Basic Programming Constructs In C

Quick Review of Variables and Constants

Variables in a Computer Program

- A variable is an identifier that indicates a computer memory space, with an specific data type, such as: integer (whole number), double (real number), char (character), string (a string of characters), etc.

```
int age;
double salary;
char gender;
```

- A variable can be initialized at the time of declaration.

  Example:

```
int age = 23;
char gender =  'F' ;
double salary = 3000.00;
```

- The value of a variable can be modified (re**assigned**), if necessary.

```
age = 24;
salary = 4000.00;
```

- You may declare several variables on the same line:

```
int diameter, area;
int day = 5, month = 1, year = 2009;
```

Example

```
#include <stdio.h>
int a = 90;          // global declaration of a

int main (void)
{
    int a = 34;  // local declaration of a
    int b;
    b = a + 2;
    int c = 68;
    for(int i =0; i < 5; i++)
    {
            int d = 50;
            // MORE CODE
             …
    }
    return 0;
 }
```

Identifiers Naming Rules

- An identifier can start either by a letter (a to z, or A to Z), or an underscore (_).
- An identifier must be either one word or two or more words connected to each other by an underscore

```
int myint;
char mychar;
double 34xyz;  // Error started with digits
float my_float;
int  $y, y$;
```

- An identifier **cannot** have any of the following characters:

/ , ; . & * + - # = “ ‘ @

What is a constant?

- An identifier that indicates a computer memory space, with an specific data type (such as int, double, char, etc.) that:
  - MUST be initialized at the time of declaration , and its value CANNOT be changed.

```
const int mycnost = 34;
const float x = 4.76;
```

- To define a constant you can also use #define preprocessor directive.

```
#define pi 3.14
```

  - We will discuss this one later in more detail

C Data Types

Some of the Data types in C

- They are similar to data types you learned in the previous courses with some exceptions:
  - Doesn't support **`class`** type and objects.
    - C is NOT and object-oriented language
  - doesn't support **`byte`** data type like Processing language.
  - Size of some of the data types might be different.
  - As of C99 – supports Boolean type: bool
  - Some of the major and initial data types that we will use in this course includes:
    - int
    - float
    - double
    - char
  - **We will revisit this topic later in more details**

