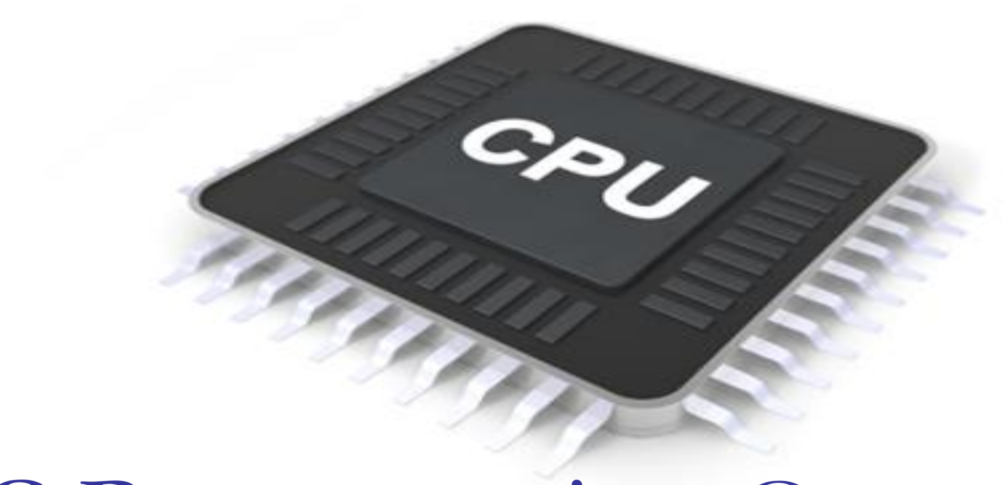


C Programming Overview

ENEE 3587

Microp Interfacing

Main Topics Contents

High vs Low Level Languages

- LLL:
  - Assembly Language
  - Native language supported by the CPU
  - Implemented as a circuit
  - Assembler
- HLL:
  - C/C++, JAVA, Python, etc.
  - Abstract logical concepts expressed in English language
  - Platformless
  - Requires a compiler t

C Language

- A relatively small language
- Disadvantage:
  - lacks features
  - Required library functions for basic tasks such as input/output (I/O
  - Many tasks unsupported by C standard such as graphics must be interfaced
  - Doesn’t try to protect programmer from mistakes
- Advantage:
  - Easy to learn
  - Less restrictive
  - Close to LLL: great for hardware control
  - Compiles fast and efficiently

Online Compiler and Program Template

- https://www.onlinegdb.com/
  - Make sure that C is chosen in Language Select
- Program Template:

```
include <stdio.h>          //standard IO functions
//add more includes as needed
//define global variable here if needed

void main()
{
   //main function body
}
```

Data Storage on a Computer

- Byte: 8 bits
- Word (2 Bytes): 16 bit
- double Word (4 bytes): 32 bit
- Quad word (8 bytes): 64 bit

C Language Data Types

- `char`: a character
  - a byte (8 bit storage)
- `Int`: an integer
  - 16 bit
- `float`: a real number
  - Aka floating-point number
  - 32 bit precision
- `double` a floating-point number
  - 64 bit precision

Type Modifiers

- Modifiers control the amount of storage associated with each type
- There are 5 modifiers
  - `short`
  - `long`
  - `signed`
  - `unsigned`
  - `long long`
- Used in as prefixes before types

Types	Range	Storage	
		Bytes	Bits
char	–127 to 127	1	8
int	–32,767 to 32,767	2	16
float	1E–37 to 1E+37 with six digits of precision	4	32
double	1E–37 to 1E+37 with ten digits of precision	8	64
long double	1E–37 to 1E+37 with ten digits of precision	10	80
long int	–2,147,483,647 to 2,147,483,647	4	32
short int	–32,767 to 32,767	2	16
unsigned short int	0 to 65,535	2	16
signed short int	–32,767 to 32,767	2	16
long long int	–(2power(63) –1) to 2(power)63 –1	8	64
signed long int	–2,147,483,647 to 2,147,483,647	4	32
unsigned long int	0 to 4,294,967,295	4	32
unsigned long long int	2(power)64 –1	8	64

Constants aka Immediates

- Characters:
  - E.g. `'A', 'c'`
  - Use single quotes
  - Each character has a byte integer value
- Strings:
  - A sequence (array) of characters
  - Use double quotes
  - E.g. `"Hello World"`
- Integer constants:
  - E.g. `123`, `456`
  - Default to smaller size
  - Forced long ints using L suffix: `1234L`
- Float/double
  - E.g. `123.45`, `1.234E-5`

ASCII

- American Standard Code for Information Interchange
- A character encoding standard
- Each character is a byte value
- Table : https://en.cppreference.com/w/c/language/ascii
  - `'0'-'9'`: 48-57
  - `'A'-'Z'`: 65-90
  - `'a'-'z'`: 97-122

Special Characters

- `\n` a ``newline'' character
- `\b` a backspace
- `\r` a carriage return (without a line feed)
- `\'` a single quote (e.g. in a character constant)
- `\"` a double quote (e.g. in a string constant)
- `\\` a single backslash
- Example usage: `"he said \"hi\""`

Identifier

- User defined name (a single English word):
  - Data variables
  - Function names
- Data variable:
  - user defined name
  - to store data
  - used for computation in the program
- Identifier:
  - Consist of letters, numbers, and underscores.
  - Must start with a letter
  - As long as you want (theoretically), but compilers truncate very long names
  - Case sensitive
  - Must not be a reserved word

