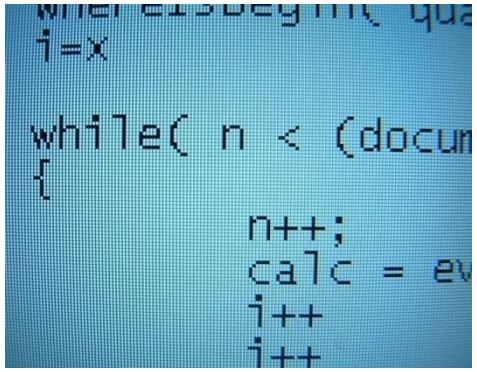
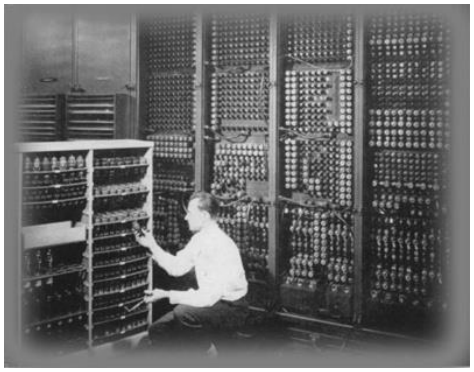




CS1PR16

Stack Memory Organization

Learning Objectives

University of Reading

- Describe how the compiler and runtime manage memory
- Define function call stack
- Defining “Call by Value“
- Execute a code snipped sequentially similarly to a computer
- Illustrate the mechanisms of passing data during the execution of a program on an example program

Outline

University of Reading

- The organization of main memory from OS to program
- Processing of programs
- Calling of functions: the Activation Record

Organization of Memory

University of Reading

- Remember: Memory contains bytes of binary data
  - Can be thought of as an array of bytes, the offset is the address
- Actually: the operating system (OS) provides **pages** of memory
  - The OS provisions the physical memory and distributes it
  - Pages have a granularity of, e.g., 4096 Bytes
- Address space covers a range of 0 to 64-bit (on 64 bit systems)
  - An application receives its own **virtual address space**
  - The OS maps the physical pages into space
- The processor supports the translation
  - Virtual <-> Physical addresses
  - Via the Memory Management Unit (MMU)