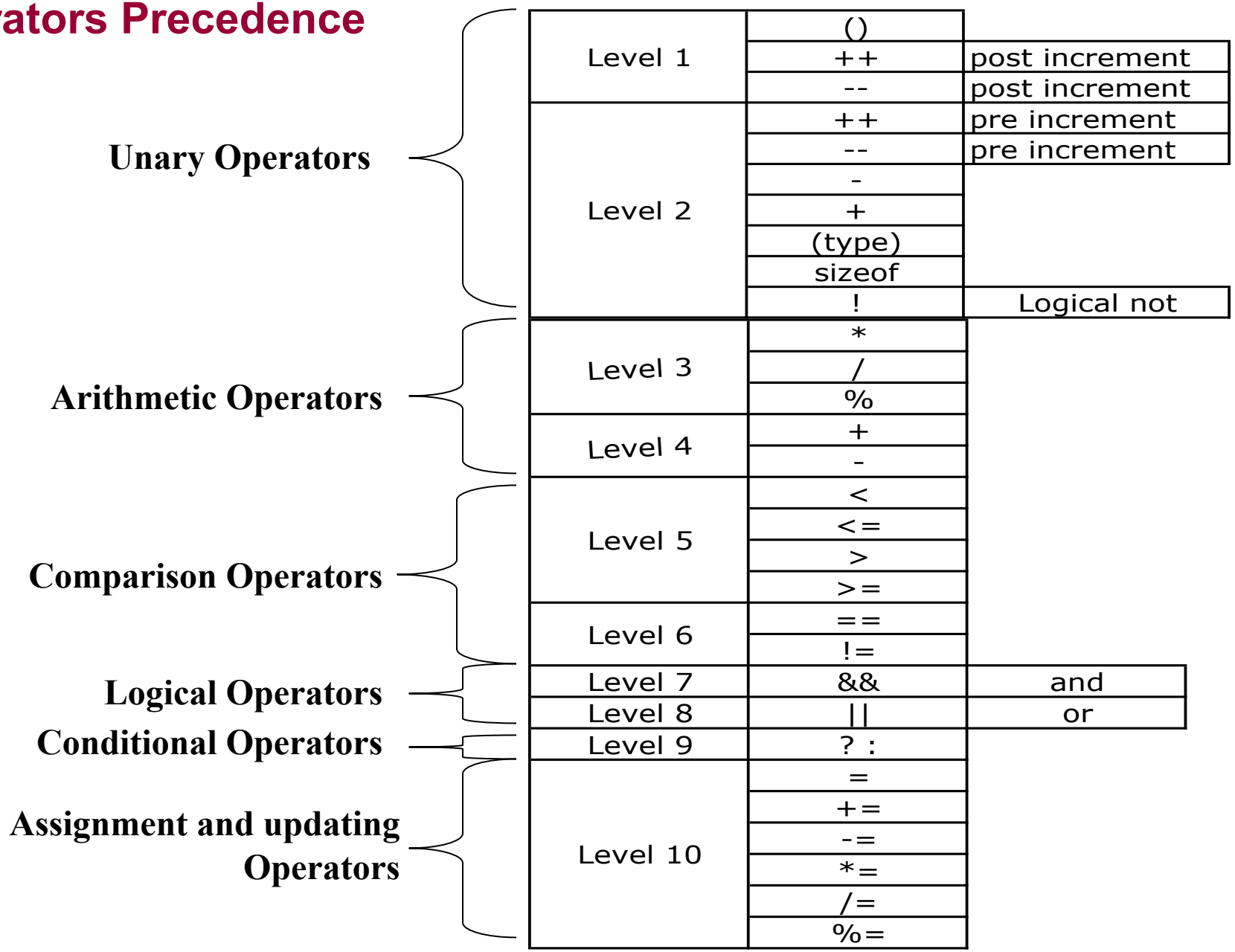
C Operators

Operators

- C is language with many operator.
- A subset of C operators can be categorized as:
  - Arithmetic Operators
  - Increment and Decrement Operators
  - Relational and Logical Operators
- Most the rules and syntax that you have learned in previous languages C++ or Processing are applicable to C.

  Examples:
  - The same rules for operators precedence
  - The same rules for integer and real division

Operators Precedence



Type Cast Operator

- The concept of type casting in languages like C, C++, Processing, Java, is the same. However the syntax might be slightly different.
- For example, C++ can use the following format to cast a double data type to an int data type:

```
double x = 2.987;
cout << int(x);   // displays 2
```

- C and Processing use similar format as shown in the following examples:
  - C code:

```
double x = 2.987;
printf("%d", (int)x);   // displays 2
```

  - Processing code:

```
double x = 2.987;
print((int)x);   // displays 2
```

Brief Introduction to Standard Input Output in C

Standard I/O Streams in C

- C uses library functions printf and scanf. You needs to Include header file <stdio.h>

```
#include<stdio.h>
int main()
{
  int age;
  printf( "Please enter your age: \n");
  scanf("%d", &age);  // reads the user input into age
  …
  printf("You will be %d years old next year.", age + 1);
  return 0;
}
```

Notes:
"%d" is a format-specifier. In this example, it specifies that input for age on the keyboard must be an integer.