C Language Reserved Word

- Words used in C for specific function
- 32 keywords:

  ```
  if, else, switch, case, default;
  break ;
  int, float, char, double, long;
  for, while, do, void, goto;
  auto, signed, const, extern, register, unsigned ;
  return ;
  continue ;
  enum ;
  sizeof ;
  struct, typedef ;
  union;
  volatile
  ```

Variables

- Variables are:
  - User defined identifiers.
  - Used to store values,
  - of specific Type
  - The value can change,
  - but the type can’t change

Variable Declaration

- Each variable must be declared
- Format: type identifier [=constant];
  - type: `char`, `int`, `float`, `double`
  - Can use modifiers: long, short, signed, unsigned,
  - Identifier: variable name
  - Initialized: =constant
  - []: Doesn’t have to initialized.
  - `;` semicolon used in C to terminate each line.
- Multiple variables can be defined on one line
  - Separated by comma.
  - Semicolon to terminate line.

Variable Declaration Examples

```
int var1;
float var2=1.5, var3;
char alpha;
```

Arrays

- Arrays are:
  - Data structure (variable)
  - of specific type
  - that stores a fixed size
  - of sequential collection of elements.
  - The elements and sequence can change
  - but their type can’t
- Strings are arrays of characters

Arithmetic Operators

`++, --`	increment, decrement
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus (remainder)

- Increment and decrement are short-hand for
  - var++ //var = var+1
  - var-- //var = var–1
- Division on integers discards any remainder
- Modulus can only be applied to integers
- Constant math expression are permitted but will follow precedence

Logical Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise NOT
>>	logical binary shift right
<<	logical binary shift left

p	q	~p	~q	p & q	p | q	p ^ q
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	1	0	1	1	1	0
1	0	0	0	0	1	1

- Shift:
  - Allows you to shift multiple times
  - Shift left once is like multiply by 2
  - Shift right once is like integer division by 2
  - Examples:

```
5<<2;          //shift 5 left twice:        5*2*2 = 20
100>>3;        //shift 100 right thrice:   100/2/2/2 = 50/2/2 = 25/2 = 12
```

Set/Clear/Complement a Bit

- Set: force to 1
  - Bitwise OR with 1
  - ORing with 0 has no effect
- Clear: force to 0
  - Bitwise AND with 0
  - ANDing with 1 has no effect
- Complement: flip bit
  - Bitwise XOR with 1
  - XORing with 0 has no effect

Examples: Bitwise Clear/Set/Complement

```
var1 &= 0x80 ; //var1 = var1 & 0x80. clear all but the msb
var1 &= ~0x80; //var1 = var1 & 0x7F. Clear the msb
var1 |= 0x80 ; //var1 = var1 | 0x80. Set the msb
var1 |= ~0x80; //var1 = var1 | 0x7F. Set all bits but the msb
var1 ^= 0x80 ; //var1 = var1 ^ 0x80. Complement the msb
var1 ^= ~0x80; //var1 = var1 ^ 0x7F. Complement all but the msb
```

Precedence of Math Ops

- When more than one operator is used in the same line
- Order of precedence:
  1. Parenthesis
  2. Increment and decrement. Left to Right.
  3. Negative and Positive.
  4. Multiplication and Division and Modulus. Left to Right.
  5. Addition and Subtraction. Left to Right.
- Examples:

```
1+2./3-6%5    //same as: 1+(2/3)-(6%5) = 1+0.67-1 = 0.67
6*2/3%4       //same as: ((6*2)/3)%4 = (12/3)%4 = 4%4 =0
```

Variables: Assignment and Mathops

- Assignment: variable = value;
- Use math operations on variables
- The type doesn’t change
- Example:

```
int y, x, a, b, c;
a = 2;
b = 3;
c = 5;
y = a*x*x + b*x + c;       //y = 2x^2 + 3x + 5
```

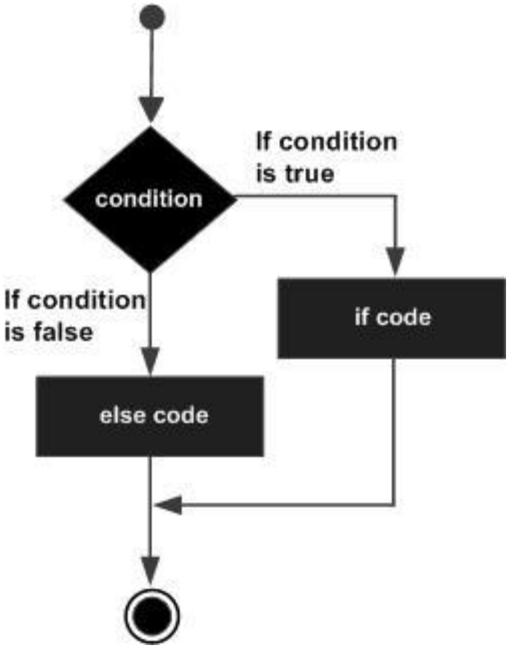
Input and Output

- In microcontroller applications we will not be using stdio for input/output

Advanced Math Operations

- Include `math.h` for these advanced ops:
  - `cos(double x)`: cosine of x rads
  - `sin(double x)`: sine of x rads
  - `tan(double x)`: tan of x rads
  - `acos(double x)`:arc cos of x in rads
  - `asin(double x)`:arc sine of x in rads
  - `atan(double x)`:arc tan of x in rads
  - `exp(double x)`: exponent of x
  - `pow(double x, double y)`: x^y
  - `sqrt (double x)`: $\sqrt{x}$
  - `log (double x)`: natural log of x
  - `log10(double x)`: common log of x
  - `fceil(double x)`: round x up
  - `floor(double x)`: round x down
  - `fabs(double x)`: absolute value of x
  - `fmod(double x, double y)`: remainder of x/y