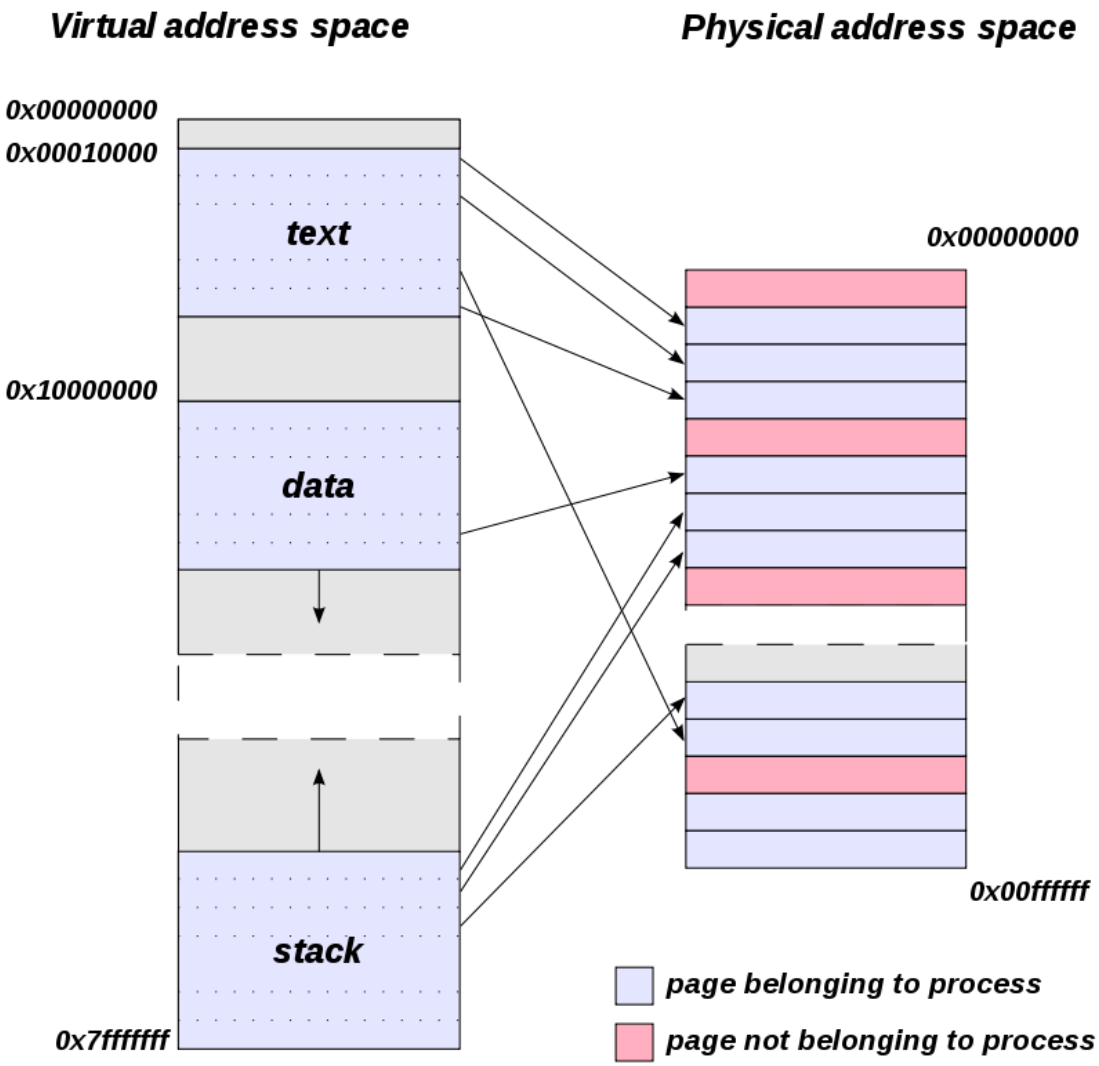
Memory Needed By Applications

University of Reading

- Data and code are both stored in memory
  - Managed by the Operating System (OS)
- When an application is executed (simplified)
  - The OS reserves memory
  - Instructions are loaded into memory into the so-called text segment
    - Nowadays into read-only memory to reduce attacks...
  - Libraries are blended (read-only) into **the virtual address space**
    - Their memory can be shared between all processes!
  - **Stack**: Memory is reserved for local variables and function calls
    - This particular memory is called stack
    - It is limited
  - The main() function is invoked

The Address Space

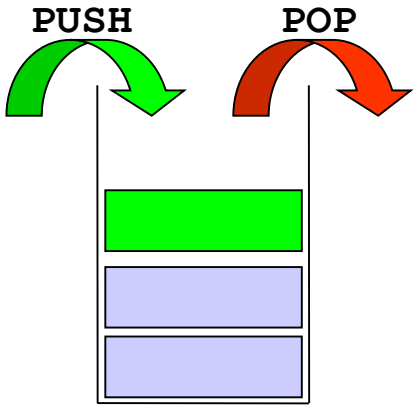
University of Reading



Source [Wikipedia]

