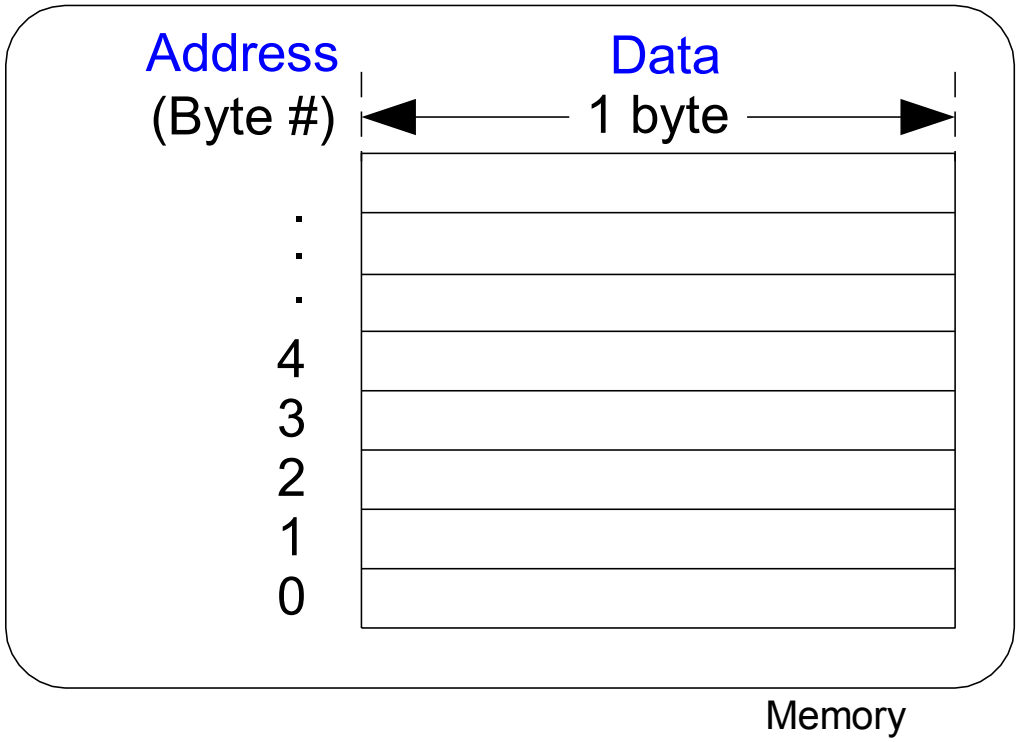
```
short y;    // type = short, name = y

y = 25;     // value = 25
```

- What about the memory location?

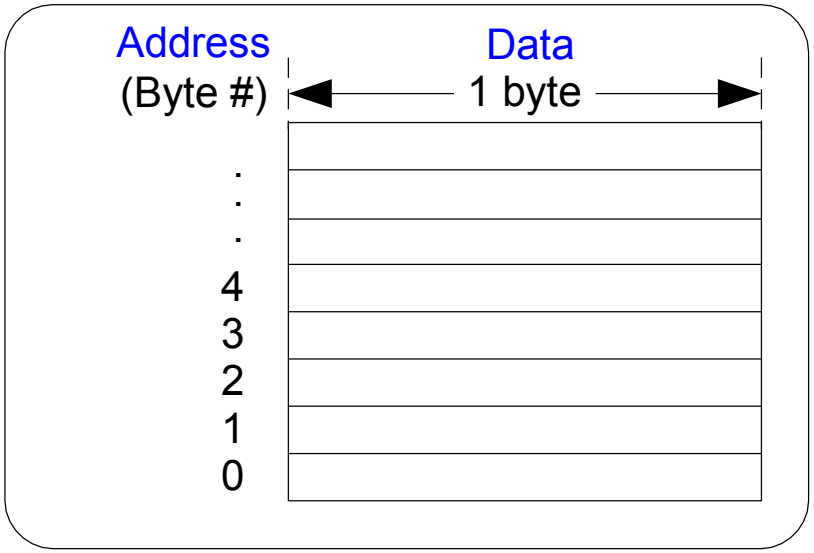
Memory

- C views memory as a group of consecutive bytes.
- Each byte has a unique number (the address)



Memory

- **The variable type indicates:**
  - How many bytes the variable occupies
  - How to interpret the bytes
- **The address of a variable occupying multiple bytes is the lowest numbered byte.**