If statement



- Format:

```
if(boolean_expression)
   {
   then_statement(s);
   }
else
   {
   else_statement(s);
   }
```

- Boolean expression: aka *control*
- then_statements: execute if the Boolean expression is TRUE
- else_statements: execute if the Boolean expression is FALSE
- { } are not needed if there is only 1 statement

Boolean Expressions

- Evaluates a TRUE or FALSE
  - FALSE = 0
  - TRUE = not 0 (anything not zero)
- Use to determine if an action should be taken
  - i.e. for purposes of control
- Boolean expressions types:
  - Are the values equal
  - Are the values less than
  - Are the value greater than
  - Are multiple conditions all TRUE/FALSE
  - Is one of these conditions TRUE/FALSE

Logical Assignment Operators

Op	Description	Example
`==`	TRUE if both side are equal	`(A == B)`
`!=`	TRUE if both side are not equal	`(i != 5)`
`>`	TRUE if left side is greater than right side	`(x > y)`
`<`	TRUE if left side is less than right side	`(b < 10.0)`
`>=`	TRUE if left side is greater than or equal to right side	`(A >= C)`
`<=`	TRUE if left side is less than or equal to right side	`(A <= B)`

If examples

```
  int a = 5, b = 20;

  if ( a == b )
     printf("Line 1 condition is true\n" );
  else
     printf("Line 1 condition is false\n" );

  if ( a > b )
     printf("Line 2 condition is true\n" );
  else
     printf("Line 2 condition is false\n" );

if ( a )
     printf("Line 3 condition is true. Weird.\n" );
  else
     printf("Line 3 condition is false. Weird.\n" );
```

Logical Operators

Operato r	Description	Example
&&	Logical AND Only when both operands are non-zero, condition is TRUE.	(A && B)
||	Logical OR Only when one operand is non-zero, condition is TRUE.	(A || B)
!	Logical NOT Reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A) !(A && B)

Examples of Logical Operators

```
int a = 5, b = 20, c = 10;

  if ( !(a < b) )
     printf("Line 1 condition is true\n" );
  else
     printf("Line 1 condition is false\n" );

  if ( a == b  && a > c)
     printf("Line 2 condition is true\n" );
  else
     printf("Line 2 condition is false\n" );

  if ( a == b  || a > c)
     printf("Line 3 condition is true\n" );
  else
     printf("Line 3 condition is false\n" );
```

else if statement

- You can include an if clause in your else statement:

```
if (a>5)
   printf("A is greater than 5");
else if (a<5)
   printf("A is less than 5");
else
   printf("A is equal to 5");
```

while Loop

- Format:

```
while (boolean expression)
{
        body_statement(s);
}
```

- Checks if a condition is TRUE **before** body is executed
- Loop: continues to execute the body as long as a condition is TRUE
- Typically, the body contains a statement that turns the condition FALSE
  - Otherwise, it will be an infinite loop!

while Example

```
int x = 2, n = 5;
while(x < 100)
{
      printf("%d\n", x);
      x *= 2;
}

while(n > 0)
{
      printf(“%d\n“,n);
      n--;
}
```

Signals

- Typical signals will be 8bits (sometimes 16bits)
  - MSB: bit 7
  - LSB: bit 0
  - If only bit n is on: signal = 2^n
    - For bits 0,1,2,3,4,5,6,7: Signal bit values would be: 1,2,4,8,16,32,64,128
    - Values in hex: 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
- Multiple bits of a signal e.g. bit n and bit m: 2^n + 2^m
  - E.g. bit 7 and bit 0 = 2^0 + 2^7 = 129
  - Easier to do in hex: bit 7 = 0x80; bit 0 = 0x01 => 0x81

Check if a Signal’s Bit is Set/Clear

- Check if a bit is Set (aka 1):
  - use AND operation
  - Operand has 1 for bit(s) in question
  - Format: `If (variable & operand)`

- Check if a bit is Clear (aka 0):
  - Complement the Set, i.e. `If !(variable & operand)`
    - Similar to `If (~variable | ~operand)`

Examples of Checking for signals

- Check if signal bit 0 is on/set (signal is 8 bits):

```
if (signal & 0x01)
```

- Check if signal bit 1 is off/clear

```
if !(signal & 0x02)
```

- Check if signal bit 0 and bit 7 is on/set

```
if (signal & 0x01) && (signal & 0x80)
```

- Check if signal bit 0 or signal bit 7 is on/set

```
if (signal & 0x01) || (signal & 0x80)
```

- Wait until signal bit 2 is on/set:

```
while !(signal &  0x04);
```

- Wait until signal bit 3 is off/clear:

```
while (signal &  0x08);
```

do … while Loop

- Format:

```
do
{
        body_statement(s);
}
while (boolean expression)
```

- Checks if a condition is TRUE **after** body is executed
  - Body is executed at least once
  - Body is executed again if condition is TRUE

Example

```
int x = 200, n = -5;
do {
     printf("%d\n", x);
     x *= 2;
}
while(x < 100)

do
{
     printf("%d\n",n);
     n--;
}
while(n > 0)
```

Programing Exercise: `while`, `if`

- Find if the integer value in Num is divisible by 3
  - If the remainder of Num/3 is 0 then it is divisible by 3
  - Algorithm?

