University of Nottingham
UK | CHINA | MALAYSIA

Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

Introduction

- Today we will cover:
- Chapter 9 – Loops
- Chapter 10 – Using functions
- Chapter 11 – Arrays
- Chapter 12 – Variables and memory allocation

Start recording!!

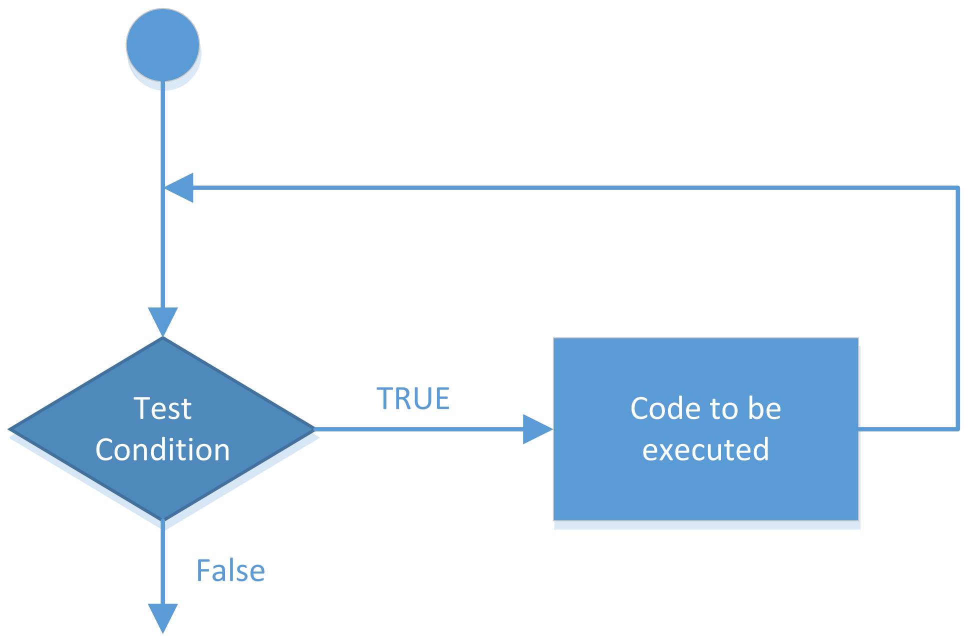
Chapter 9

Loops – Repeating Things

Looping….

Often, when coding, we need a block of code to be repeated multiple times.

In C (and indeed all programming languages) there are different ways/types of loops however they all (basically) allow the following to be implemented

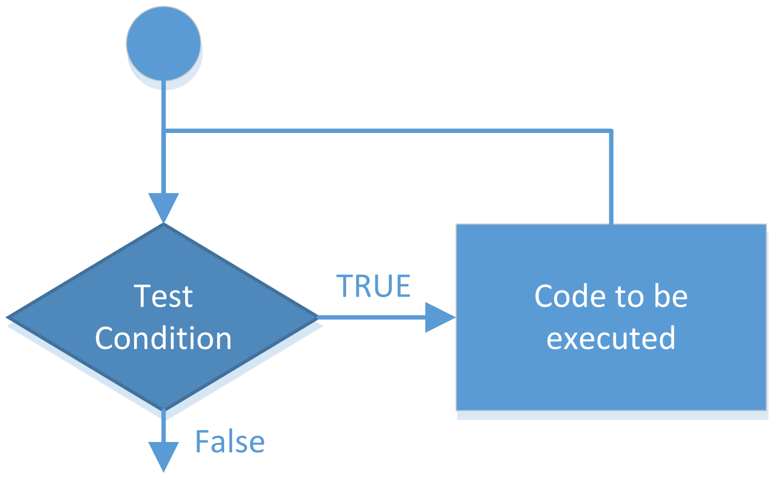


Looping

In C we have three types of loops at our disposal, which we use is dependent on what we need to achieve in our code. They are

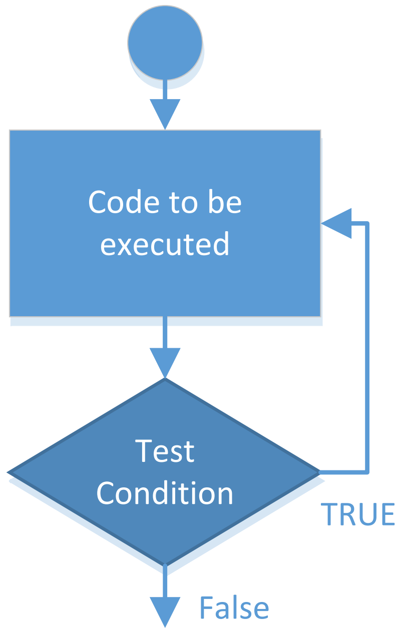
- `while` : This form of loop repeats statement(s) while the condition remains true. The test is made BEFORE executing the ‘loop code’
- `do … while` : This form of loop repeats statement(s) while the condition remains true. The test is made AFTER executing the ‘loop code’
- `for` : This style of loop repeats statement(s) a set number of times – with us being able to identify the number of times the loop has run (and make use of this value)

Pictorially: while and do...while



`while`

Condition is tested **before** the code is executed
(so it may NOT be executed)



`do … while`

Condition is tested **after** the code is executed
(so code will execute at least once)

While and do…while syntax

while

```
x = getch();
while ( x != ‘q’ )
{
      printf ( “Press a key\n”);
      x = getch();
}
```

do…while

```
do
{
      printf ( “Press a key\n”);
      x = getch()
} while ( x != ‘q’ );
```

Some examples in code

We will look at how we can use these two types of loops in code

LC9\while_example_version_1.c , LC9\while_example_version_2.c , LC9\do_while_example.c

for loops: For repeating a number of times

A for loop is a style of loop where we ‘build in’ the conditions at the point where we define the loop.

They are used when we want a loop to count over a range of numbers (e.g. from 1 to 10)

We need to provide three things:

- The starting value
- A condition statement
- How we will be changing the value each time the loop completes

for loops

Consider the case when we wish to count over the range 0 to 100 inclusive using a variable **i** to hold the count value

i.e. we want values of **i** to be 0,1,2,3,…100

The start value is easy: `i = 0`

for loops – Test Condition

The test condition is one that needs to be non-zero (true) over the range we wish to count, so either of:

```
i < 101

i <= 100
```

for loops - Incrementing

We wish to count up in 1s, so the change we make each time to i can be written as either:

```
i = i + 1
```

```
i++
```

for loops - Format

Putting this into the correct format for C we get the options below

```
for ( i = 0 ; i < 101 ; i++ )
for ( i = 0 ; i <= 100 ; i++ )
for ( i = 0 ; i < 101 ; i = i + 1 )
for ( i = 0 ; i <= 100 ; i = i + 1 )
```

Note
DO NOT put a semicolon here!

If you do this will 'be' the line of code controlled by the loop.

Note:

- A ***for*** statement acts on the next line of code so, if we wish to have multiple lines 'controlled' by the for loop we need to put the code in {}
- You can, if you wish, use {} for a single statement

for loop: Some examples in code

We will look at how we use for loops in practice

LC9\for_loops.c

Chapter 10

Function Programming (Part 1)

Functions: The building blocks of code (1)

To date, we have looked at developing all our code inside main()

For short programs (even some long ones!) this is fine however it does lead to problems

- Only one person can work on the code at any one time
- Each time we write code, the only way to reuse existing code is via copy & paste
- We cannot validate components in the system
- So - we cannot test at a functional level, just a system level (BAD!)

Functions: The building blocks of code (2)

To date, we have looked at developing all our code inside main()

For short programs (even some long ones!) this is fine however it does lead to problems

- Only one person can work on the code at any one time
- Each time we write code, the only way to reuse existing code is via copy & paste
- We cannot validate components in the system
- So - we cannot test at a functional level, just a system level (BAD!)

The solution to this is to break our code up into functions

- These are ‘blocks’ of code to which, if required, we pass values and receive (if appropriate) values back
- Each function can be separately validated against known criteria
- We can easily make use of functions in other programs knowing they perform as expected

Functions (1)

All functions consist of three parts

- A return type
  - This is any valid C variable type or void

- A Name
  - You get to pick this – it cannot replicate an existing reserved word in C, must meet naming restrictions and, ideally, should indicate what the function does

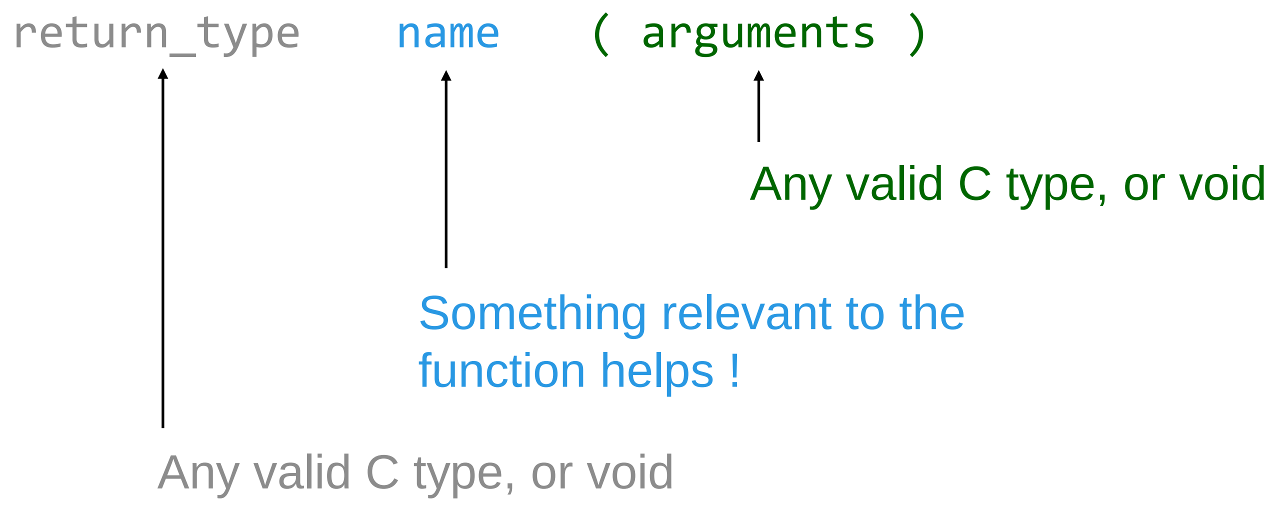
- An argument list
  - The argument list **MUST** contain at least one valid C variable type or void[+]
  - The format is: variable_type name_of_variable (comma separated, repeated as required)
  - Multiple arguments are comma separated

Plus some code!

(+) void is the ‘C’ term for denoting nothing

Functions (2)

Or, to put it another way…



```
return_type    name    ( arguments )
```

Any valid C type, or void
Something relevant to the function helps !
Any valid C type, or void

Our analysis of the problem will allow us to determine

- The return type
- Number of (and type of) inputs to the function (the arguments)

Function declarations

Some possible examples (excluding the code) might be

- float CalculateArea (float Radius)

- float CalculateCylinderVolume (float Radius, float Length)

- int IsItComplex (float a, float b, float c)

As ever, **you** need to consider the type of variables types to pass/return

- which will have come from your design stage!