Some of the most commonly used format-specifier include:

- %d : for scanning or printing an integer.

```
int age;
printf( “Please enter your age: “);
scanf(“%d”, &age);
printf(“you are %d  years old.\n”, age);
```

You may also use “%i” for integers

- %f : for scanning or printing a floating-point number.

```
float salary;
printf( “Please enter your salary: “);
scanf(“%f”, &salary);
printf(“your salary is %f.”, salary);
```

Some of the most commonly used format-specifiers include:

- %c : for scanning or printing a character (char).

```
char gender;
printf( "Please enter your gender (M/F): ");
scanf("%c", &gender);
```

- %lf : for scanning or printing a double

```
double salary;
printf( "Please enter your salary: ");
scanf("%lf", &salary);
/*allowed to use "%f" for printing */
printf("\nyour salary is %f", salary);
```

- Scanning more than one data:

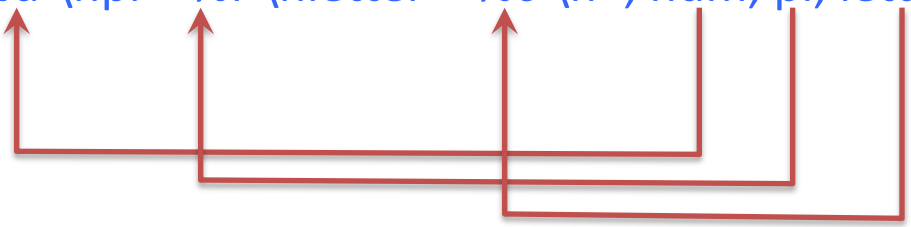
```
int age;
float salary;
printf("\nPlease enter your age and your salary: ");
scanf("%d%f", &age, &salary);
```

Printing Variable Values in C

- The format specifier is used to indicate that something is to be inserted at that position with a specific format.
- The first variable listed after the closing quote is inserted into the first location, the second one into the second, and so on.

Example:

```
#include <stdio.h>
void main( void )
{
    int num = 42;
    float pi = 3.14;
    char letter = 'a';
    printf("num  = %d \npi = %f \nletter = %c \n", num, pi, letter );
}
```



Control Structures in C

C Control Structures

- A Quick review of control structures in C and C++.

if (logical-expression) {
  some statements…
}
	if (x < 0) {
	  printf (" x is negative.");
	}

while (logical-expression) {
  some statements…
}
	int i = 0;
	while (i < 5) {
	  printf("%d\n", i);
	  i++;
	}

for (initialization-expr ; logical-exp; updating-exp) {
  some statements…
}
	int i = 0;
	for (i = 0; i < 5; i++) {
	  printf("%d\n", i);
	}

do {
  some statements…
} while (logical-expression) ;
	int = 0;
	do {
	  printf("%d\n", i);
	  i++;
	} while (i < 5) ;

What is True and What is False in C?

- Any number other than 0 is true, and zero is false
  - Example. What is the value of x after the following code?

```
int x = -3;

if( x )
        x++;
else
        x--;
```

  - Answer: -2
- Another example:

```
while(1)
{
   // do_something
}
```

- What is the value of y in the following statement?

```
y = 9 >= 21;
```

  - Answer: 0

Questions to answer:

Suppose x and y are variables of type double.

What is the output of the following code segment if :

- x is -1.25 ? Answer: y is -1.250000.
- x is -3.0 ? Answer: y is -2.500000.
- x is 2.25 ? Answer: y is 2.250000.
- x is 2.75 ? Answer: y is 2.500000.

```
if (x < -2.5)
      y = -2.5;
else if (x > 2.5)
      y = 2.5;
else
      y = x;
printf("y is %f.\n", y);
```

Review of while loop

- What is the output?

```
int i = 4;
while (i >= 0)
{
   i--;
   printf("%d ", i);
   i--;
   printf("%d ", i);
   i--;
   printf("%d\n", i);
}
```

Answer:

3 2 1

0 -1 -2

Review of do loop

- What is the output?