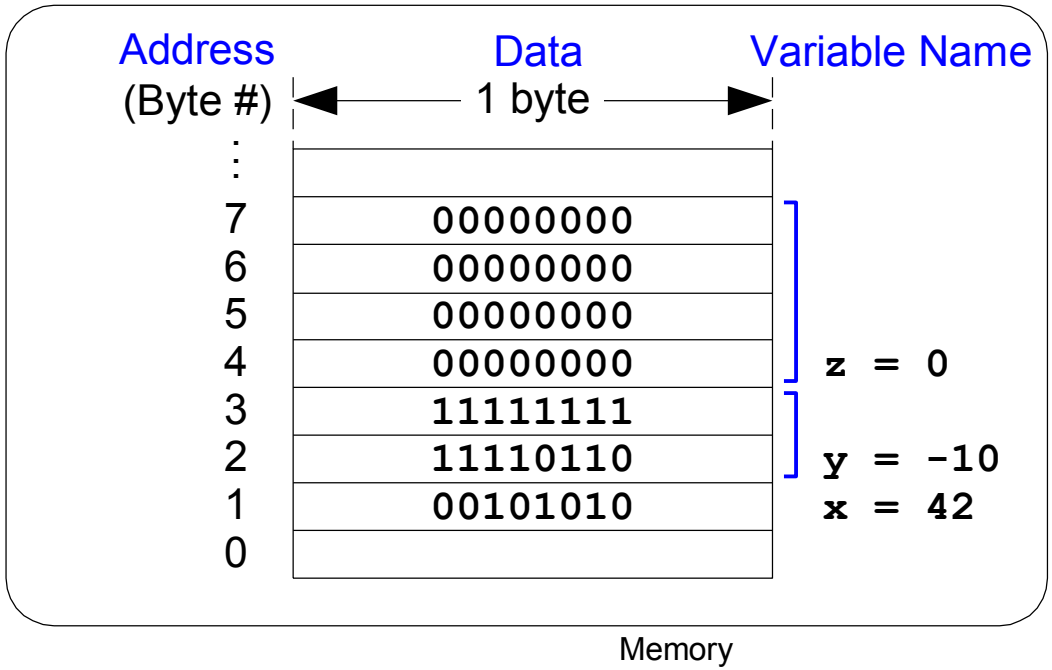


C's Data Types and sizes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

C Code Example 2

```
unsigned char x = 42; // x = 00101010
short y = -10;        // y = 11111111 11110110
unsigned long z = 0;  // z = 00000000 00000000
                      //     00000000 00000000
```



Operators: Part 1

Category	Operator	Description	Example
Monadic	++	post-increment	`a++; // a = a+1`
	--	post-decrement	`x--; // x = x-1`
	&	memory address of a variable	`x = &y; // x=the memory // address of y`
	~	bitwise NOT	`z = ~a;`
	!	Boolean NOT	`!x`
	-	negation	`y = -a;`
	++	pre-increment	`++a; // a = a+1`
	--	pre-decrement	`--x; // x = x-1`
	(type)	casts a variable to (type)	`x = (int) c; // cast c to an // int and assign it to x`
	sizeof()	size of a variable in bytes	`long int y; x = sizeof(y); // x = 4`
Multiplicative	*	multiplication	`y = x * 12;`
	/	division	`z = 9 / 3; // z = 3`
	%	modulo	`z = 5 % 2; // z = 1`
Additive	+	addition	`y = a + 2;`
	-	subtraction	`y = a - 2;`

Operators: Part 2

Category	Operator	Description	Example
Bitwise Shift	<<	bitshift left	`z = 5 << 2; // z = 0b0001 0100`
	>>	bitshift right	`x = 9 >> 3; // x = 0b0000 0001`
Relational	==	equals	`(y == 2)`
	!=	not equals	`(x != 7)`
	<	less than	`(y < 12)`
	>	greater than	`(val > max)`
	<=	less than or equal	`(z <= 2)`
	>=	greater than or equal	`(y >= 10)`
Bitwise	&	bitwise AND	`y = a & 15;`
	|	bitwise OR	`y = a | b;`
	^	bitwise XOR	`y = 2 ^ 3;`
Logical	&&	Boolean AND	`(x && y)`
	||	Boolean OR	`(x || y)`
Ternary	? :	ternary operator	`y = x ? a:b; // if x is true, // y=a, else y=b`

Operators: Part 3

Category	Operator	Description	Example
Assignment	=	assignment	`x = 22;`
	+=	addition and assignment	`y += 3;  // y = y + 3`
	-=	subtraction and assignment	`z -= 10;  // z = z - 10`
	*=	multiplication and assignment	`x *= 4;  // x = x * 4`
	/=	division and assignment	`y /= 10;  // y = y / 10`
	%=	modulo and assignment	`x %= 4;  // x = x % 4`
	>>=	bitwise right-shift and assignment	`x >>= 5;  // x = x>>5`
	<<=	bitwise left-shift and assignment	`x <<= 2;  // x = x<<2`
	&=	bitwise AND and assignment	`y &= 15;  // y = y & 15`
	|=	bitwise OR and assignment	`x |= y;  // x = x | y`
	^=	bitwise XOR and assignment	`x ^= y;  // x = x^y`

Example 3

```
#include <stdio.h>
main()
{
int count = 0, loop;
loop = ++count; /* same as count = count + 1; loop = count; */
printf("loop = %d, count = %d\n", loop, count);

loop = count++; /* same as loop = count; count = count + 1; */
printf("loop = %d, count = %d\n", loop, count);

}
```

Sample Program Output

loop = 1, count = 1

loop = 1; count = 2

C Code Example 4

```
int x = 14; int y = 43; int z; // x = 0b1110, y = 0b101011
printf("x = %d, y = %d\n", x, y);
z = y / x;  printf("y/x = %d/%d = %d\n", y, x, z);
z = y % x;  printf("y %% x = %d %% %d = %d\n", y, x, z);
z = x && y; printf("x && y = %d && %d = %d\n", x, y, z);
z = x && 0; printf("x && 0 = %d && 0 = %d\n", x, z);
z = x || y; printf("x || y = %d || %d = %d\n", x, y, z);
z = x || 0; printf("x || 0 = %d || 0 = %d\n", x, z);
z = x & y;  printf("x & y = %d & %d = %d\n", x, y, z);
z = x | y;  printf("x | y = %d | %d = %d\n", x, y, z);
z = x ^ y;  printf("x XOR y = %d XOR %d = %d\n", x, y, z);
z = x << 2; printf("x << 2 = %d << 2 = %d\n", x, z);
z = y >> 3; printf("y >> 3 = %d >> 3 = %d\n", y, z);
x += 2;     printf("x += 2 = %d\n", x);
y &= 15;    printf("y &= 15 = %d\n", y);
```