Function prototypes (1)

Write a prototype for functions which:

- Converts a temperature in degrees Fahrenheit to degrees Celsius
- Inputs 3 integer numbers and returns the largest

Function prototypes (2)

Write a prototype for functions which:

- Converts a temperature in degrees Fahrenheit to degrees Celsius
- Inputs 3 integer numbers and returns the largest

```
float ConvertDegFToC( float degreesF )
```

Function prototypes (3)

Write a prototype for functions which:

- Converts a temperature in degrees Fahrenheit to degrees Celsius
- Inputs 3 integer numbers and returns the largest

```
float ConvertDegFToC( float degreesF )

int  ReturnLargestInteger( int num1, int num2, int num3)
```

Functions: prototypes

We can place functions anywhere in our code (remembering it will start at main())

If we need to use a function before we have ‘written’ it (or it is elsewhere) we need to make the compiler ‘aware’ of the function so it can check we use it correctly

To do this we put a description of the function at the top of our code (or into another file which we include)

It takes the same form as the function definition, we simply pop a semicolon on the end

The code does of course have to be written at that point!

In fact…

- You have already been making use of function prototypes, they are in stdio.h, stdlib.h which we include in all out code

functions: Some examples in code

We will take a 1st look at functions in C

A quick reminder:

- No matter where it is placed, program execution always begins with the first statement in the function `main()`

LC10\simple_functions_1.c , LC10\simple_functions_2.c

void Functions

Note: Sometimes when we write a function we do not require a value back (ie nothing returned).

- This type of function is referred to as a void function

The only difference is that the return type is void e.g.

- `void DisplaySomeInformation ( int a, int b, int c)`

And the return returns no value

- `return ;`

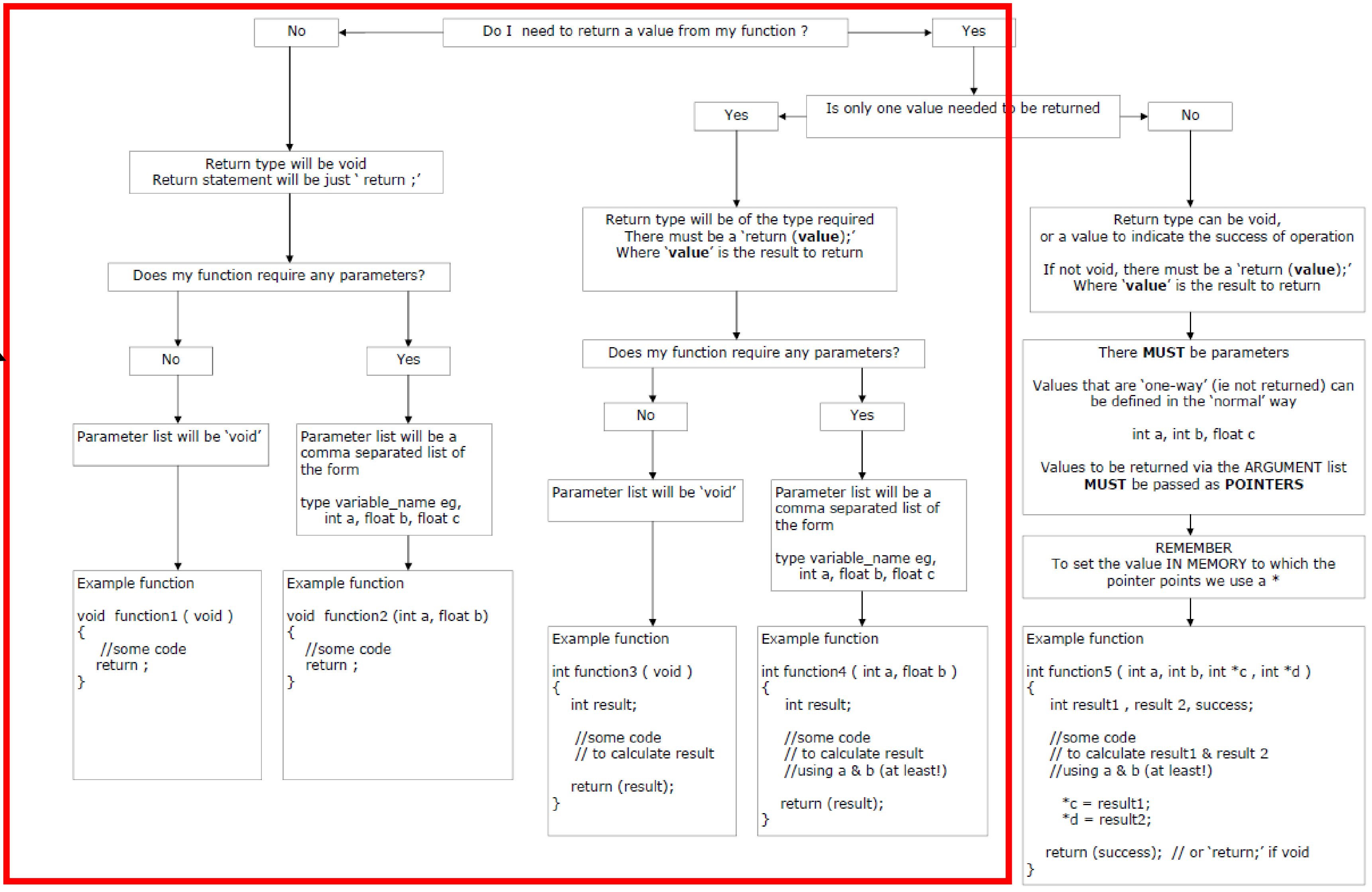
Functions: A template

The function flow chart you have been provided with in the course book will enable you to develop any function (you need to provide the actual code).

You may wish to refer to this when starting to develop your own functions.

Functions Flowchart

We have now covered this part of the flowchart



Do I need to return a value from my function ?

No

Return type will be void
Return statement will be just ' return ;'

Does my function require any parameters?

No

Parameter list will be 'void'

Example function

```
void  function1 ( void )
{
    //some code
   return ;
}
```

Yes

Parameter list will be a comma separated list of the form

type variable_name eg,
int a, float b, float c

Example function

```
void  function2 (int a, float b)
{
   //some code
   return ;
}
```

Yes

Is only one value needed to be returned

Yes

Return type will be of the type required
There must be a 'return (**value**);'
Where '**value**' is the result to return

Does my function require any parameters?

No

Parameter list will be 'void'

Example function

```
int function3 ( void )
{
   int result;

    //some code
   // to calculate result

   return (result);
}
```

Yes

Parameter list will be a comma separated list of the form

type variable_name eg,
int a, float b, float c

Example function

```
int function4 ( int a, float b )
{
    int result;

    //some code
    // to calculate result
    //using a & b (at least!)

    return (result);
}
```

No

Return type can be void,
or a value to indicate the success of operation

If not void, there must be a 'return (**value**);'
Where '**value**' is the result to return

There **MUST** be parameters

Values that are 'one-way' (ie not returned) can be defined in the 'normal' way

int a, int b, float c

Values to be returned via the ARGUMENT list **MUST** be passed as **POINTERS**

REMEMBER
To set the value IN MEMORY to which the pointer points we use a *

Example function

```
int function5 ( int a, int b, int *c , int *d )
{
   int result1 , result 2, success;

   //some code
   // to calculate result1 & result 2
   //using a & b (at least!)

      *c = result1;
      *d = result2;

   return (success);  // or 'return;' if void
}
```

Chapter 11

Arrays

Arrays (matrices to mathematicians!)

So far we have looked at creating individual variables in which we can store values – looking to pick the correct type and using sensible names

So far, so good…

- But what if we needed to store 1,000,000 values? If we tried to create each variable independently the time to do this would be wildly impractical

There is a solution!

- We create an array of variables – this allows us to define a ‘matrix’ (1D, 2D etc.) into which we can store (and retrieve) values

Arrays: in practice (1)

We define an array of variables in a similar method to any other variable

```
int c;          // Define an integer c
```

The difference is for an array is we provide ‘dimensions’

```
int x[10];      // Define x, a 1D array of integers of size 1x10
```

Things to note

- The size of the array is indicated using square brackets ‘[‘ & ‘]’
- The number in the brackets is the size of the array

A **very, very important** thing to note

- When referencing an array we start at 0 (zero), so for the above we have items
- `x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9]`

Arrays: in practice (2)

- We can have multidimensional arrays, we just use more [] terms, e.g.

```
int data[10][20];
float samples[2][10][5];
```

- The most commonly used ones are however 1D & 2D
- To set/get values we indicate the element we are interested in (much like the item in a matrix)

```
data[5][5] = 2;
Y = data[6][6];
```

- Arrays are often used with for loops (for example to set all values)

```
int data[10][20];
for ( i = 0 ; i < 10 ; i++)
    for ( j = 0 ; j < 20 ;j++ )
       data[i][j] = 0;
```

Or, if you prefer the {} approach →

```
int data[10][20];
for ( i = 0 ; i < 10 ; i++)
{
    for ( j = 0 ; j < 20 ; j++ )
   {
        data[i][j] = 0;
   }
}
```

Arrays: Some examples in code

- We will look at how we use arrays in practice

LC11\loop_into_array.c

Chapter 12

Variables: Part 2

Variables in C/C++

Whenever we create variables in C/C++ (or any language) memory is set aside to hold the values

Variables are created when functions begin

- Remember main() is a function
- When the function finishes (ie ‘return’ is executed’) the memory released.
- Provided we have done things correctly, when a program finishes all the memory we started with is freed up

The ‘technical’ term for such variables are ‘Automatic Variables’

Variables & functions

You will have noticed that when we created functions we defined variables in two places:

- The argument list – values were copied into these when the function was called
- Within the function itself – we used these to allow the function to function

You may have noticed that the same variables were used in main() and the functions()

How can this be done?

The answer to the last point is covered by what we refer to as SCOPE

Scope of Variables

- There are two types of acceptable variables
  - Those defined within a function – we call these **LOCAL** variables
  - Those defined as a function parameters – these are called **FORMAL** parameters

- There is a third type (which I hate to mention)
  - Global variables – we DO not use these[(+)], they are EVIL and the work of lazy programmers!

[(+)] Apart from exceptional cases where the is no other alternative – which is not the case in the C programming part of the module!