The Stack

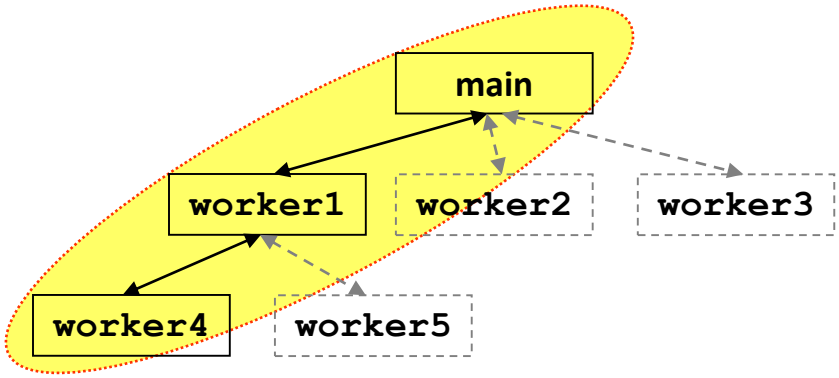
University of Reading

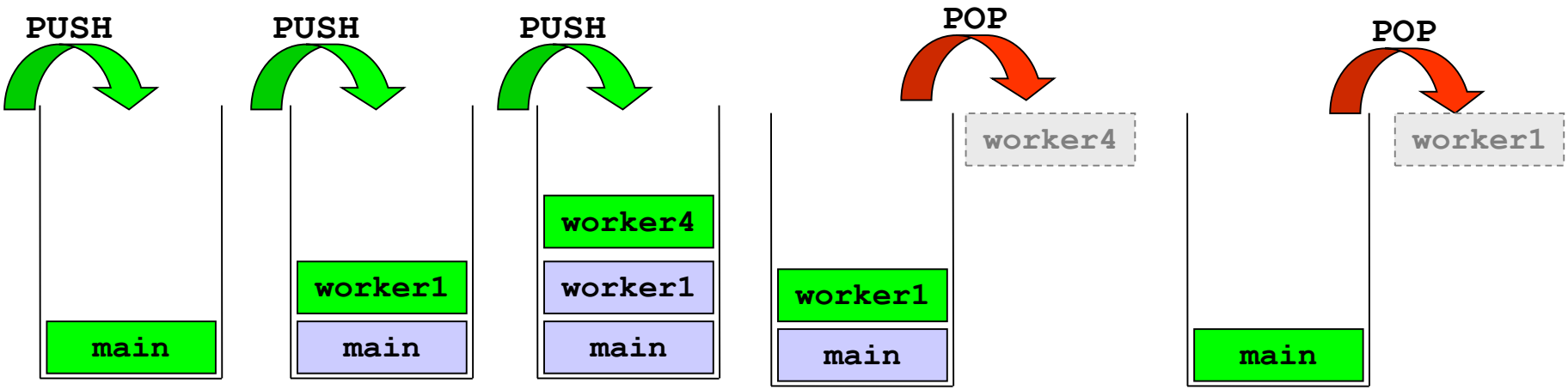


- Stack
  - An abstract data type (more about that later)
  - Operations:
    - Push() – adds data on top
    - Pop() – remove the last added data
  - Last In First Out policy (imagine a paper stack)
- The stack is used to manage function calls and storage for them
  - Call a function: push to stack
  - End a function: pop it from the stack

University of Reading

Function Call Stack





Function Call Stack

University of Reading

- Function Activation Record (AR)
  - The stack frame contains:
    - The function activation record
    - Memory allocation for local variables, including input parameters
    - The instruction pointer (return address) of the caller
      - so we can continue once the function returns
    - The return value (this value is then copied over!)
- When starting an application, the AR of main() is pushed
  - The local variables are organized by the compiler!
    - Relative to the location on the stack
  - As programmers, we should not care how, but we will learn it ☺

Function Call Stack: Example

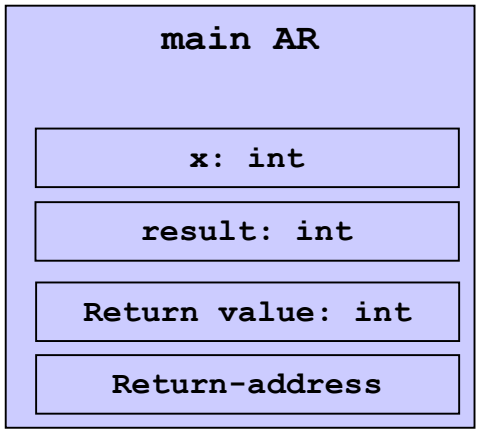
University of Reading

Assume: the Instruction Pointer is on the marked line

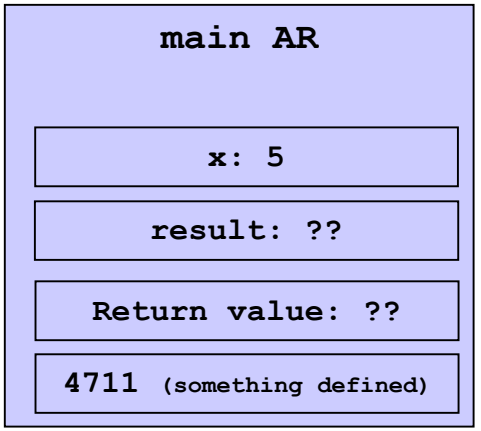
IP →

```
int main(){
    int result;
    int x = 5;
    result = worker1(x);
    return result;
}
```

For the code, the compiler may have assigned Stack memory like this:



At the IP, the values look like:



Some are not initialized "??"