C Code Example 3: Console Output

```
x = 14, y = 43
y/x = 43/14 = 3
y % x = 43 % 14 = 1
x && y = 14 && 43 = 1
x && 0 = 14 && 0 = 0
x || y = 14 || 43 = 1
x || 0 = 14 || 0 = 1
x & y = 14 & 43 = 10
x | y = 14 | 43 = 47
x XOR y = 14 XOR 43 = 37
x << 2 = 14 << 2 = 56
y >> 3 = 43 >> 3 = 5
```

Control Statements

- **Enables a block of code to execute *only* when a condition is met**
- **Conditional Statements**
  - if statement
  - if/else statement
  - switch/case statement
- **Loops**
  - while loop
  - do/while loop
  - for loop

if Statement

- **The if block executes only if the condition is true (in this case, when `input` is equal to 1).**

```
if (input == 1)
  result = data; // if block

x = 2;
```

if/else Statement

- **When the condition is true, the if block executes. Otherwise, the else block executes.**

```
if (a > b)
  return a;  // if block
else
  return b;  // else block
```

switch/case Statement

- **Depending on the value of the switch variable, one of the blocks of code executes.**

```
int amount, color = 3;
switch (color) {
  case 1:  amount = 100; break;
  case 2:  amount = 50;  break;
  case 3:  amount = 20;  break;
  case 4:  amount = 10;  break;
  default: printf("Error!\n");
}
```

switch/case Statement

- **Equivalent to a nested if/else statement**

```
int amount, color = 3;
switch (color) {
        case 1: amount = 100; break;
        case 2: amount = 50;  break;
        case 3: amount = 20;  break;
        case 4: amount = 10;  break;
        default: printf("Error!\n");
}
```

```
int amount, color = 3;

if      (color == 1) amount = 100;
else if (color == 2) amount = 50;
else if (color == 3) amount = 20;
else if (color == 4) amount = 10;
else    printf("Error!\n");
```

Control Statements

- **Enables a block of code to execute *only* when a condition is met**
- **Conditional Statements**
  - if statement
  - if/else statement
  - switch/case statement
- **Loops**
  - while loop
  - do/while loop
  - for loop

while Loop

- **The code within the while loop executes while the condition is true.**

```
// this code computes the factorial of 9
int i = 1, fact = 1;
while (i < 10) {
      fact = fact * i;
      i++;
}
```

do/while Loop

- **The do block executes while the condition is true. The condition is checked only after the do block is run once.**

```
// this code computes the factorial of 9
int i = 1, fact = 1;
do {
   fact = fact * i;  // do block
   i++;
} while (i < 10);
```

for Loop

- **Similar function as while and do/while but gives support for loop variable (in previous code, the variable `i`).**
- **General format:**

```
for (initialization; condition; loop operation)
```

for Loop

```
// This code computes the factorial of 9
int i, fact = 1;

for (i=1; i<10; i++)
     fact *= i;
```

Loops

- **Loops**
  - while loop

```
int i = 1, fact = 1;
while (i < 10) {
        fact = fact * i;
        i++;
}
```

  - do/while loop

```
int i = 1, fact = 1;
do {
        fact = fact * i;
        i++;
} while (i < 10);
```