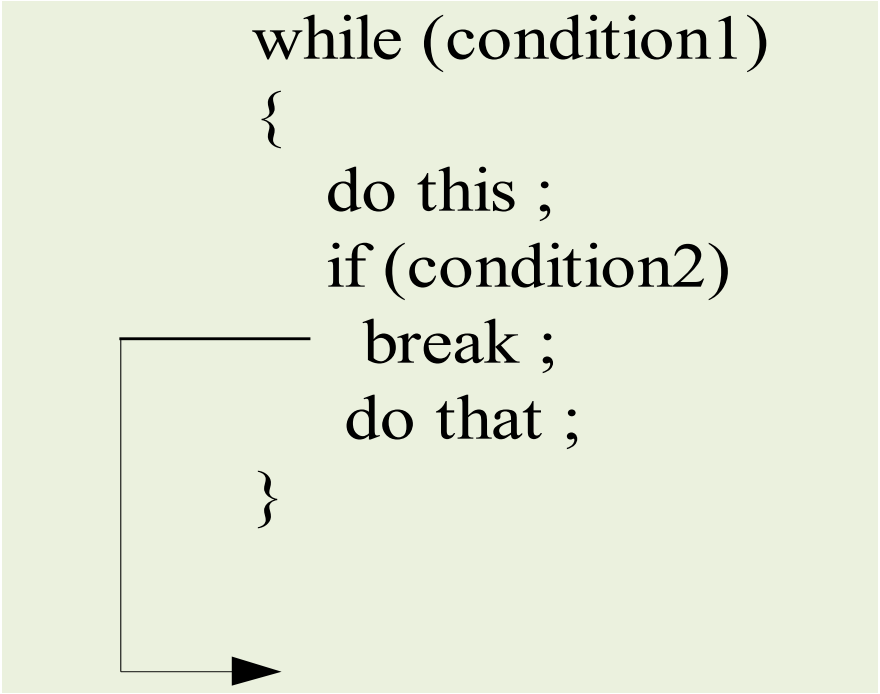
```
int j = 10, k;
do {
   j -= 4;
   printf("j is %d.\n", j);
} while (j > 0);

k = j;
do {
   k -= 3;
   printf("k is %d.\n", k);
} while (k > 0);
```

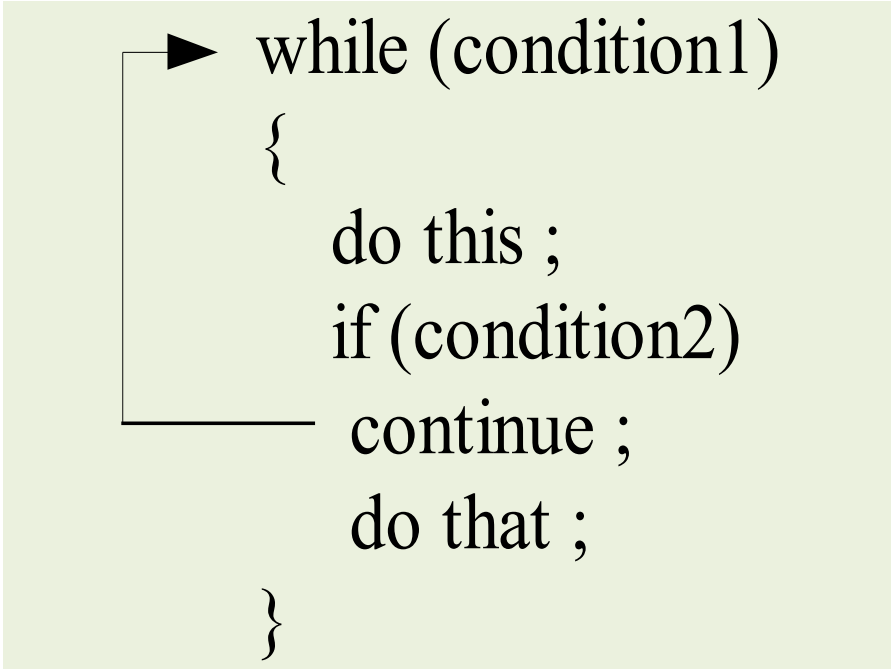
Answer:

j is 6.
j is 2.
j is -2.
k is -5.