Programing Exercise: `while`, `if`

- Find if the positive integer Num is prime.
  - A number is prime if it’s only divisible by 1 and itself.
  - Algorithm:
    1. prime = 1 (i.e. assume number is prime)
    2. Set denom = 2,
    3. Check if Num/denom has a remainder
    4. Increment denom
    5. Repeat step 3
    6. Stop when …?

`for` Loop

- Format:

```
for (initialization(s);  Boolean_expression;  continuation(s))
{
      body_statement(s);
}
```

- Boolean expression is the stopping condition
  - No Boolean expression = infinite loop!
- Initialization: assign initial values to your variables
  - Initialization happens **once** only **before** any looping and doesn't repeat
  - Multiple initializations separated by commas
- Continuation: assign how variables are to be updated
  - Updates are **after** **each** loop.
  - Multiple initializations separated by commas

Examples

```
for (int i = 0; i < 10; i ++)
    printf("i is %d\n", i);

for (i = 0, j=10; i != 10; i ++, j--)
    printf("\ni,j = %d,%d", i,j);

for (i=0; i<5; i++)
    for (j=5, j>0, j--)
        printf("\ni,j = %d,%d\n", i,j);
```

break

- Allows you to jump out of the current control structure
- Example

```
for (i=0; i<5; i++)
     for (j=5; j>0; j--)
     {
          printf("\ni,j = %d,%d\n", i,j);
          if (i==j)
              break;
     }
```

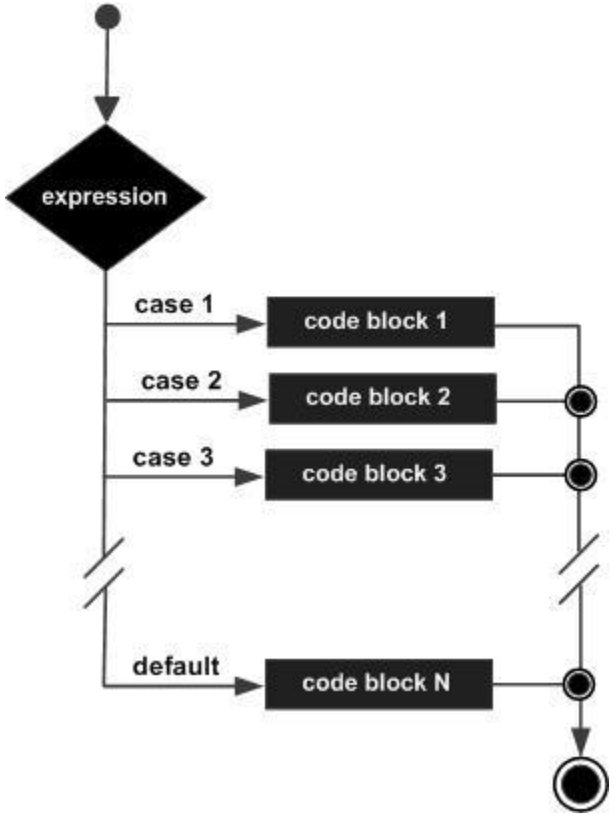
Programing Exercise: `for`

- Find if the positive integer Num is prime.
  - A number is prime if it’s only divisible by 1 and itself.
  - Algorithm:
    1. prime = 1 (i.e. assume number is prime)
    2. Set denom = 2,
    3. Check if Num/denom has a remainder
    4. Increment denom
    5. Repeat step 3
    6. Stop when …?

switch Statement

- Test a variable against a list of values
- Format:

```
switch(expression)
{
   case constant-expression1: statement(s); break;
   case constant-expression2: statement(s); break;
   ...
   default : statement(s);
}
```



- break is needed to exit the case

Example 2: switch

- A 20 pt grade distribution is organized as follows:
  A=20-19, B=18-17, C=16-15, D=14-13, F=13-0
  Write a switch statement to convert the 20point grade into a letter grade.

```
int grade;
char letter;

switch (grade)
{    case 20:
     case 19: letter = ‘A’; break;
     case 18:
     case 17: letter = ‘B’; break;
     case 16:
     case 15: letter = ‘C’; break;
     case 14:
     case 13: letter = ‘D’; break;
     default: letter = ‘F’; break;}
```

Arrays

- Sequence of data items
- same type
- stored contiguously in memory.