  - for loop

```
int i, fact = 1;
 for (i=1; i<10; i++)
        fact *= i;
```

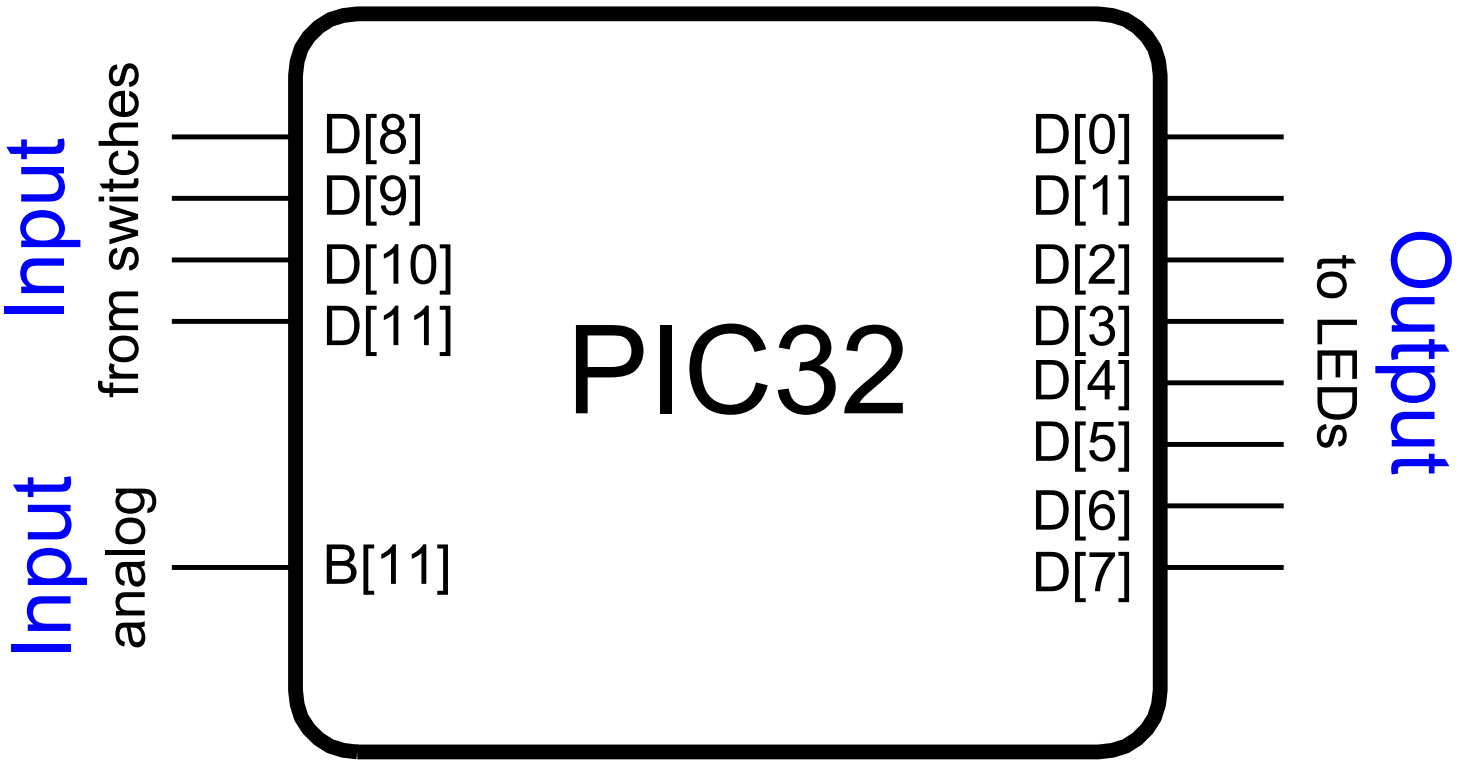
C Chapter :: Topics

- Introduction to C
- Why C?
- Example Program
- Compiling and running a C Program
- Variables and Data types
- Operators
- Control Statements
- Programming the PIC32

Programming the PIC32

- Physical pin connections
- Setting up (initializing) the pins
- Reading/Writing the pins

Programming the PIC32



Programming the PIC32

- **Included libraries define ports (groups of pins) and allow writing and reading from them**

```
#include <p32xxxx.h>
#include <plib.h>
#include <sys/appio.h>
```

- **Note: `int` and `unsigned int` are both 32 bits**

Predefined variables for accessing pins

- **`TRISD:`** for setting up port D as input or output
- **`PORTD:`** for reading from port D
- **`LATD:`** for writing to port D

Setting up the pins on PIC32

- **Pre-defined variable for setting the pins as outputs or inputs:**
  - **TRISD:** tristate of port D
    - **0:** output
    - **1:** input
- **Example 1: setting PORTD as inputs**

```
TRISD = 0xFFFFFFFF;
```

- **Example 2: setting PORTD as outputs**

```
TRISD = 0x00000000;
```

- **Example 3: D[7:0] as outputs, D[11:8] as inputs**

```
TRISD = 0xFFFFFF00;
```

Reading the pins on PIC32

- **Pre-defined variable used to read port D**
  - **PORTD**
- **Example 1: Reading PORTD**

```
int val = PORTD;
```

- **Example 2: Reading only D[3:0]**

```
int val = PORTD & 0x000F;
```

- **Example 3: Reading only D[11:8]**

```
int val = (PORTD >> 8) & 0x000F;
```

Writing the pins on PIC32

- **Pre-defined variable used to write port D**
  - **LATD**
- **Example 1: writing PORTD**

```
int val = 10;
LATD = val;
```

- **Example 2: Writing only D[7:0]**

```
int val = 10;
LATD = val & 0x00FF;
```

Extras

- **How to create a loop that executes forever:**

```
while (1) {
   ...
}
```

- **Generating random numbers:**

```
#include <stdlib.h>
int x, y;

x = rand();       // x = a random integer
y = rand() % 10; // y = rand integer from 0 to 9
```

C Programming Part 2

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

ELSEVIER

C Chapter :: Topics

- Function Calls
- Global vs. Local Variables
- More Data Types
  - Pointers
  - Arrays
  - Characters and Strings
- C Standard Libraries
- Compiler Options
- Command-Line Arguments

Functions

- Function calls enable **modularity**
- Like hardware modules, functions are defined by: input, output, operation (what they *do*)
- General function definition:

```
return type  function_name  (type arg1, type arg2){
  // function body – what it does
 }
```

- **return type:** output
- **function name:** describes what it does
- **arguments:** inputs
- **function body:** performs operation

Example Function

```
int sum3(int a, int b, int c) {
      int result = a+b+c;

      return result;
}
```

- return type: `int`
- function name: `sum3`
- arguments: `int a, int b, int c`
- function body: sums inputs and returns the result

Function without inputs/outputs

- Inputs and outputs are not required. For example:

```
void printPrompt()
{
  printf("Please enter a number from 1-3:\n");
  printf("\t1: $20\n");
  printf("\t2: $40\n");
  printf("\t3: $60\n");
}
```

Note: could also be written:

```
void printPrompt(void)
...
```

Function Prototypes

- In C, either the function or its prototype must be declared before the function is used.

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt();

int main()
{
  int y = sum3(10, 15, 20);
  printf("sum3() result: %d\n", y);
  printprompt();
}
// functions are defined further down on in code
```

Functions

- Write a function that returns the minimum of 3 integers. Also show how to call the function in your code.

Functions

- Write a function that returns the minimum of 3 integers. Also show how to call the function in your code.