Memory Usage in C/C++ (1)

This is very important

- Variables are accessible ONLY in the function in which they are defined

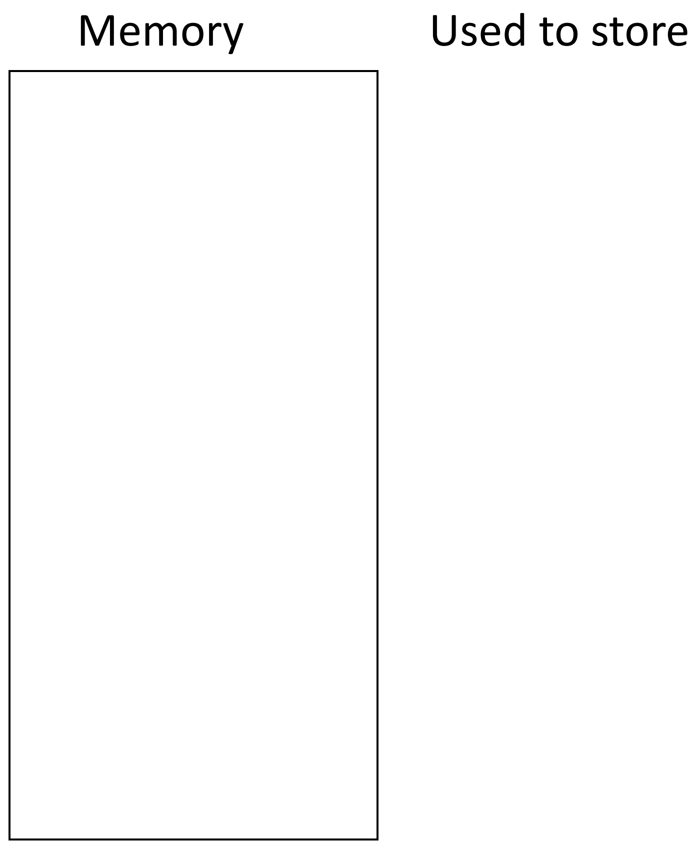
You can define ‘global variables’ but this is very (very) poor programming practice

- We are NOT going to consider using them!

Memory Usage in C/C++ (2)

So considering a simple main() programme

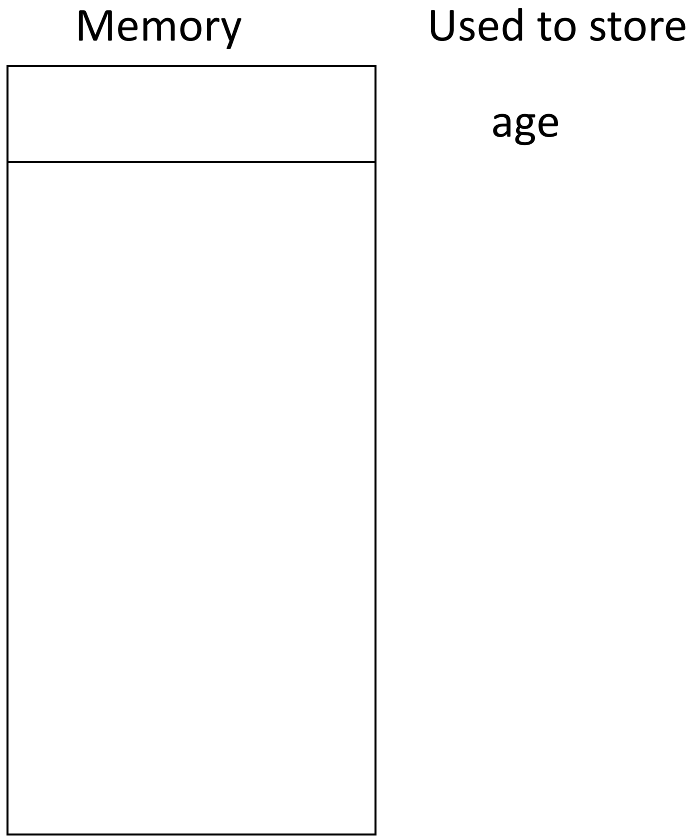
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (3)

So considering a simple main() programme

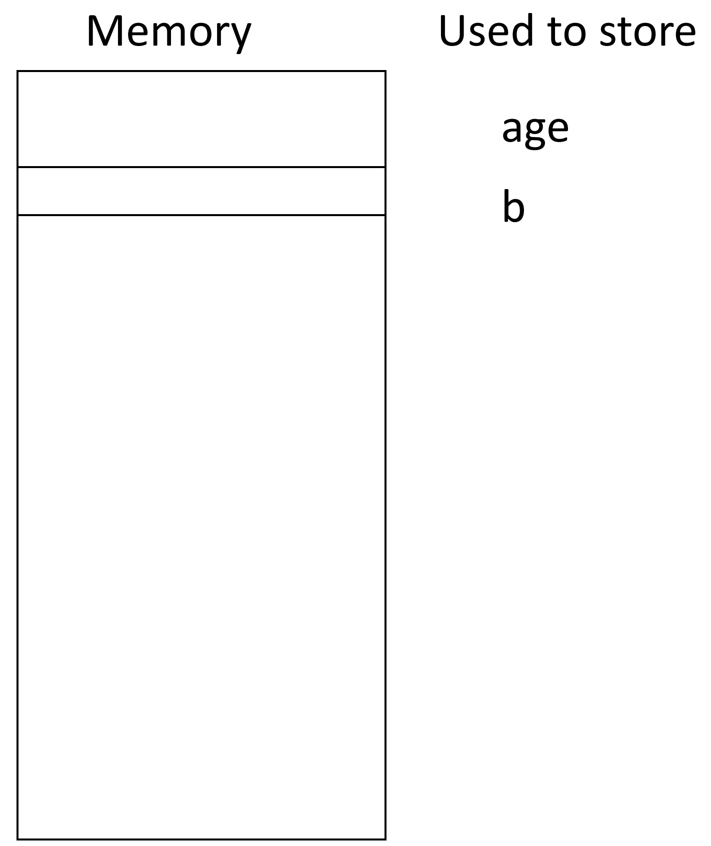
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (4)

So considering a simple main() programme

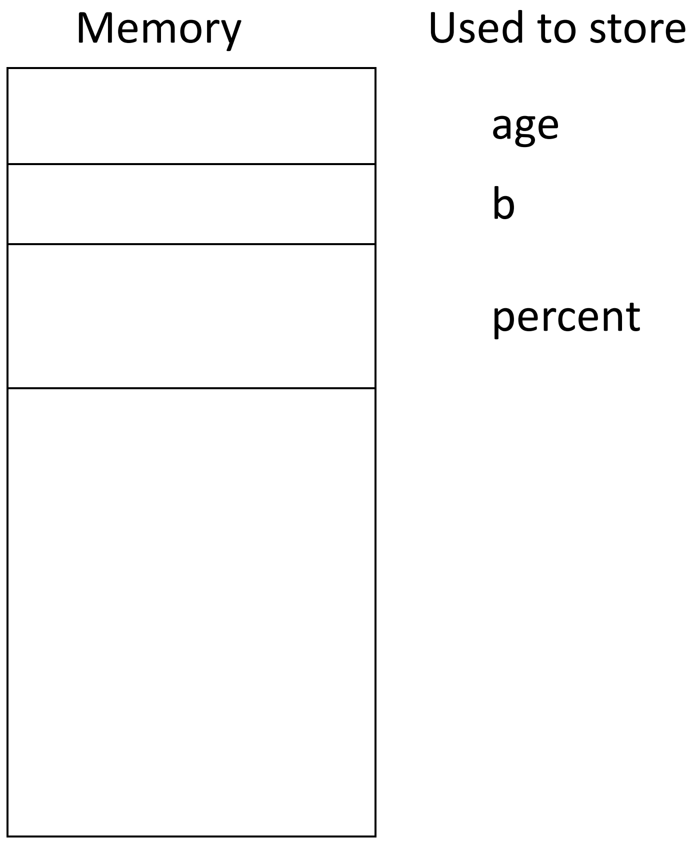
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (5)

So considering a simple main() programme

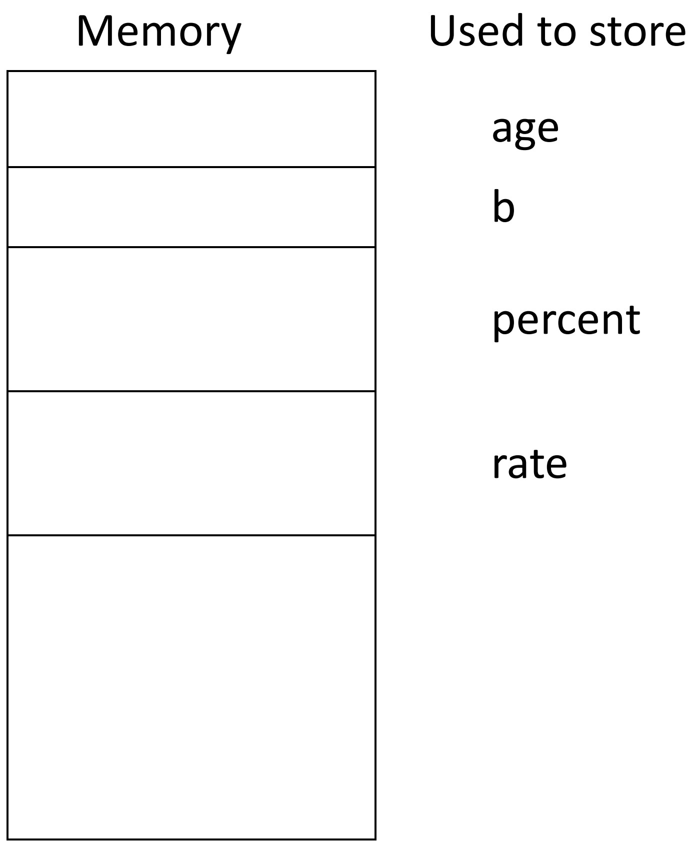
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (6)

So considering a simple main() programme

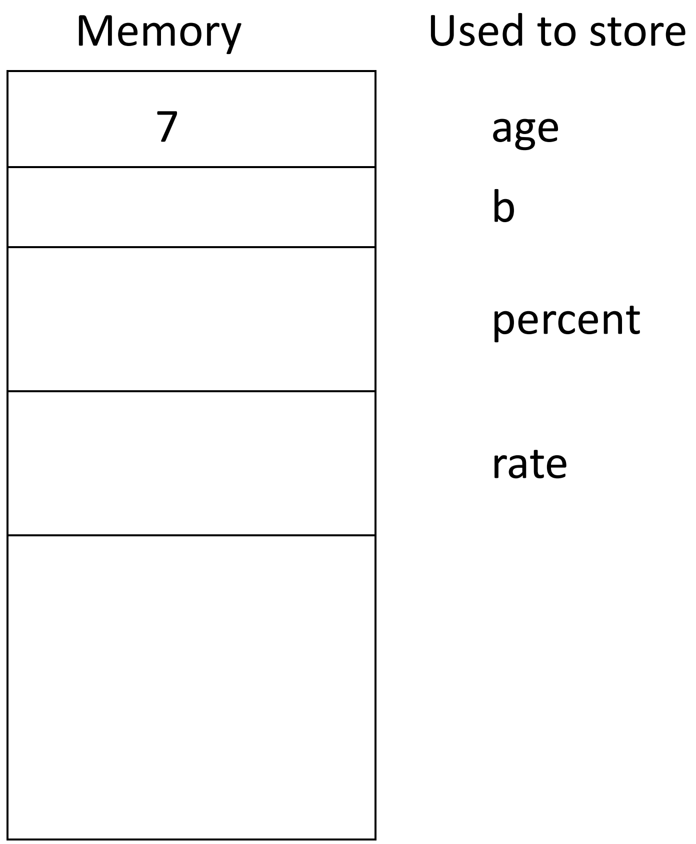
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (7)