Declaration of Arrays

- Declaration of arrays: type indentifier[size] = {elements};
- Declaration of strings: char indentifier[size] = "characters";
- Specifying size is not necessary
  - empty brackets can be used
  - brackets must be always be used
- To assign elements to array use { }

Multi-dimensional Arrays

- Use `[]` for each dimension
- Embed `{}` for each dimension data
- Matrices are 2D arrays
- Example:

```
float m[3][4] = {  {0, 1, 2, 3},
                   {4, 5, 6, 7},
                   {8, 9,10,11}};   //3x4 matrix
```

Array Declaration Examples

```
int list[5]={1,2,3,4,5};
char string[10], message[]="Hello World";
float matrix[3][4]= {    {0, 1, 2, 3},
                         {4, 5, 6, 7},
                         {8, 9,10,11}};   //3x4 matrix
int d3[2][2][3] = { {{0,1,2},{3,4,5}  },
                    {{6,7,8},{9,10,11}}};
```

Indexing in an array

- Use index of the data element
- Index always starts from 0
- Example:
  - `var[0]`: element 0 (1st element)
  - `mat[2][3]`: row 2 (3rd row); column 3 (4th column)
  - `data[10][1][5]`: access set 10 (11th set), row 1 (2nd row), column 5 (5th col)

Programming: Single dimension example

```
int sum=0, m[10] = {1,2,3,4,5,6,7,8,9,10};

for (int i=0; i<10; i++)
   sum += m[i];
```

Programming example

```
int m[4][3][2] = {   {{0,0}, {0,1}, {0,2}},
                     {{1,0}, {1,1}, {1,2}},
                     {{2,0}, {2,1}, {2,2}},
                     {{4,0}, {4,1}, {4,2}}   };

for (int i=0; i<4; i++)
   for (int j=0; j<3; j++)
         for (int k=0; k<2; k++)
               printf("\nm[%d,%d,%d] = %d", i,j,k,m[i][j][k]);
```

Programming: single dimensional array

1. Display a message on screen but each character is on a separate line.
2. Ask the user to enter a message. Display that message on screen but flip upper case characters to lower case
3. Display the message on screen but alternate the case of each character. E.g. “HeLlO wOrLd”
4. Given an array of grades. Calculate the average the grades.
5. Find the average of a given 4x3 matrix
6. Find the average each column in a 4x3 matrix

Problem 1: Display char on new line

Ideas:

- Strings are stored in sequence
- Strings are terminated by a null (0)
- Use a for loop to go through each index of the string
- Condition to stop should be the null
- Print a character at a time
- Use formatting to print a CR before each character

Solution : Use null termination of string to stop

```
char str[] = "Hello World!"
for (int i = 0; str[i]; i++)
         printf("\n%c", str[i]);
```

Problem 2

Ideas:

- Note that the lowercase = uppercase + 32
- Scanf the entire string using option [^\n]
- Use a for loop to go through each index of the string
- Condition to stop should be the null
- Check if the character is
    - Upper case: add 32 to make it lower case then print
    - Lower case: sub 32 to make it upper case then print
    - Not a letter: print

Problem 6: Find the average of each column

Ideas:

- There are 3 columns in 4x3
- Use 2 for loop for row and column
  - Process columns then rows
- Add each element in column
  - Divide by total elements in a column (i.e 4)

Code:

```
float M[4][3]={{00,01,02},{10,11,12},{20,21,22},{30,31,32}};
float col_av[3] = {0,0,0};
for (int c = 0; c < 3; c++)
{
    for (int r=0; r < 4; r++)
        col_av[c] += M[r][c]/4;
    printf("\nAverage of column %d is %2.2f",c,average);
}
```

Functions

- A portion of a program which can be called to perform a task
- Returns to the calling program.
- Example of functions: `printf()`, `scanf()`, `pow()`, … etc.
- `main()` is a function
- Modular programming
- Allowed to declare local variables in functions

Function Definition

```
return_type function_name([parameter_list])
{
      function_body;
      [return [value_or_variable]];
}
```

- Functions types: int , float , double , character
  - void: returns nothing
  - Can use modifiers: long, signed, unsigned, etc.
- return
  - Only needed when function has return type
  - Can be value or variable

Parameter List

- Aka Input arguments
- List of variables needed for the function
- Must specify type
- Can be empty: functions don’t have to have arguments
- Variable names don’t need to match variable names outside function

Function Examples

```
int sum(int num1, int num2)        //returns the sum of num1,num2
{
          return num1+num2;
}

int max(int num1, int num2)         //returns the max of num1,num2
{

   if (num1 > num2)
      return num1;                  //num1 is greater
   else
      return num2;                  //num2 is greater or equal to num1

}
```

Calling Functions

- Every C code must have `main()` function
- Functions must be defined outside `main()`
- Calling: using a user defined C function
  - Call using function name
  - Use parenthesis
  - Pass arguments in the parenthesis
- Passing arguments can be by value or variable
  - Variable names don’t need to match

Function Call Example

```
int max(int num1, int num2)
{
         if (num1>=num2)
             return num1;
         else
             return num2;
}

void main ()
{  int i= 10, j = 100;
   printf("max of %d,%d is %d", 5,4,max(5,4));
   printf("max of %d,%d is %d", i,j,max(i,j));
}
```

Function Prototype

- Only needed if the function is called before it is defined!
  - Good form to have a prototype to avoid such problems
  - Typically it’s a problem when functions call other functions
- Functions prototype is needed to avoid error due to lack of declaration
  - Place prototype at the beginning of code
- Prototype declaration:

  return_type function_name([parameter_list_**types**_only]);

  - Don’t include variable names in the parameter list only types
  - Remember to use semicolon

Example

```
double adj (double hyp, int deg)  //calculates adjacent of triangle
{
   return hyp*cosd(deg);
}

double cosd(int deg)          //calculates cosine in degrees
{
   return cos(deg*3.14159265358979323846/180);
}

void main()
{   printf("%lf",adj(100,10));
}
```

Local Variables

- Variables declared in a {} block are local
  - {} blocks used in conditionals, functions
  - Blocks within blocks can exist: e.g. nested ifs
- Local variables exist only within {} block or sub-blocks
  - Can’t be referenced outside
- Example:

```
for (int i=1, j=0; i<5; i++)
{        j += i;
}
printf("%d,%d",i,j);
```

Global Variables

- Variables declared outside `main()`
  - Not inside any function
- They can be used anywhere in the code.