Jump Structures

- C also supports several jump structures:
  - break
  - continue
  - goto (not recommended and we never use it in ENSF 337)





Nested Control Structures

- Like C++ and Processing you can use several nested levels of control structures in a C program:
  - Nested if, else, or both:

```
if ( logical_expression) {
        …
        if(another_logical_expression) {
              …
              if(more_logical_expression) {
                    …
                    …
              }    // closing brace for the most inner if
        }          // closing brace for middle if
}                  // closing brace for first if
```

  - Similarly, you can have nested loops in your C program

Review of for loop

- What will the output be?

```
int i, j;

for (i = 4; i > 0; i--) { /* outer loop */
   for (j = 2; j < 10; j += 2) /* inner loop */
      printf("%d ", 100 * i + j);
   printf("end\n");
}
```

Answer:

402 404 406 408 end

302 304 306 308 end

202 204 206 208 end

102 104 106 108 end

Functions in C

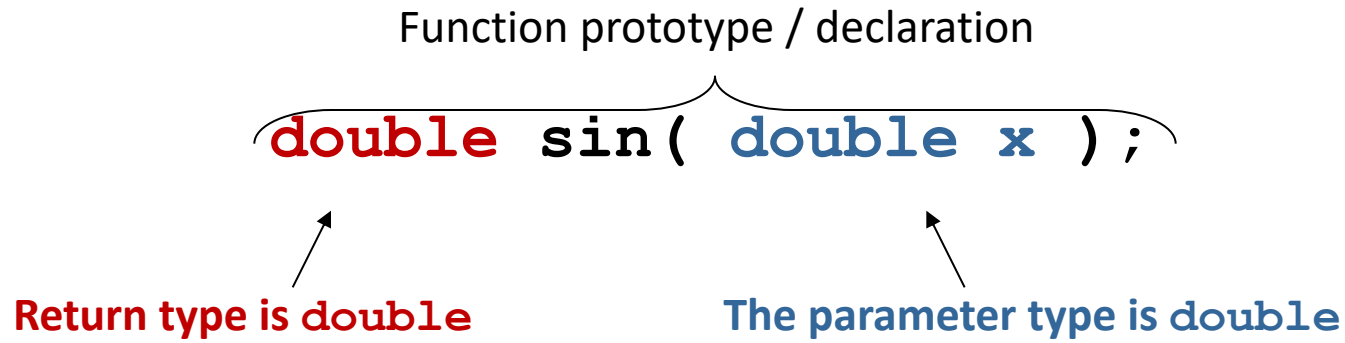
What is a Function?

- Function is a small unit of programming that can be used and called like a black box.
- You can call them whenever and wherever you need them.
- The format and rules to define functions in C/C++ and Processing is almost identical.
- In General each of these languages provides many predefined function stored and readily available for programmers.
- All you need to know is the outside view of the function to call.
  - precondition (what the function requires)
  - post-condition (what the function promises)

Some of the Predefined Functions in C

Predefined Functions

- In C the predefined functions may take one or more *argument* of a specific type, and they *return a value* of a specific *type*.
- Consider the following function *prototype/declaration* for computing sine of angles in degrees:



- A function prototype gives the function's ***signature***
  - The return type (if any)
  - The function name
  - The number of parameters
  - The parameter types (may be different)

parameter type = double

Predefined Functions

- The function call **sin(x)** is a C *expression*, hence it has a *type* and a *value*
- Function arguments may be an expression
  - Constant
  - Variable
  - Math expression

Math Expression	Function Call	Function Prototype
|x| **|2*3|** **|x + y/z|**	**fabs(x)** **fabs(2*3)** **fabs(x + y/z)**	**double fabs(double);**
x^3	**pow(x, 3)**	**double pow(double, double);**
e^{x+2}	**exp(x + 2)**	**double exp(double);**

User Defined Functions

User Defined Functions

- Why should we bother to write functions?
  - Hide the implementation detail of complicated or tricky operations (procedural abstraction)
  - Produce reusable code segments
  - Produce more robust code
  - Eases merging of code from multi-programmer teams

- Implementation details can be ignored once functions are written and tested
  - Don't need to worry about the algorithm itself, or how it has been implemented
  - This speeds up the programming process and helps limit errors