So considering a simple main() programme

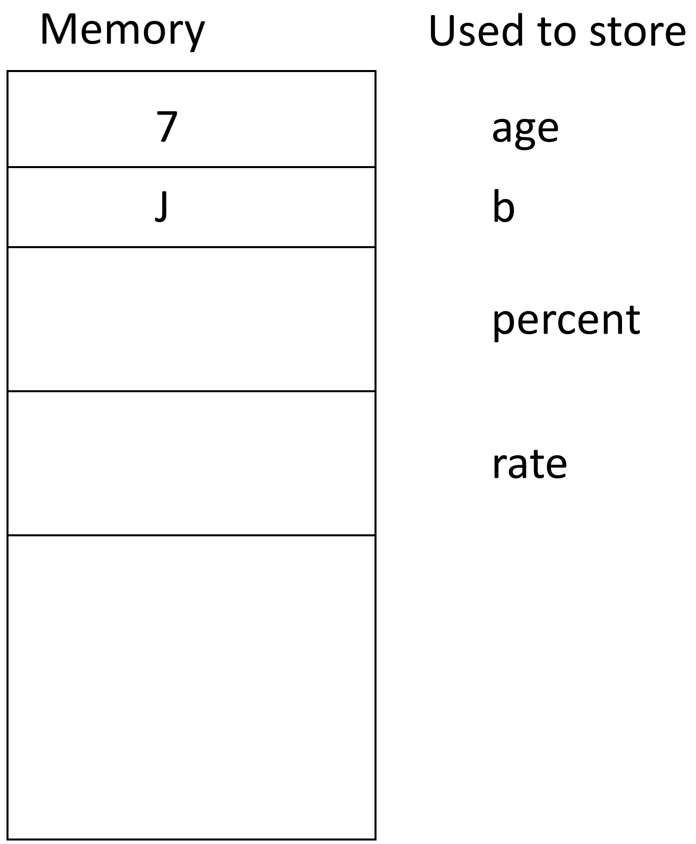
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = 'J';
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (8)

So considering a simple main() programme

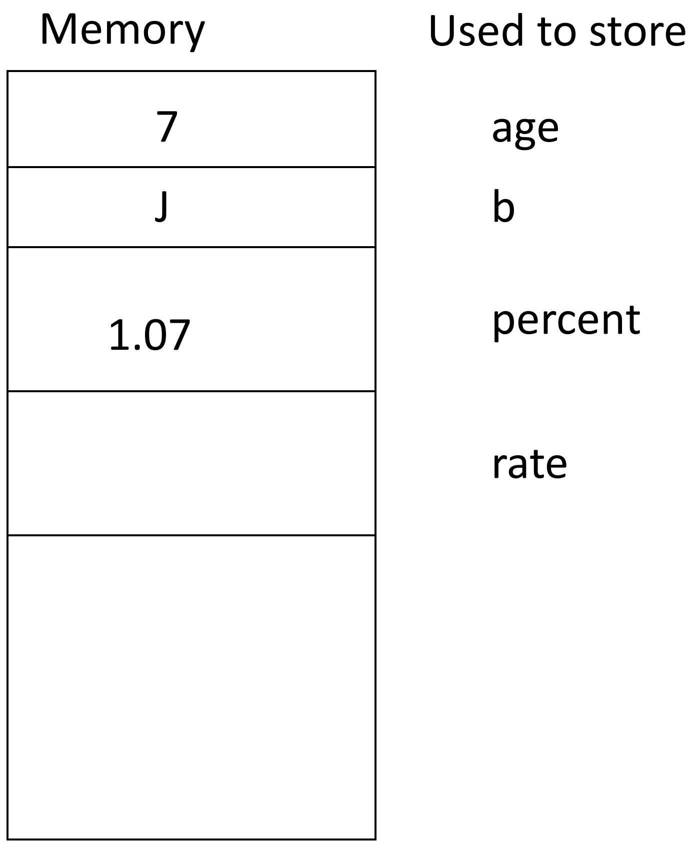
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (9)

So considering a simple main() programme

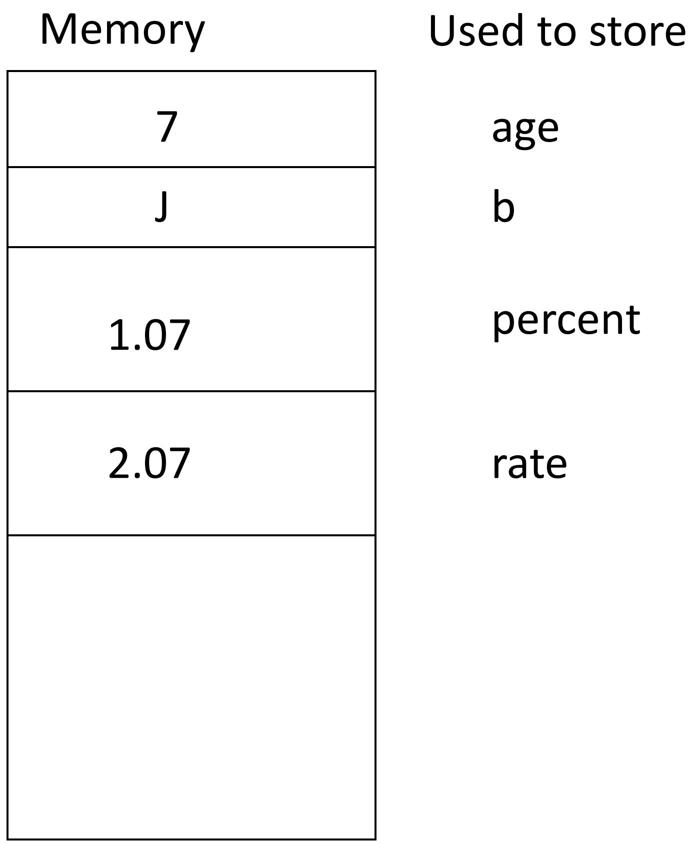
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (10)

So considering a simple main() programme

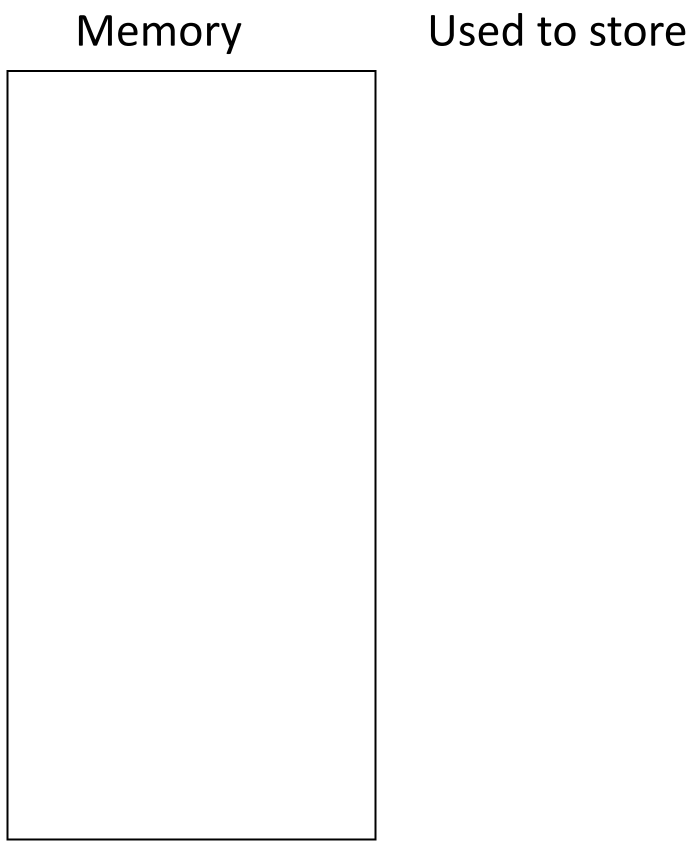
```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (11)

- So considering a simple main() programme

```
int main( void )
{
    int  age;
    char b;
    double percent, rate;

     age = 7;
     b = ‘J’;
     percent = 1.07;
     rate = percent + 1.0;

     return 0;
}
```



Memory Usage in C/C++ (12)

Now what happens when we start using functions?

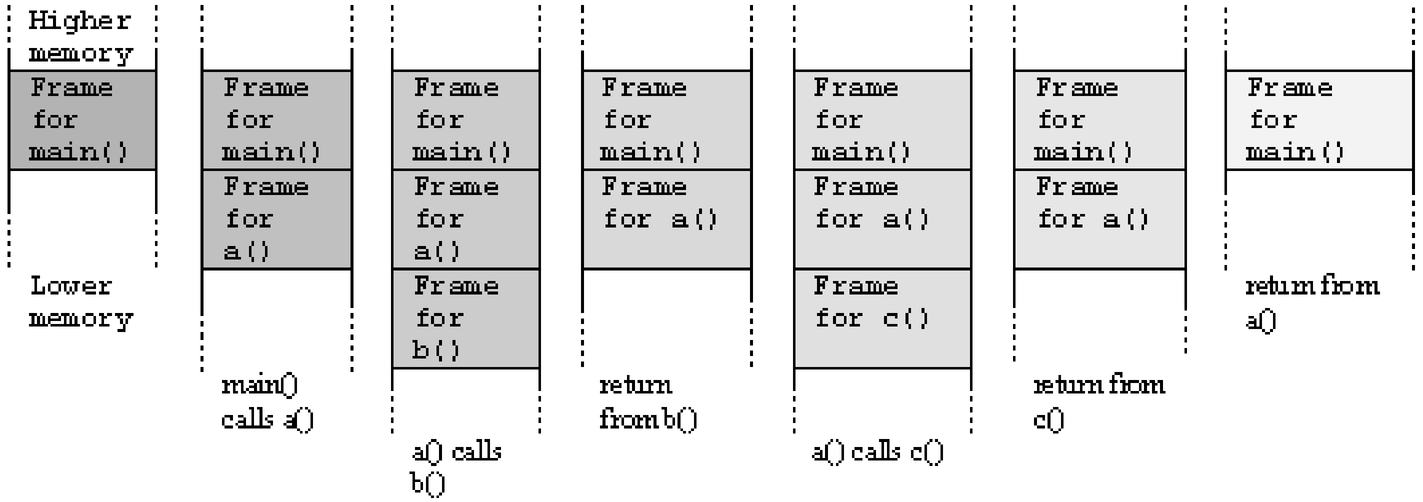
- Each time a function is called the same thing happens as in the case of main
  - Variables are created as they are needed
  - And the memory released when the return statement is executed

- In graphical form..

Memory Usage in C/C++ (12)

http://www.tenouk.com/ModuleZ.html