Example

```
int i = 5, j = 10;

double ipowerj()
{
   double m=1.0;

   for (int k = 0; k<j; k++)
         m *=i;
   return m;
}

void main()
{
    printf("%d^%d = %lf",i,j,ipowerj());
}
```

Example

- Write a function called even that will return 0 or 1 if the signed integer number that is pass to it is odd(0), or even(1).
- Write a function called factor that will return all the integer denominators of an unsigned integer that is passed to it. The function returns nothing.
- Write a function called calc that will perform a two number calculation. You must pass to calc: two floating point variables, and a character for the operation (+,-,*,/). The function returns the result of the calculation.

Recursive Functions

- Functions can call other functions
- Recursive functions call themselves
  - Smart way of writing code.

Example:

```
int factorial(int n)
{
    if (n != 1)
         return n*factorial(n-1);
    else
         return n;
}
```

Pointers

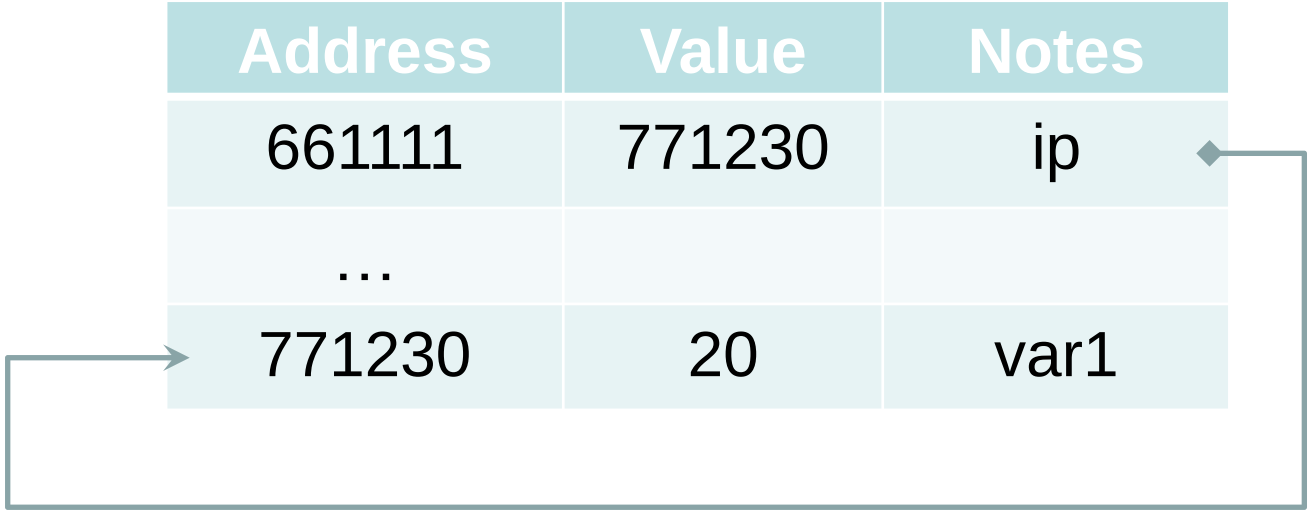
- A pointer is a variable
- whose value is the address.
- Typically used to point to values belonging to variable
- Especially useful for arrays
- Especially useful as function parameters
- Must have a type
  - Type must match variable it points to

Pointer Format

- Use * to specify that the variable is a point
- Use & to access address
- Example:

```
int  var = 20;
int  *ip;           //pointer declaration

ip = &var;

printf("Address of var variable: %x\n", &var  );
printf("Address stored in ip variable: %x\n", ip );
printf("Value of *ip variable: %d\n", *ip );
```

Pointer Diagram



Address	Value	Notes
661111	771230	ip
…		
771230	20	var1

Pointer Example

```
int *ptr, a=0, b=1;
printf( "a=%d\tb=%d\n", a, b );
printf( "&a=%d\t&b=%d\n", &a, &b );

ptr = &a;
*ptr = 3;
printf( "ptr=%d\t*ptr=%d\n", ptr, *ptr );
printf( "a=%d\tb=%d\n", a, b );

b = *ptr;
printf( "ptr=%d\t*ptr=%d\n", ptr, *ptr );
printf( "a=%d\tb=%d\n", a, b );
```

Pointer Arithmetic

- Increment or decrement the pointer address
- Increment or decrement the value that pointer points to
- Examples of address arithmetic:

```
*(p-2) = 5;
ptr +=2;
```

- Examples of value arithmetic:

```
k = *p + 2
m = k/ *ptr
```

Pointers and Arrays

- Arrays are actually defined as addresses
- To assign address of array to pointer
  - Method 1: use index 0, i.e. `[0]`
  - Method 2: assign an address of an array to a pointer **drop** &
- Pointer arithmetic can be used to access elements of an array:

```
int array[] = {1,2,3,4,5}, *p;
p = &array[0];    //p -> first element in array
*(p+1) = 5;       //next element in array: array[1]=5
```