User Defined Functions

- Problem: write a function to square a number (a common operation)
- The function prototype gives the interface details and has the same syntax as a predefined function

```
double square( double x );
```

- The function ***definition*** gives the implementation details:

```
double square( double x )
{
    return x * x;
}
```

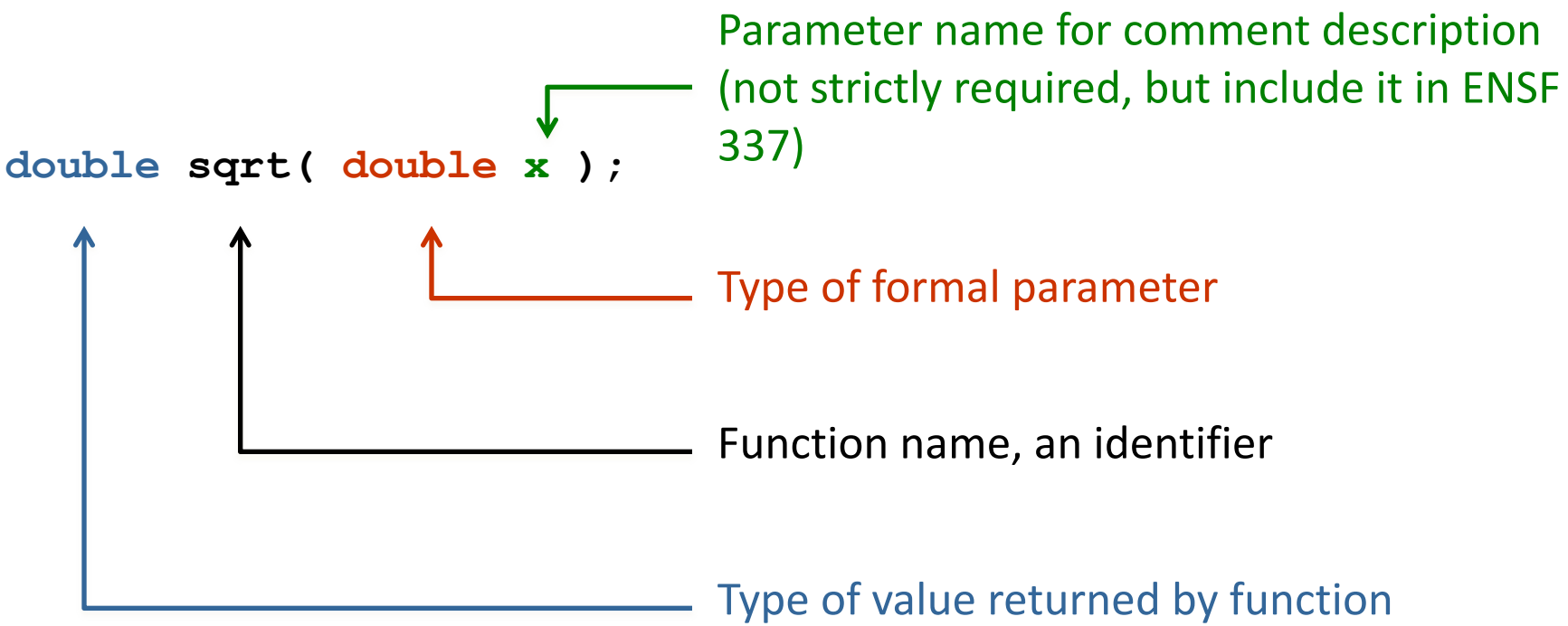
formal parameter (required)

- The signature of the prototype and definition must match

Function Declaration (Function Prototype)

- Lets take a closer look at the function declaration/prototype
- The syntax is as follow

```
<return_type> <function_name>( <parameter_list> );
```