```
int a (void );  // Function prototypes
int b (void );  // memory is not allocated until
int c (void );  // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```

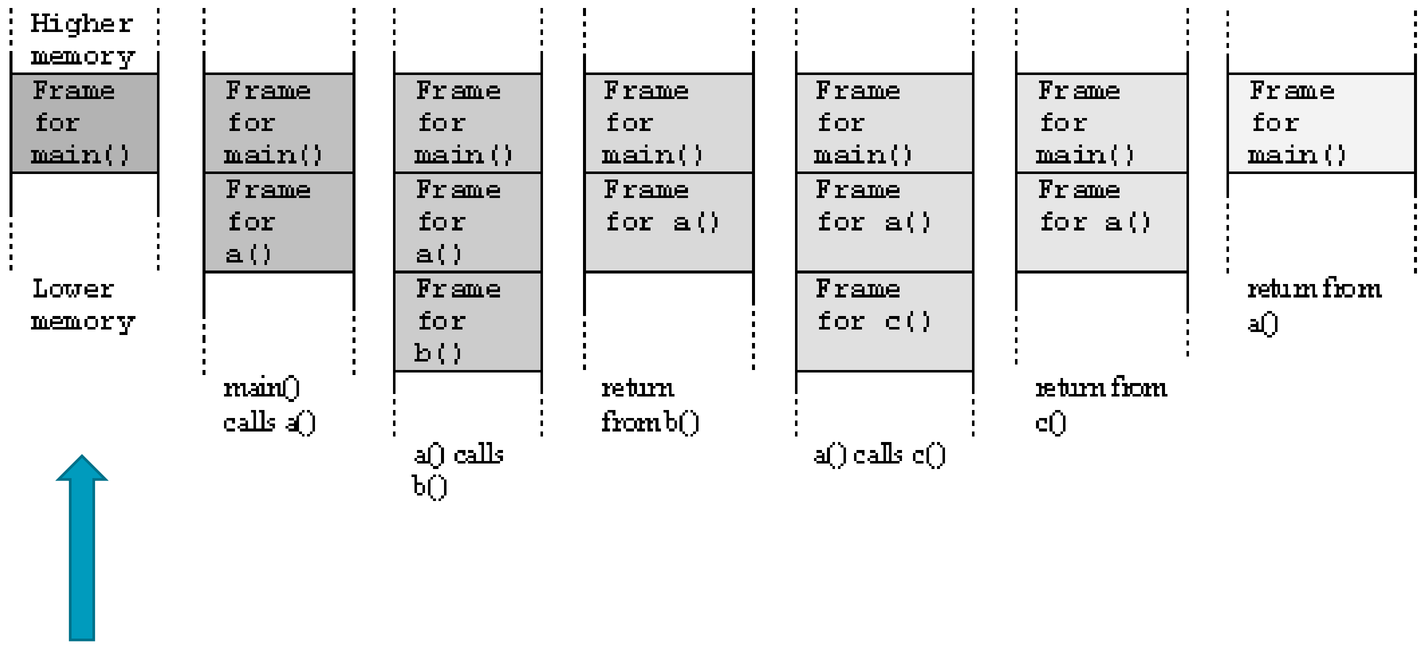


A 'Frame' is the term for the block of memory used by a function

Memory Usage in C/C++ (12)

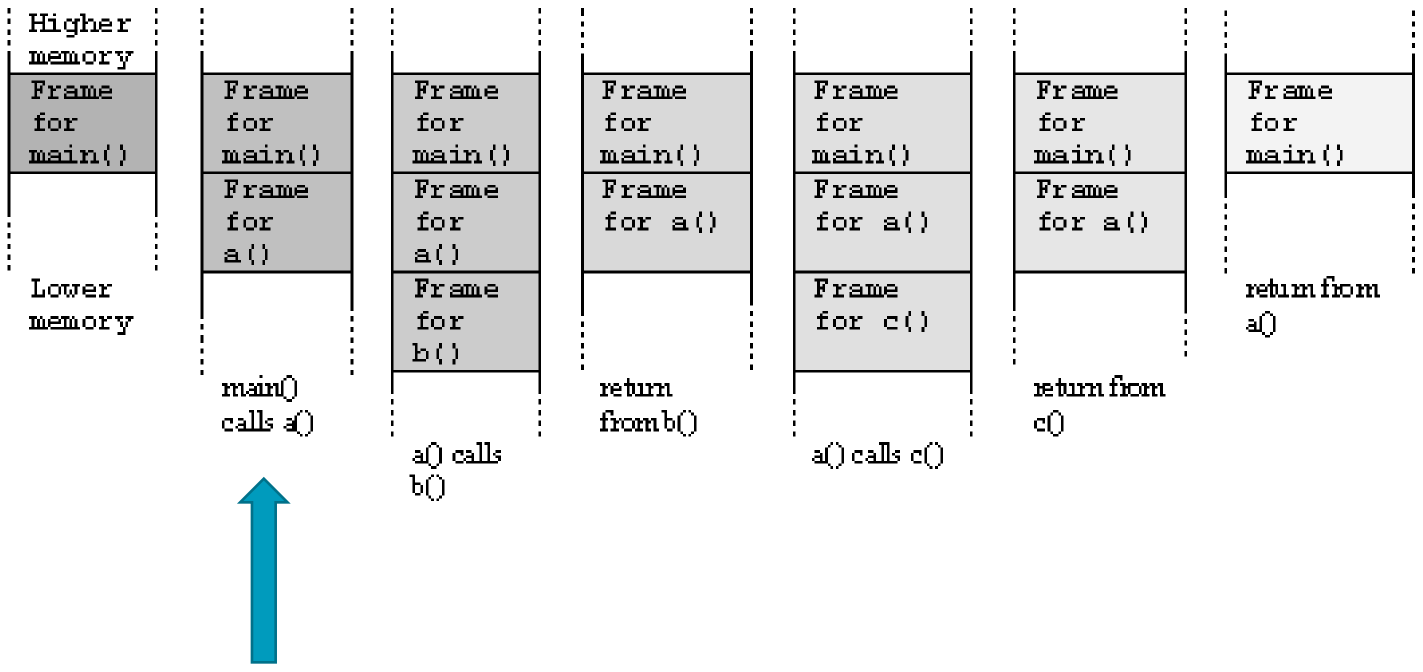
http://www.tenouk.com/ModuleZ.html

```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (13)

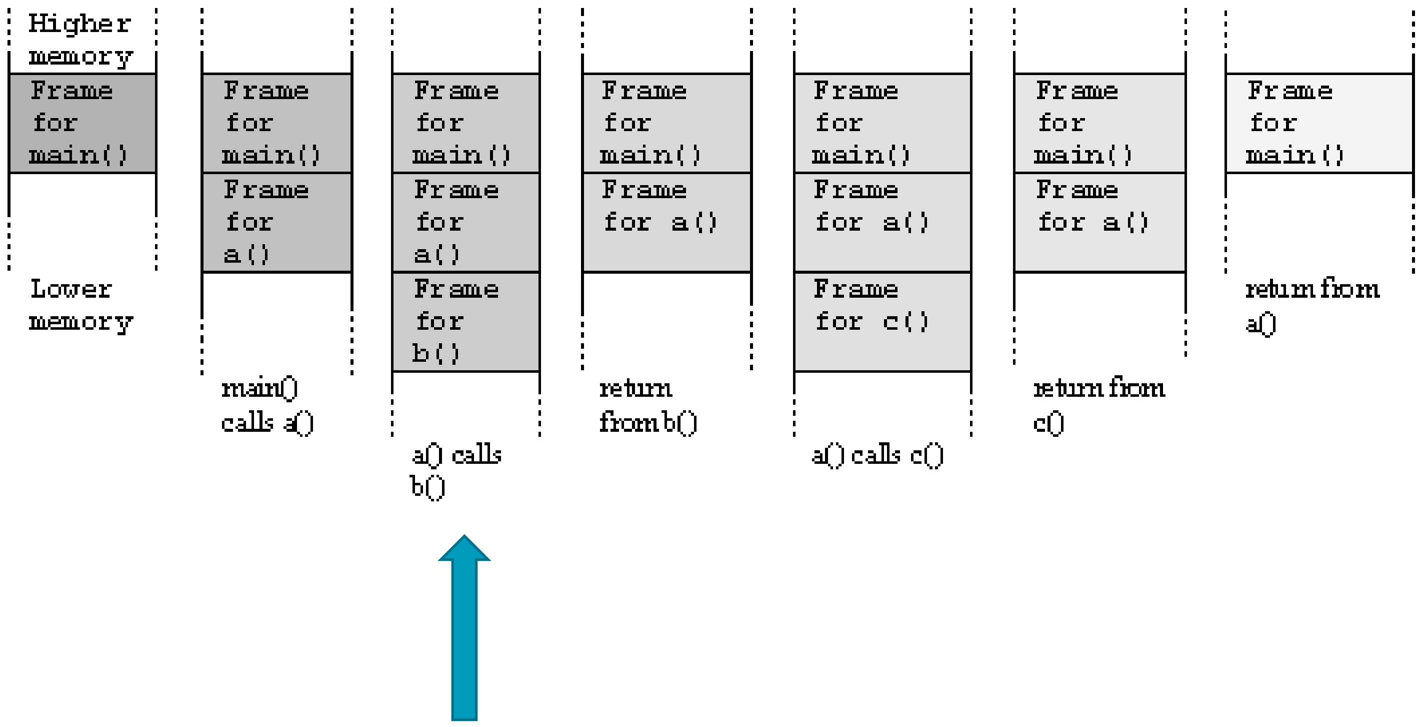
```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (14)