- Pointers can be used to process arrays:

```
int array[] = {1,2,3,4,5}, *p;
for (p=&array[0], int i=0; i < 5; i++)
{           *(p+i) *=2;
            printf("%d  ",*(p+i));}
```

Pointer Arithmetic Example

```
int a[] = {3,7,2}, *p;

p = &a;

printf(“ p=%d\n*p=%d\n*p+1=%d\n*(p+1)=%d\n*p+2=%d\n*(p+2)=%d",
         p,    *p,    *p+1,    *(p+1),    *p+2,    *(p+2) );
```

Pointers and Multidimensional Arrays Example

```
int a[2][3][4] = {{{1,2,3,4},    {5,6,7,8},    {9,10,11,12}},
                  {{20,30,40,50},{60,70,80,90},{100,110,120,130}}};
int *p;

p = &a[0][0][0];

for (int i=0;i<2*3*4;i++)
     printf("%d  ",*(p+i));
```

Pointers as Function Parameters

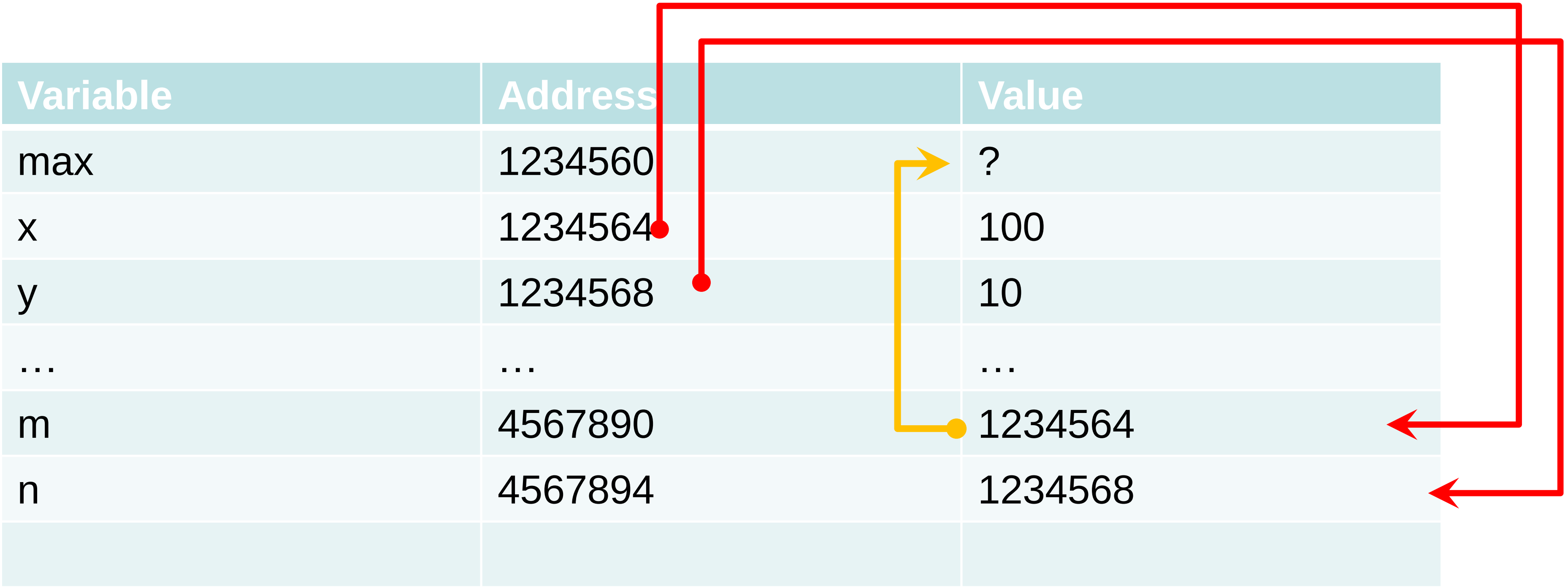
- Use * for parameters
- Pass parameters as addresses using &
- Exampple:

```
void swap (int *a, int *b)
{  int temp;
   temp = *a;
   *a = *b;
   *b = temp;}

void main()
{  int x=10, y=100;
   swap (&x, &y); }
```

Functions Returning Pointers

- Use * when defining the return
- Remember that what you're returning has to be an address

```
int *getMax(int *m, int *n) {
{  if (*m > *n)
          return m;
   else
          return n;
}

int *max, x=100, y = 10;
max = getMax(&x, &y);
```



Variable	Address	Value
max	1234560	?
x	1234564	100
y	1234568	10
…	…	…
m	4567890	1234564
n	4567894	1234568

Pointer Coding Problems

- Write a function to display the elements of an array separated by space
- Write a function that will apply cumulative sum to an array
- Write a function that will sequentially sort an array of integers
- Write a function that will bubble sort an array of integers

Display Array

- Solution 1: Use pointer arithmetic

```
void printa(int *in, int len)
{    for (int i=0; i<len; i++)
                    printf("%d  ", *(in+i));
}
```

- Solution 2: use array style

```
void printa(int *in, int len)
{    for (int i=0; i<len; i++)
                    printf("%d  ", in[i]);
}
```