```
int min3 (int a, int b, int c) {
      int tmpmin = a;
      if (b < tmpmin) tmpmin = b;
      if (c < tmpmin) tmpmin = c;
      return tmpmin;
}
# include <stdio.h>
int main ( ) {
int y = min3 (4, 3, 2);
printf ("min result: %d\n", y);
}
```

Globally vs. Locally Declared Variables

- **Global variable:**
  - declared outside of all functions
  - Accessible by all functions

- **Local variable:**
  - declared inside a function
  - Accessible only within function

Globally vs. Locally Declared Variables

```
// This program uses a global variable to find and print the
maximum of 3 numbers

int max;        // global variable holding the maximum value

void findMax(int a, int b, int c) {
  int tmpmax = a;  // local variable holding the temp max value
  if (b > tmpmax) {
    if (c > b) tmpmax = c;
    else       tmpmax = b;
  } else if (c > tmpmax) tmpmax = c;
  max = tmpmax;
}

int main(void) {
  findMax(4, 3, 7);
  printf("The maximum number is: %d\n", max);
}
```

More Data Types

- **Pointers:** a variable whose value is an address
- **Arrays:** collection of similar elements
- **Strings:** used for representing text

Pointers

- A variable whose value is an address
- Example:

```
int a = 10;
int *b = &a;    // b stores the address of a
```

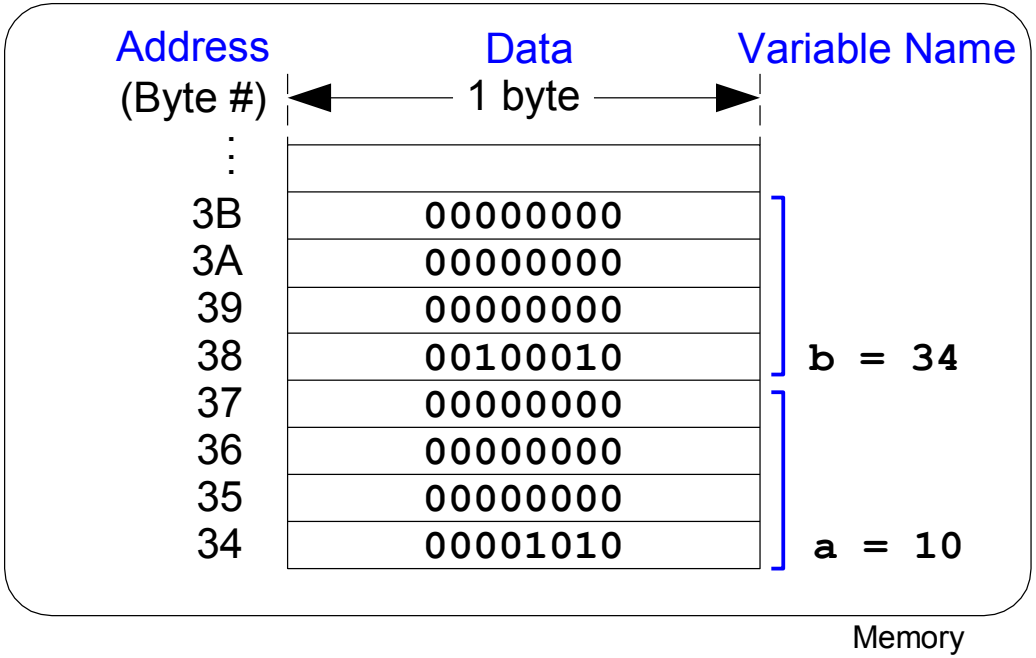
- Symbols:

`int *`	indicates that the variable is a pointer to an `int`
`&a`	returns the address of a

Pointers

- A variable whose value is an address
- Example:

```
int a = 10;
int *b = &a;    // b stores the address of a
```

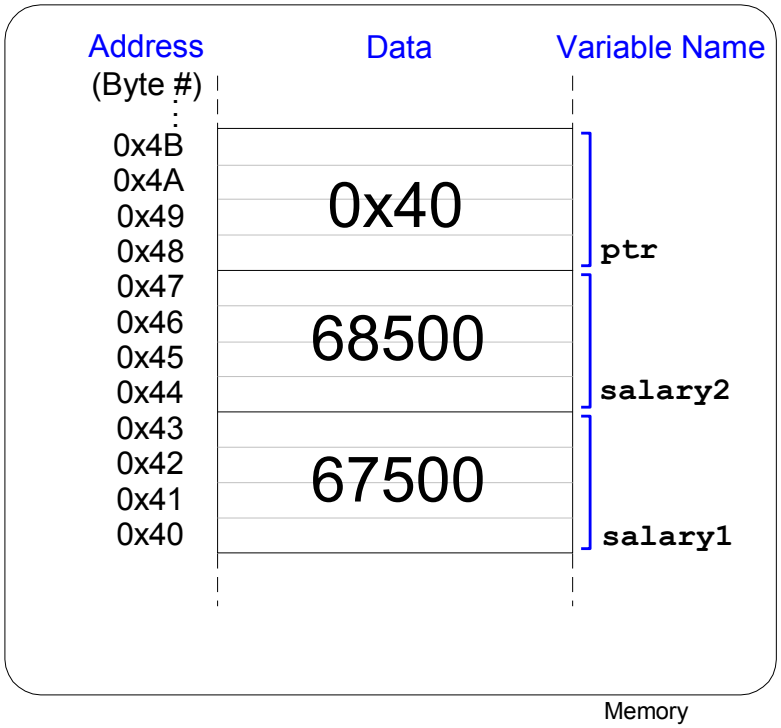


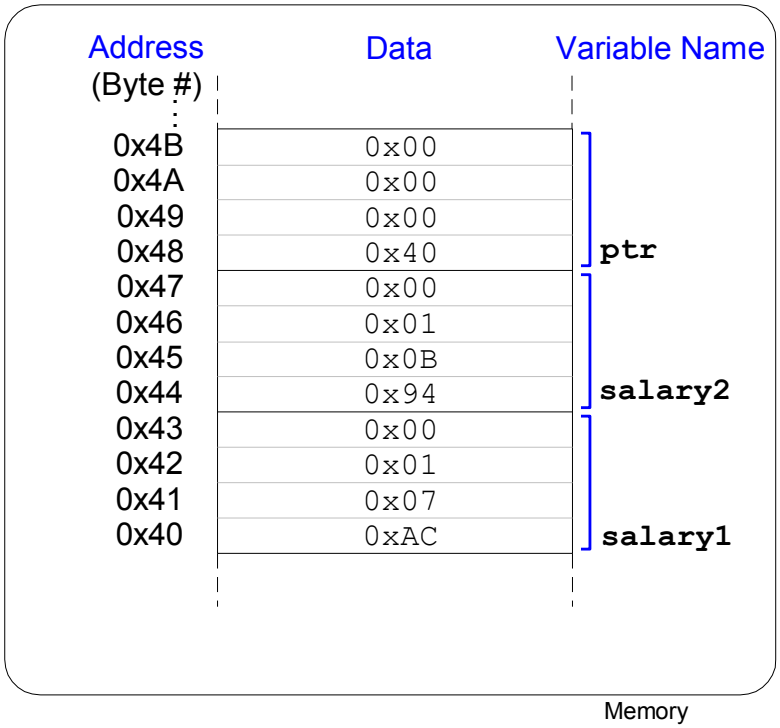
Pointer Example