http://www.tenouk.com/ModuleZ.html

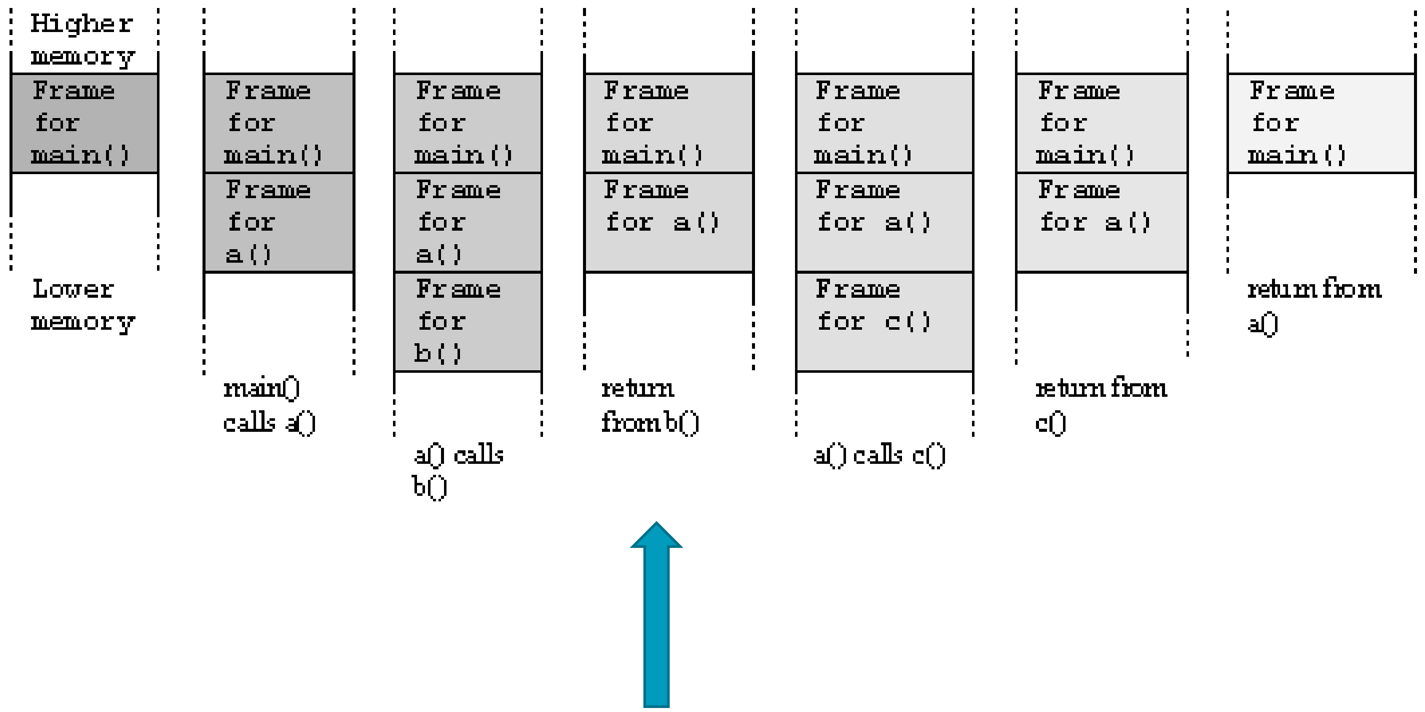
```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (15)

http://www.tenouk.com/ModuleZ.html

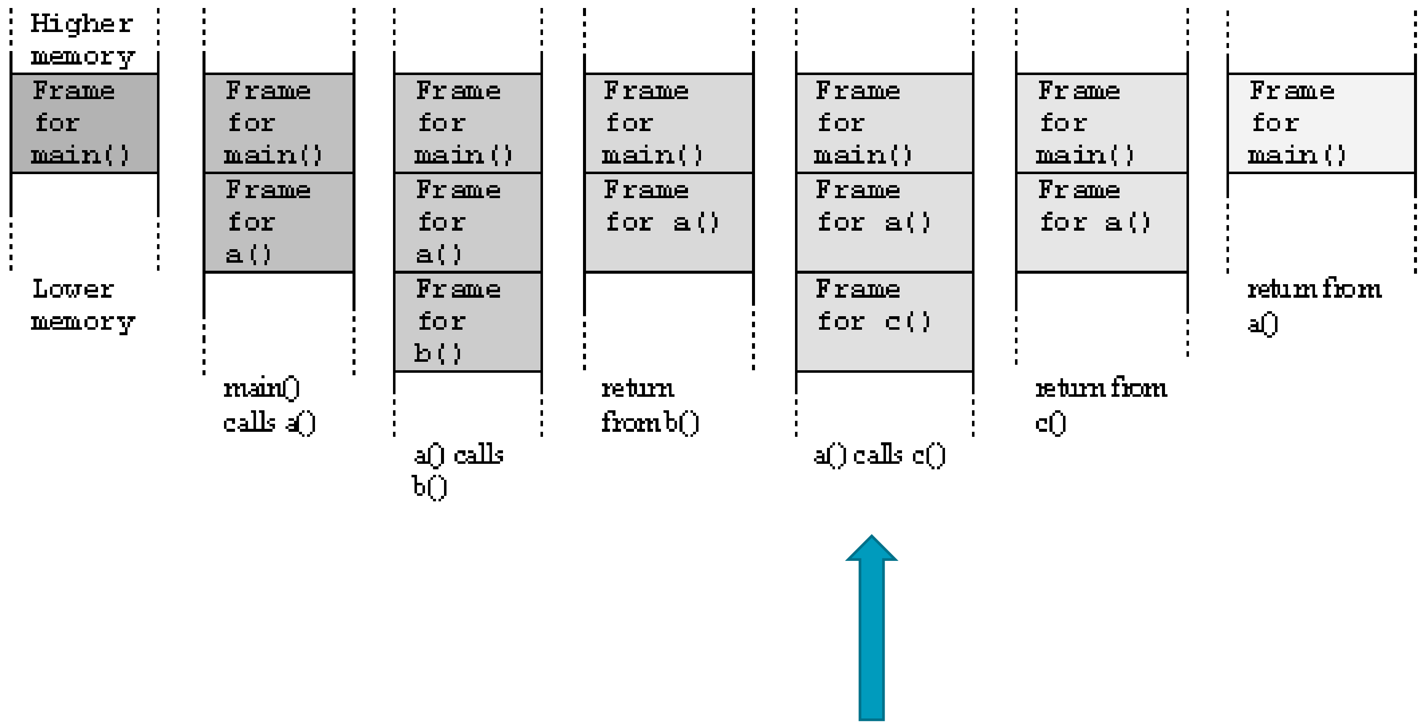
```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (17)

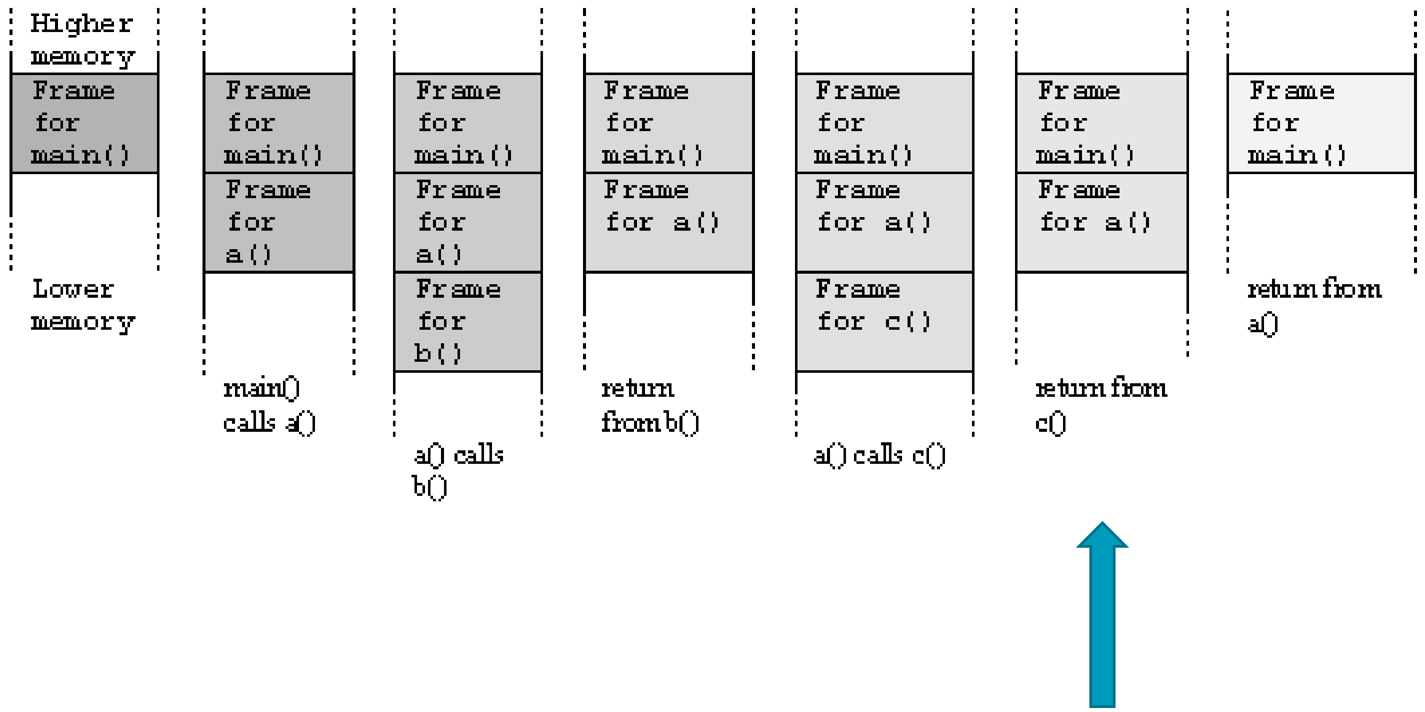
http://www.tenouk.com/ModuleZ.html

```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (18)

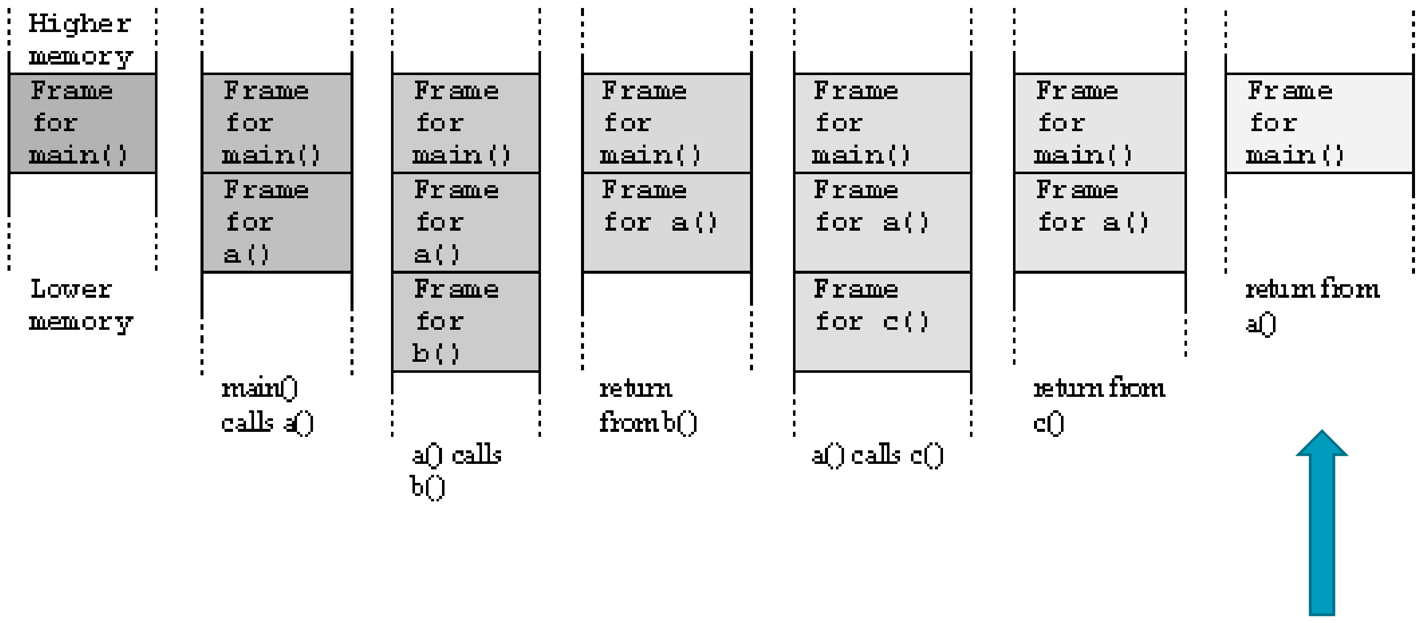
```
int a (void );   // Function prototypes
int b (void );   // memory is not allocated until
int c (void );   // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (19)

http://www.tenouk.com/ModuleZ.html

```
int a (void );  // Function prototypes
int b (void );  // memory is not allocated until
int c (void );  // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

int main( void )
{
    a();
    return 0;
}
```



Memory Usage in C/C++ (20)

Let us consider a function that takes parameters:

- For example a ‘CalculateArea’ function defined as

```
double CalculateArea ( double )
```

- When we call the function,
  - Memory is allocated for variable(s) to hold the parameters that are being passed
  - The value(s) passed are COPIED into these newly created variable ready for us to use them.