Function Call

- Here's how to use a user-defined function

```
#include <stdio.h>


double square(double x);

int main()
{
   double y = 6.2;
   double y_squared = square(y);

   printf("The square of the number is %f.\n", y);

   return 0;
}


double square(double x)
{
   return x * x;
}
```

function prototype

function call

argument

function definition/implementation

User Defined Functions

- Difference between *arguments* and *formal function parameters*
  - An argument is used in the function call (sometimes called a parameter)
  - A formal parameter is effectively a variable within the function

a is the argument to the function call

```
int main()
{
 double a = 4.0;
 double b = square(a);
 // MORE CODE
 return 0;
}
```

The formal parameter **x** in function square is the copy of **a in the main.**

```
double square(double x)
{
  double result = x * x;
  return result;
}
```

- This is called ***pass-by-value*** because the *value* of the argument is passed (assigned) to the formal parameter

User Defined Functions

- Some notes regarding pass-by-value

  – The arguments (in main, on the previous slides) and the formal parameter do not need to have the same name!
    - They are different variables and therefore have different memory addresses and different ***scope*** (more on scope later in the chapter)
    - Think of the formal parameter as a new variable available for use only within the function definition and that is it initialized with the value of the argument

  – Because the formal parameter and the argument have separate memory addresses, changing the value of the formal parameter ***does not*** affect the value of the argument!

User Defined Functions

Let's look what's going on…

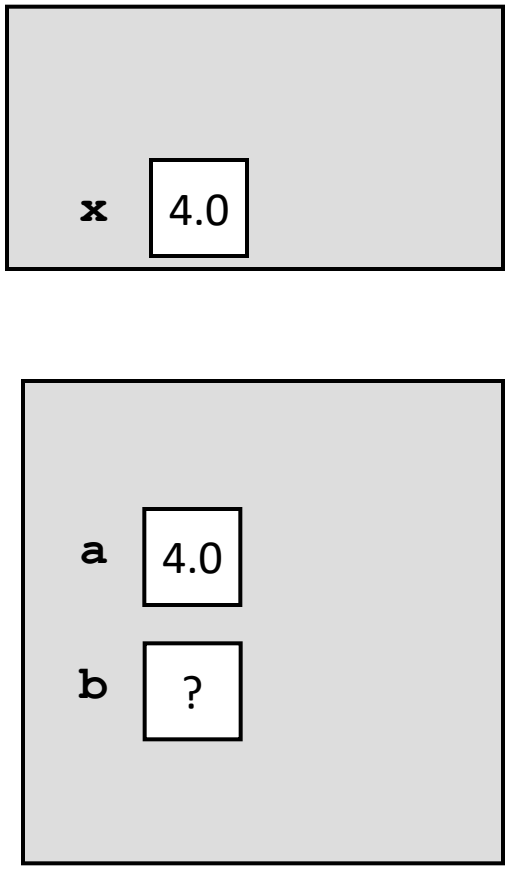
…in code

```
int main()
{
    double a = 4.0;
    double b = square(a);
    // MORE CODE
}
```

value of a

```
double square( double x )
{
  // Point 1
    return x * x ;
}
```

…on the memory



User Defined Functions

- Example of function with multiple parameters

  – Function prototype

```
double f( double a, double b, double c );
// returns polynomial value
// a^2 - 2b + c
```

  – Function definition

```
double f( double a1, double a2, double a3 )
{
  return  a1 * a1 – 2 * a2 + a3;
}
```

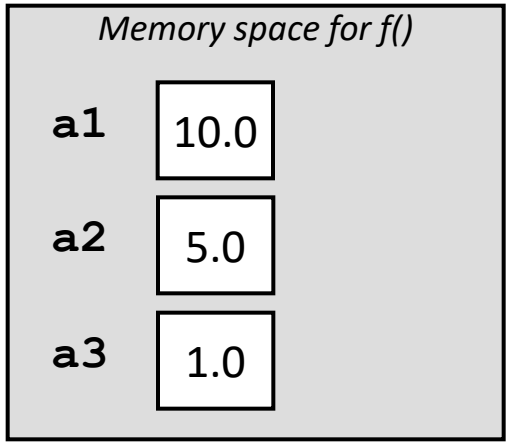
Note: Argument names in the prototype and definition do not have to match. In fact, arguments in the prototype do not even require a name!