Cumulative Sum

- Create an array whose elements represents the sum of all elements up to that index
  - Example: array = {1,2,3,4,5} => csum = {1, 3, 5, 9, 14}
- Algorithm:

```
1. Initialize csum[0] = array [0]
2. For i = 1 to length – 1
3.         csum[i] = array [i] + csum[i-1]
```

Sequential Sort

- Sort in ascending or descending order, sequentially.
  - Ascending order: lowest to highest
  - Compare each element to those in front of it and swap when needed
- Ascending Sort Algorithm:

```
For i = 0 to length – 2
    For j = i+1 to length – 1
        if array[i] > array[j] then swap
```

Examples: Ascending Sequential Sort

- array = {1,9,0,2,3}:

{1,	9,	0,	2,	3}						
{1,	9,	0,	2,	3}	swap:	{**0**,	9,	**1**,	2,	3}
{0,	9,	1,	2,	3}						
{0,	9,	1,	2,	3}						
{0,	9,	1,	2,	3}	swap:	{0,	**1**,	**9**,	2,	3}
{0,	1,	9,	2,	3}						
{0,	1,	9,	2,	3}						
{0,	1,	9,	2,	3}	swap:	{0,	1,	**2**,	**9**,	3}
{0,	1,	2,	9,	3}						
{0,	1,	2,	9,	3}	swap:	{0,	1,	2,	**3**,	**9**}

Bubble Sort

- Faster and simpler sorting than sequential sort
  - Compare 2 consecutive elements and sort
  - To stop: repeat (array length-1) times
- Ascending Order Algorithm 1:

```
For i = 0 to length – 2
    For j = 0 to length – 1
        if array[j] > array[j+1] then swap
```

Example 1: Ascending Bubble Sort

- array = {1,9,0,2,3}:

{1,	9,	0,	2,	3}						
{1,	9,	0,	2,	3}	swap:	{1,	**0,**	**9,**	2,	3}
{1,	0,	9,	2,	3}	swap:	{1,	0,	**2,**	**9,**	3}
{1,	0,	2,	9,	3}	swap:	{1,	0,	2,	**3,**	**9**}
{1,	0,	2,	3,	9}	swap:	**{0,**	**1,**	2,	3,	9}
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						
{0,	1,	2,	3,	9}						

Wastes 11 cycles

Bubble Sort Speedup

- Modify the algorithm so that it stops early
  - Use a flag to indicate that no swap has been made
- Ascending Order Algorithm 2:

```
1.  Initialize flag = 0
2.  For i = 0 to length – 1
3.          if array[j] > array[j+1]
                then swap, set flag = 1
4.  If flag = 1 then repeat 1
```

Example 2: Ascending Bubble Sort

- array = {1,9,0,2,3}:

flag=0	{1,	9,	0,	2,	3}						
	{1,	9,	0,	2,	3}	swap:	{1,	**0,**	**9,**	2,	3}
flag=1	{1,	0,	9,	2,	3}	swap:	{1,	0,	**2,**	**9,**	3}
	{1,	0,	2,	9,	3}	swap:	{1,	0,	2,	**3,**	**9**}
Repeat											
flag=0	{1,	0,	2,	3,	9}	swap:	**{0,**	**1,**	2,	3,	9}
flag=1	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						
Repeat											
flag=0	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						
Stop											

Wastes 7 cycles. 12 cycles compared to 16: 16/12 = 1.33x faster

Bubble Sort Double Speedup

- Note that bubble sort will bubble through the max(min)
  - So no need to check the right end
  - Each iteration decreases the array length
- Ascending Sort Algorithm 3:

```
1. Initialize i =0, flag = 0
2.  For j = 0 to (length – 1 – i )
3.          if array[j] > array[j+1]
                then swap, set flag = 1
4. Increment i
5. If flag = 1 then repeat 1
```

Example 3: Ascending Bubble Sort

- array = {1,9,0,2,3}:

i = 0, flag=0

	{1,	9,	0,	2,	3}						
	{1,	9,	0,	2,	3}	swap:	{1,	**0,**	**9,**	2,	3}
flag=1	{1,	0,	9,	2,	3}	swap:	{1,	0,	**2,**	**9,**	3}
	{1,	0,	2,	9,	3}	swap:	{1,	0,	2,	**3,**	**9**}

Repeat i=1, flag = 0

flag=0	{1,	0,	2,	3,	9}	swap:	**{0,**	**1,**	2,	3,	9}
flag=1	{0,	1,	2,	3,	9}						
	{0,	1,	2,	3,	9}						

Repeat i = 2, flag = 0

flag=0	{0,	1,	2,	3,	9}
	{0,	1,	2,	3,	9}

Stop

Wastes 4 cycles. 16/9 = 1.78x faster

Function Returning Pointers: Arrays

- To return arrays use STATIC when defining array variables.
- Problem: array size can’t be variable length

```
int *csum(int *a, int len)
{   static int r[10];
    r[0] = a[0];

    for (int i=1; i<len; i++)
        r[i] = r[i-1]+a[i];

    return &r[0];            }

int *asum, a[] = {1,2,3,4,5,6,7,8,9,0};
asum = csum(&a[0],10);
```

Alternative to returning pointers

- Pass input array and return array as parameters
  - Function has no return type

```
void csum(int *in, int *out, int len)
{   out[0] = in[0];
    for (int i=1; i<len; i++)
        out[i] = out[i-1]+in[i];
}

int sum[10], array[] = {1,2,3,4,5,6,7,8,9,0};
csum(&array[0],&sum[0], 10);
```