It is EXTREMELY important that you remember this!

Memory Usage in C/C++ (21)

Let us consider this graphically...

Memory Usage in C/C++ (22)

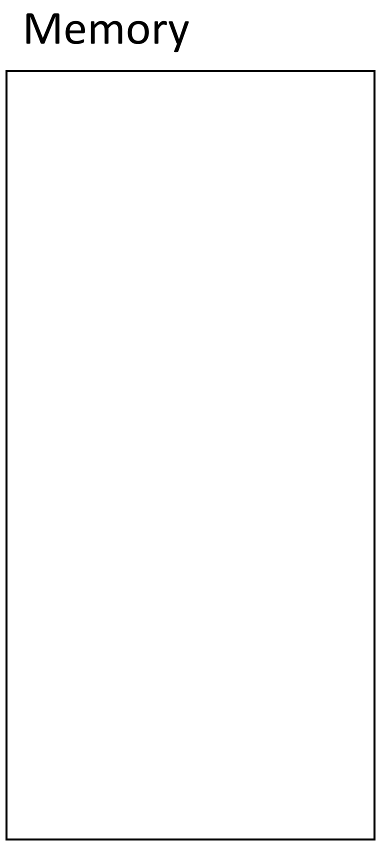
First a reminder…

When a function is called the parameters passed to it are **COPIED** into new variables local to the function

(just in case you forgot ☺)

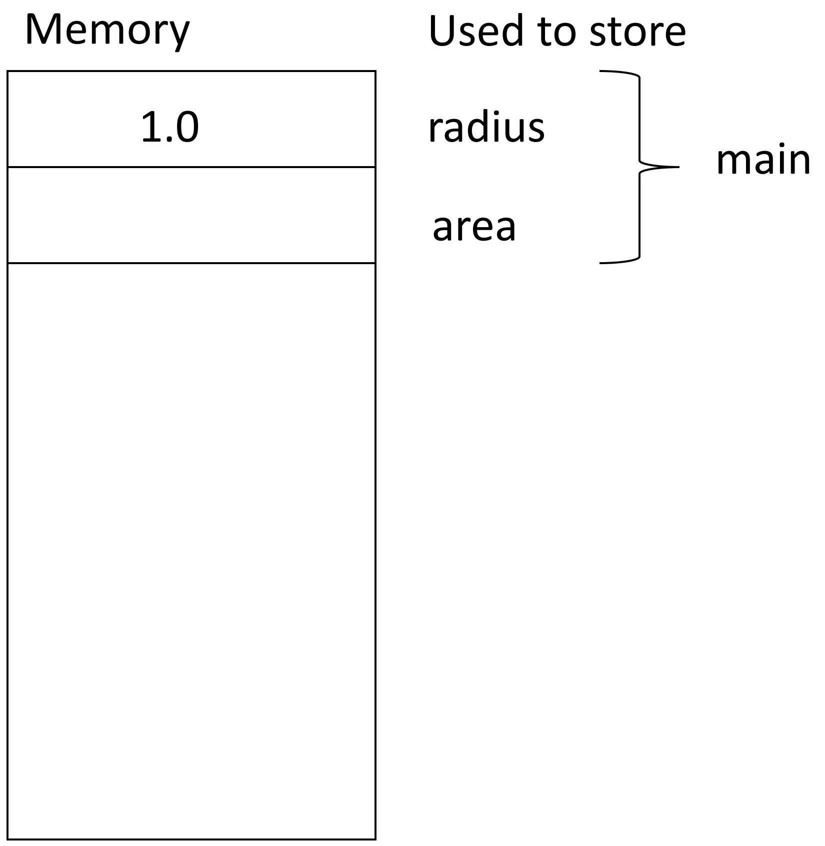
Memory Usage in C/C++ (23)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



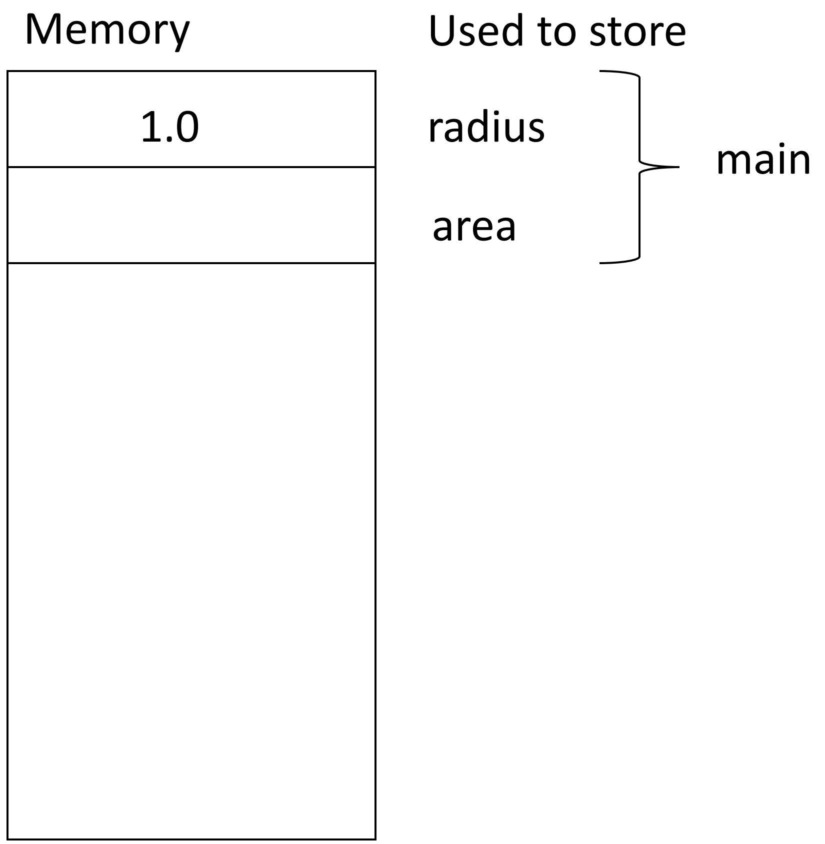
Memory Usage in C/C++ (24)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



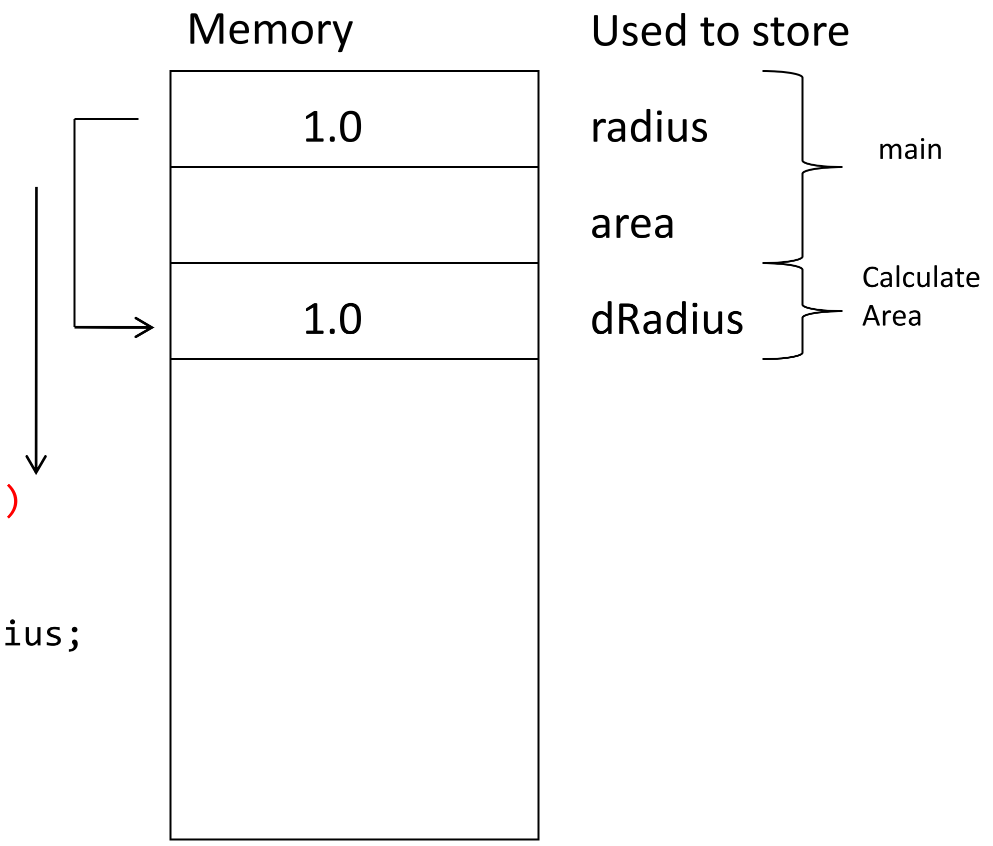
Memory Usage in C/C++ (25)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