User Defined Functions

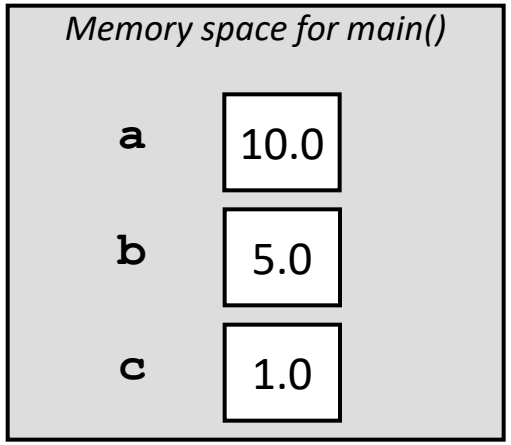
```
double f(double a1, double a2, double a3);

void main()
{
   double a = 10, b = 5, c = 1;
   pintf("%f", f(a, b, c));
   return;
}

double f(double a1, double a2, double a3)
{
   return a1 * a1 – 2 * a2 + a3;
}
```

Alternate function definition…what's the difference?

stack



AR f



AR main

Order of arguments **always** matches the order of formal parameters:

- Value of 1st argument is copied to 1st formal parameter
- Value of 2nd argument is copied to 2nd formal parameter
- And so on…

More on Functions

- Formal arguments can also be made constant so that they cannot be changed within the function
- Consider the following function definition

```
double CircleArea( const double pi, double radius )
{
   pi *= 3.00; // illegal
   return   pi * radius * radius;
}
```

- The above is illegal because the formal parameter ‘pi’ is declared to be constant
- Constant formal parameters will be more important when we discuss classes, arrays, strings, etc. later in the course

Functions That Do Not Return a Value

Functions That Do Not Return a Value

- *Void functions* do not return a value

  – Function prototype

```
void printDouble(double x);
// prints x to the screen
```

  – Function definition

```
void printDouble(double x)
{
  printf("The value is %f\n", x);
  return;
}
```

`return` is not strictly required for void functions but it is good programming practice

  – Function call

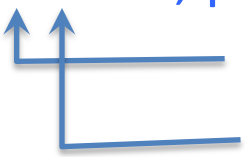
```
printDouble(14.5);
```

Closer Look at the Library Function `printf`

Formatted Output

- In C the default format to display real numbers is 6 digits after the decimal point (3.140000). To control the format of the output we can use the following syntax to indicate the field size and the number of decimal points.
- E.g.

```
#include <stdio.h>
void main( void )
{
    int x = 42;
    float pi = 3.14;
    printf("pi is %8.3f! ", pi );
    printf("x is %10d! ", x );      // prints 42 in a field of 10
}
```



8 is the field size

3 is the number of digits after decimal

Output is:

Pi is 3.140!
x is 42!

Special (non-printing characters)

- There is an entire set of special characters that we can insert.
  - \a beep
  - \b backspace
  - \f form feed
  - \n newline
  - \r carriage return (move to the beginning of the current line)
  - \t tab
  - \v vertical tab
  - \\ backslash
  - \’ single quote
  - \” double quote
  - \0 Null
- Example:

```
printf(“%c%c%c”, ‘\a’,’\a’,’\a’);
pirntf(“The \”file path\” is C:\\moussavi\\ENSF337”);
```

First line makes three beep sounds:

Second line prints:

The ”file path” is C:\moussavi\ENSF337

New Line Character

- To print character to the next line, you should use the newline-character, '\n'.
- Example: the following simple program prints:

```
First line,
Second line,
Third line.
```

```
#include <stdio.h>

int main( void )
{
    printf("First line, \n");
    printf("Second line, \n");
    printf("Third line. \n");
    return 0;
}
```

- Alternately, we could have put everything into one printf statement:

```
#include <stdio.h>
void main( void )
{
    printf("First line, \nSecond line, Third line.\n");
}
```