```
// For concreteness, suppose salary1 is at address 0x40
unsigned long salary1, salary2; // 32-bit numbers
unsigned long *ptr;     /* a pointer specifying the
                           address of an unsigned long variable */

salary1 = 67500; //assign salary1 to be $67500 = 0x000107AC
ptr = &salary1; //assign ptr to be 0x0040, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of
                          address 40 = 67500, then add $1000
                          and set salary2 to $68500 */
```

Pointer Example





Pointers: Passing an input by reference

```
#include <stdio.h>

void quadruple(int *a)
{
        *a = *a * 4;
}

int main()
{
        int x;

        x = 5;
        printf("x before: %d\n", x);
        quadruple(&x);
        printf("x after: %d\n", x);

        return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

Arrays

- A collection of similar elements
- Example:

```
long scores[3]; // a 3-element array of longs
```

- This allocates 3 * 4 bytes = 12 bytes.
- Index goes from 0 to N-1 (N is # of elements)
  - `scores[0], scores[1], scores[2]`
- The value of scores is the address of element 0

Initializing Arrays

- At declaration:

```
// scores[0]=93; scores[1]=81; scores[2]=97;
long scores[3]={93, 81, 97};
```

- After declaration, 1 element at a time:
  - Individually:

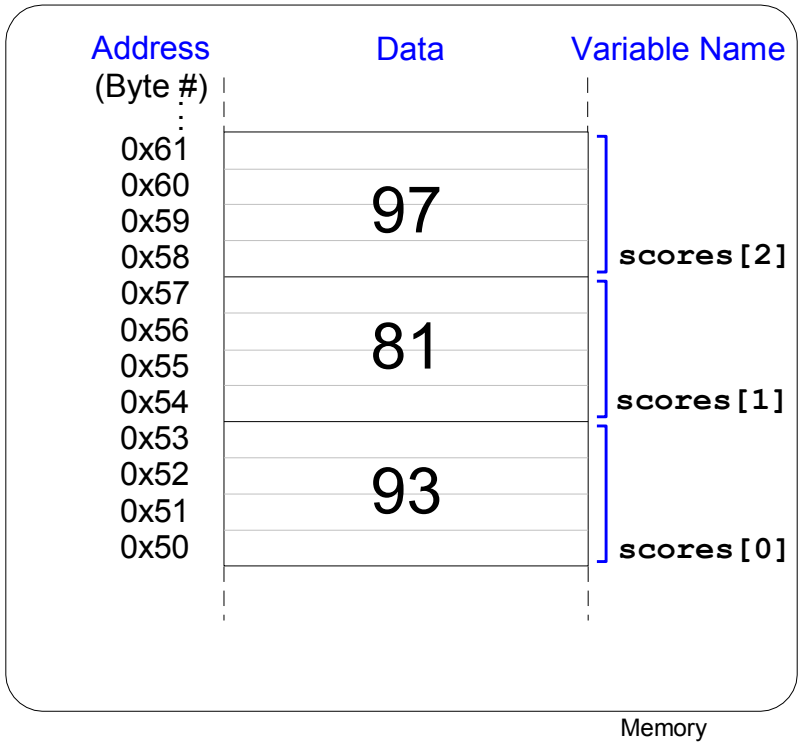
```
scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

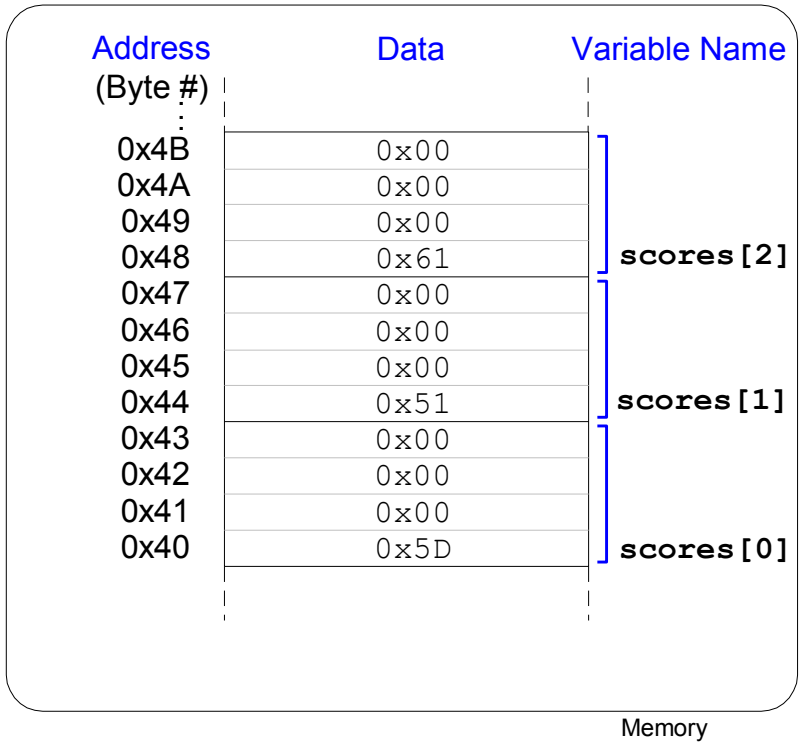
  - Using a for loop:

```
for (i=0; i<3; i++)
  scores[i] = 100-i;
```

Arrays in memory

```
long scores[3]={93, 81, 97};
```





- `scores` is of type `long *`
- The value of `scores` is 0x50

Passing an Array into a function

```
#include <stdio.h>
int getMean(int *arr, int len);
 int main() {
  int data[5] = {78, 14, 99, 27, 16};

  int avg = getMean(data, 5);
  printf("The average data value was: %d.\n", avg);
  return 0;
}

int getMean(int *arr, int len)
{
  int i, mean = 0;
   for (i=0; i < len; i++)
    mean += arr[i];
   mean = mean / len;
  return mean;
}
```

Arrays as input arguments

The following function prototypes could be used for the `getMean()` function:

```
int getMean(int *arr, int len);
int getMean(int arr[], int len);
int getMean(int arr[5], int len);
```

Another Array Example

```
short wombats[10]; // array of 10 2-byte quantities
                   // suppose they're stored at 0x20-0x33.
short *wombptr;    // a pointer to a short

wombats[0] = 342;  // store 342 in addresses 0x20-21
wombats[1] = 9;    // store 9 in addresses 0x22-23
wombptr = &wombats[0]; // wombptr = 0x20
*(wombptr+3) = 7;  /* offset of 3 elements, or 6 bytes. Thus
                      addresses 0x26-27 = 7, so this is another
                       way to write wombats[3] = 7. */
```

Strings

- Strings:
  - used for representing text
  - array of `chars`
  - last element of string is `NULL` character '`\0`'
- Example:

```
char strVar[10] = "Hello!";
```

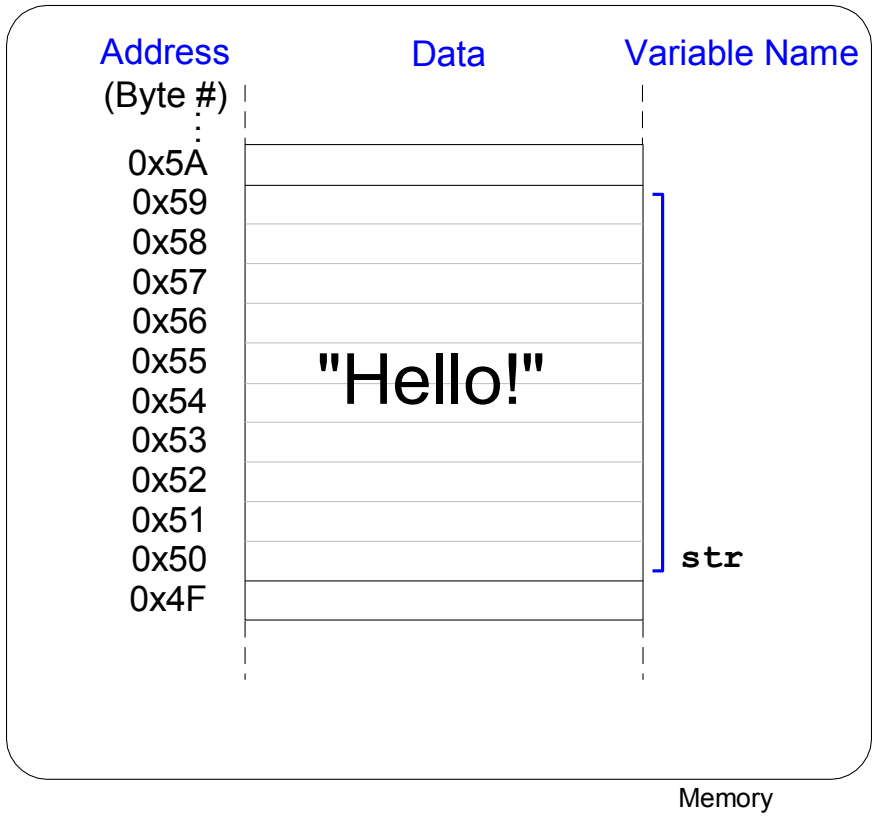
ASCII Codes

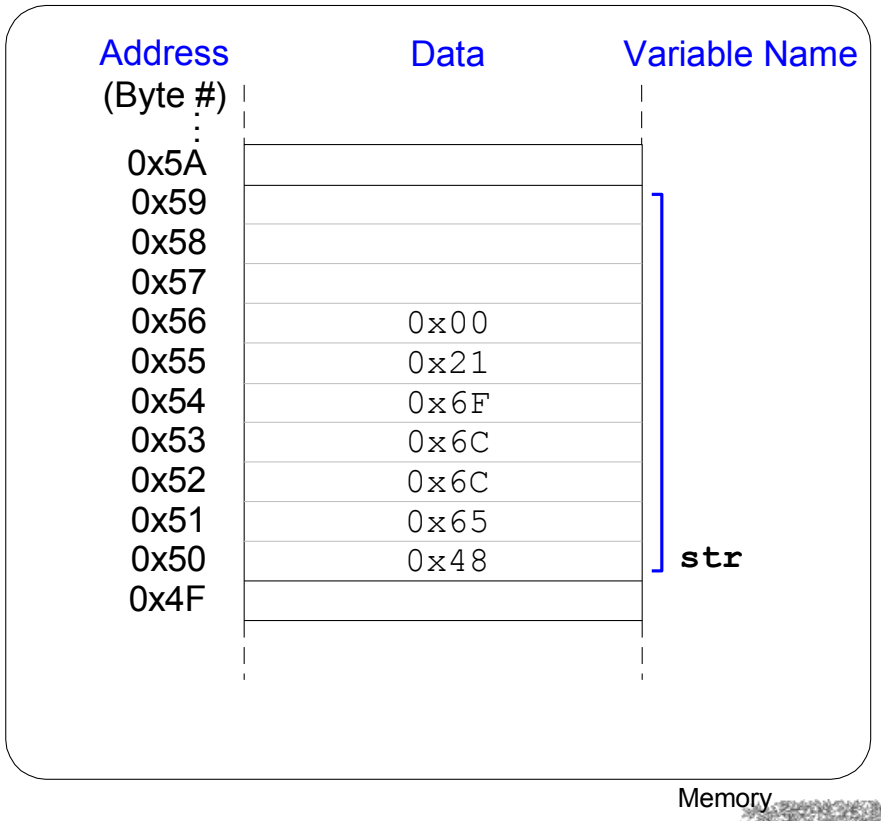
#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

American Standard Code for Information Interchange

String Stored in Memory