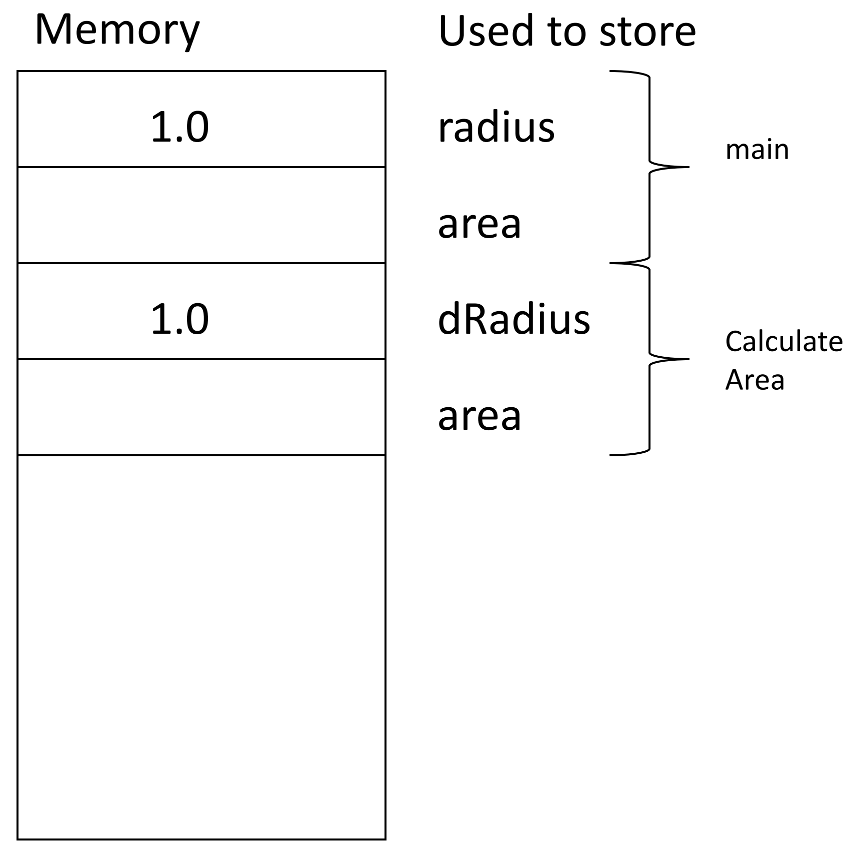
Memory Usage in C/C++ (26)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



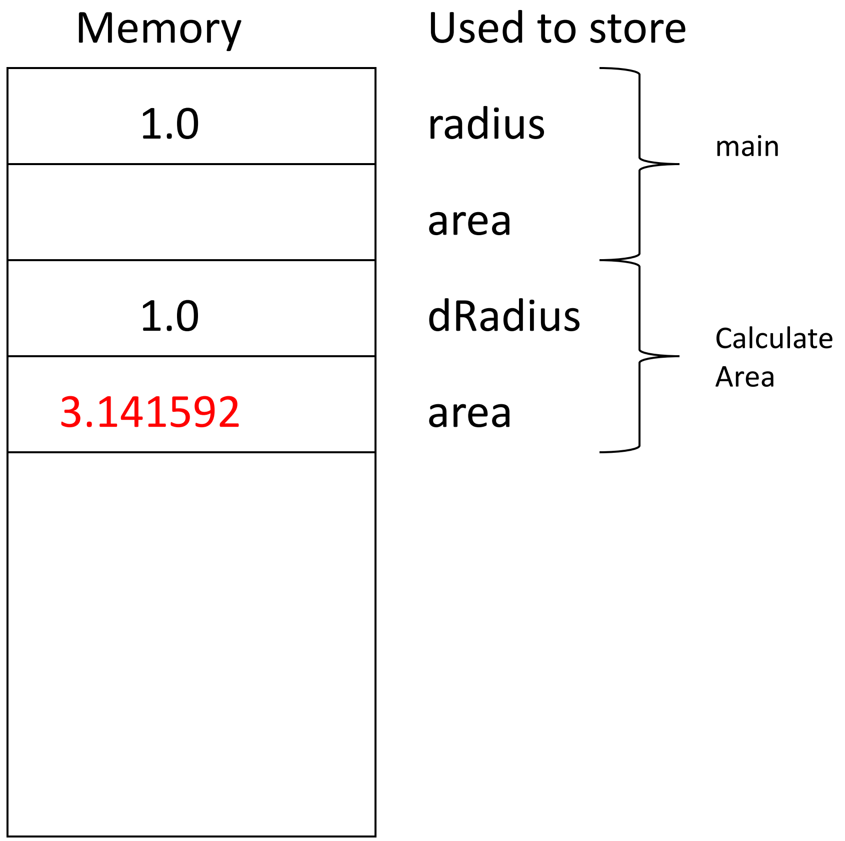
Memory Usage in C/C++ (27)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



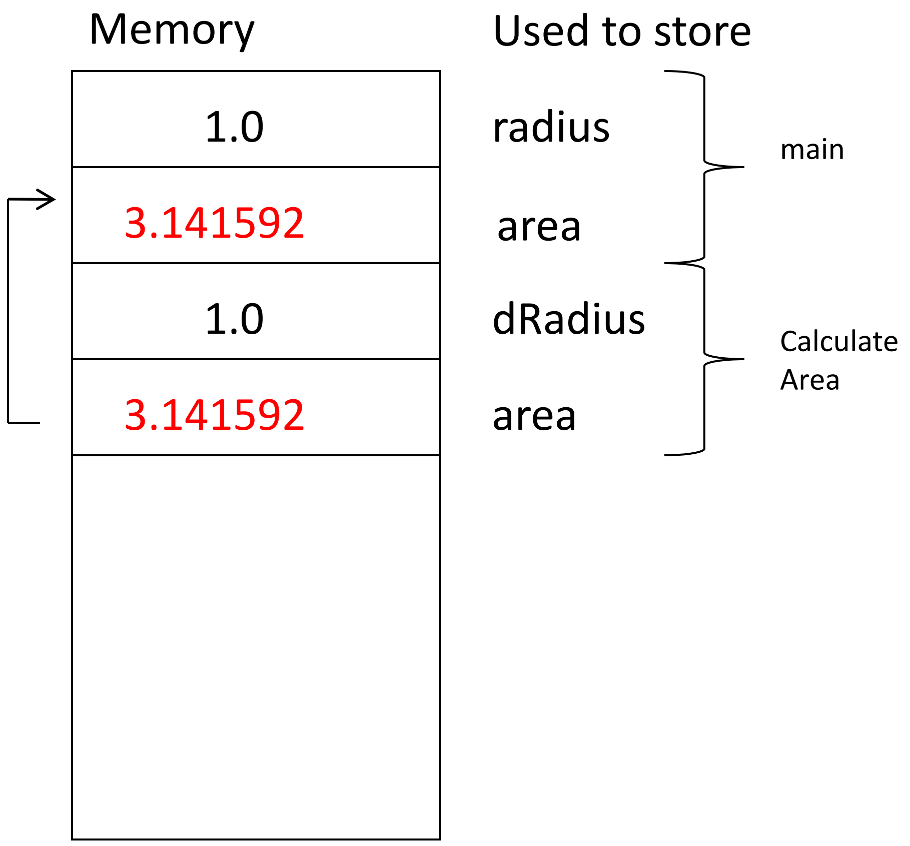
Memory Usage in C/C++ (28)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.141592 * dRadius * dRadius;
    return ( area);
}
```



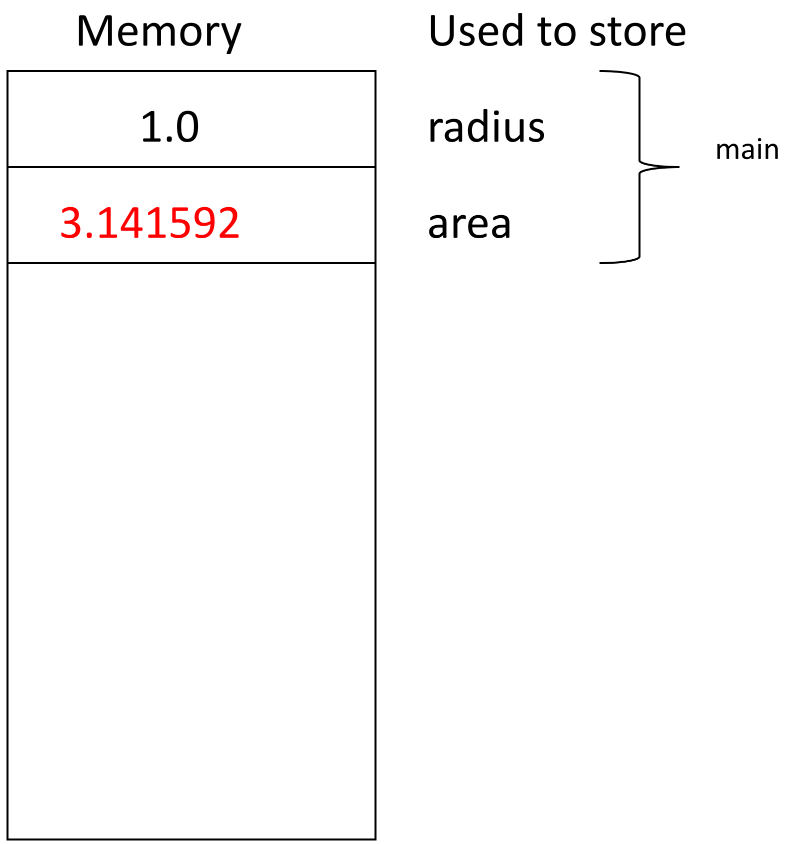
Memory Usage in C/C++ (29)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.141592 * dRadius * dRadius;
    return ( area);
}
```



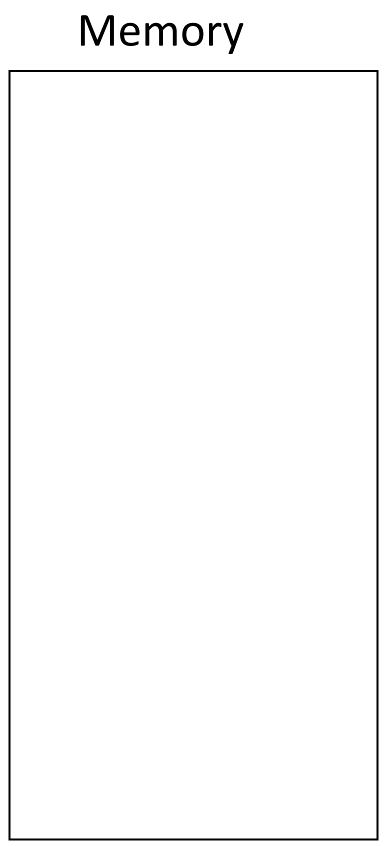
Memory Usage in C/C++ (30)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



Memory Usage in C/C++ (31)

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



VSCode and main()

Any C program can only have one main() function

main() is the entry point for the program

A folder can have many files but only one of them can contain a main() function

Why are global variables bad

Defined at the top, outside of any function and so available to ALL functions

HOWEVER: Avoid them as they:

- Hinder modularization
- Can cause VERY ODD behavior
- Are not even permitted in some languages (this is a good thing!)

Basically: **THEY ARE EVIL!**

Why are global variables bad: proof!

- This is the only time I will ever show code with a global variable in it!

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

int y,k,ans;                              /* Define GLOBAL variables */

void NastyGlobalFunction (void )     /* Define function */
{
    ans =  ( y * k );                     /* y, k and ans are defined globally above */
    return ;
}

int main( void )
{
   y = 2;              /* Set value of y */
   k = 3;              /* Set value of k */

   NastyGlobalFunction();   /* call the function */

   printf("%d multiplied by %d is %d " ,y ,k ,ans );   /* Display values */
   return 0;
}
```

LC12\global_ex1.c , LC12\global_ex2.c , LC12\global_ex3.c