```
char strVar[10] = "Hello!";
```





Format codes

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating point number (float or double)
%e	Floating point number (float or double) in scientific notation (e.g., 1.56e7)
%c	Character (char)
%s	String (null-terminated array of characters)

Floating Point Examples:

```
float pi = 3.14159, e = 2.7182, c = 3e8,
y = 23444.3;

printf("pi = %3.2f\ne = %6.3f\nc = %5.3f
\ny = %10.0f\n", pi, e, c, y);
```

Console Output

```
pi = 3.14
e =   2.718
c = 300000000.000
y =       23444
```

Printing Strings

```
char *str = "Hello!";
printf("str: %s", str);
```

Console Output

```
str: Hello!
```

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating point number (float or double)
%e	Floating point number (float or double) in scientific notation (e.g., 1.56e7)
%c	Character (char)
%s	String (null-terminated array of characters)

C Standard Libraries

- Built-in functions provided by the compiler
  - `stdio.h`
  - `stdlib.h`
  - `math.h`

Standard Input/output Library

- `stdio.h`
  - Write the following line at the top of your code:

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

  - Includes functions such as:
    - `printf()`
    - `scanf()`

printf()

```
int a=4, b=5, c=17, d=14;

printf("a = %d, b = %d, c = %d, d = %d\n",
       a, b, c, d);
```

scanf()

```
int a;
char str[80];
float f;

printf(“Enter an integer.\n”);"
scanf(“%d”, &a);
printf(“Enter a floating point number.\n”);
scanf(“%f”, &f);
printf(“Enter a string.\n”);
scanf(“%s”, str);
// note no & needed: str is a pointer to
// the beginning of the array
```

Standard Library

- `stdlib.h`
  - Write the following line at the top of your code:

```
#include <stdlib.h>
```

  - Includes functions such as:
    - `rand()`
    - `srand()`

rand()

```
#include <stdlib.h>
int x, y;

x = rand();   // x = a random integer
y = rand() % 10;
// y = a random number from 0 to 9
```

srand()

```
#include <stdlib.h>
#include <time.h>

int main()
{
  int x;

  srand(time(NULL));     // seed the random number
generator
  x = rand() % 10;       // random number from 0 to 9
  printf("x = %d\n", x);
}
```

Math Library

- `math.h`
  - Write the following line at the top of your code:

```
#include <math.h>
```

```
#include <stdio.h>
#include <math.h>
float a, b, c, d, e;
a = cos(0);              // 1, note: the input argument is in radians
b = 2 * acos(0);      // pi
c = sqrt(144);        // 12
d = log10(1000);      // 3
e = floor(178.567);   // 178, rounds to next lowest whole number
printf("a = %f, b = %f, c = %f, d = %f, e = %f\n", a, b, c, d, e);
```

Console Output

```
a = 1.000000, b = 3.141593, c = 12.000000, d = 3.000000, e =
   178.000000
```

Compiler Options

Compiler Option	Description	Example
`-o outfile`	specifies output file name	`gcc -o hello hello.c`
`-S`	create assembly language output file (not executable)	`gcc -S hello.c` `note: this produces hello.s`
`-v`	verbose mode – prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-Olevel`	specify the optimization level (level is typically 0 through 3)	`gcc -O3 hello.c`
`--version`	list the version of the compiler	`gcc --version`
`--help`	list all command line options	`gcc --help`

Command Line Arguments

- **`argc`:** argument count
- **`argv`:** argument vector

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;

  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```