Upon returning to the caller, we do know where the return value is and we can copy it over

Function Call Stack: Example

University of Reading

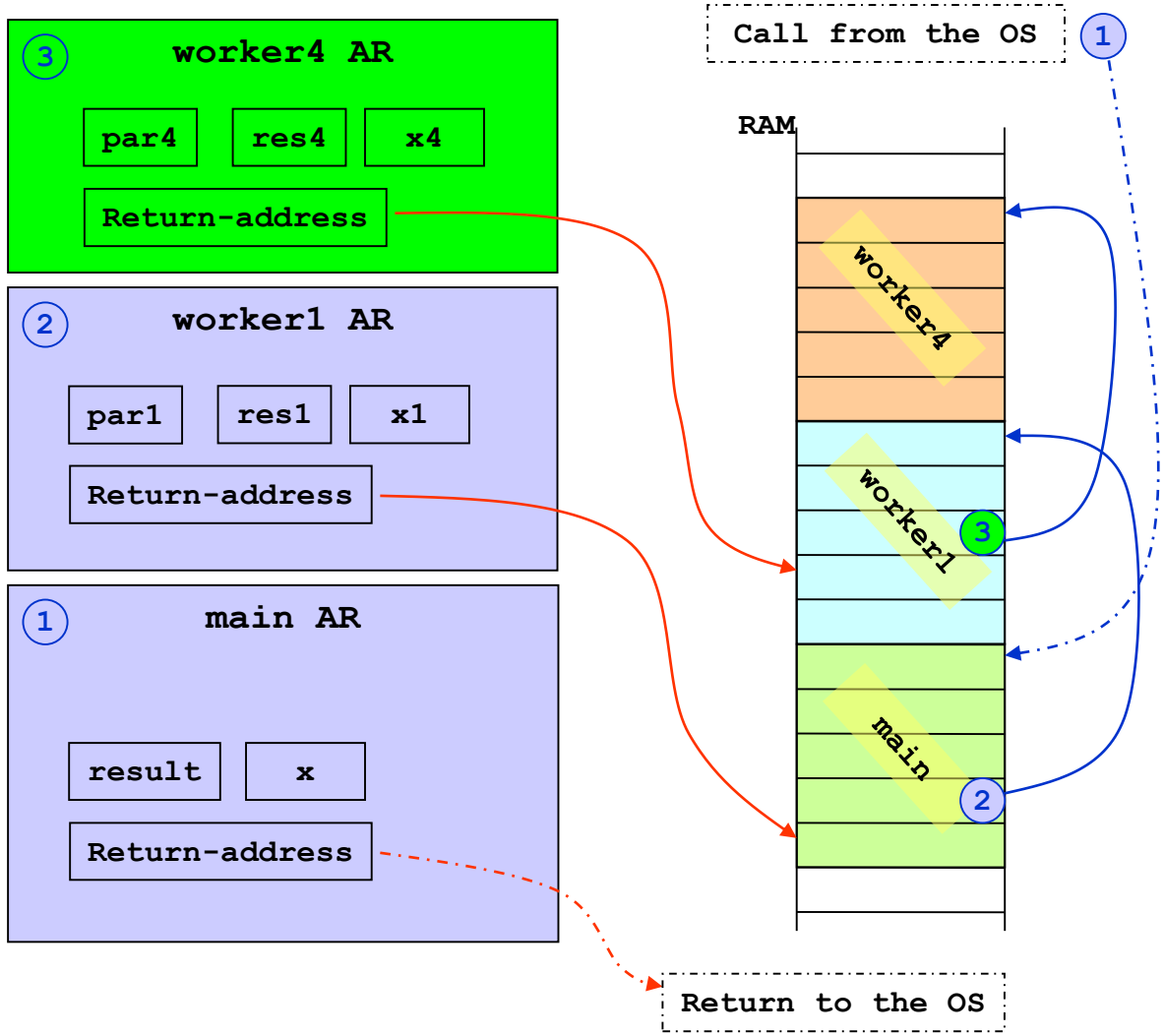
```
/* file: example.c */
#include <stdio.h>
...

int worker4(int par4){
    int res4;
    int x4;
    ...
    return res4;
}

int worker1(int par1){
    int res1;
    int x1 = 3;
    ...
3   res1 = worker4(x1);
    ...
    return res1;
}

int main(){
    int result;
2   int x = 5;
    result = worker1(x);
    ...
}
```

```
1   gcc example.c
    ./a.out
```



Function Call Stack: Actual Values

University of Reading

Assume: the Instruction Pointer is on the marked line

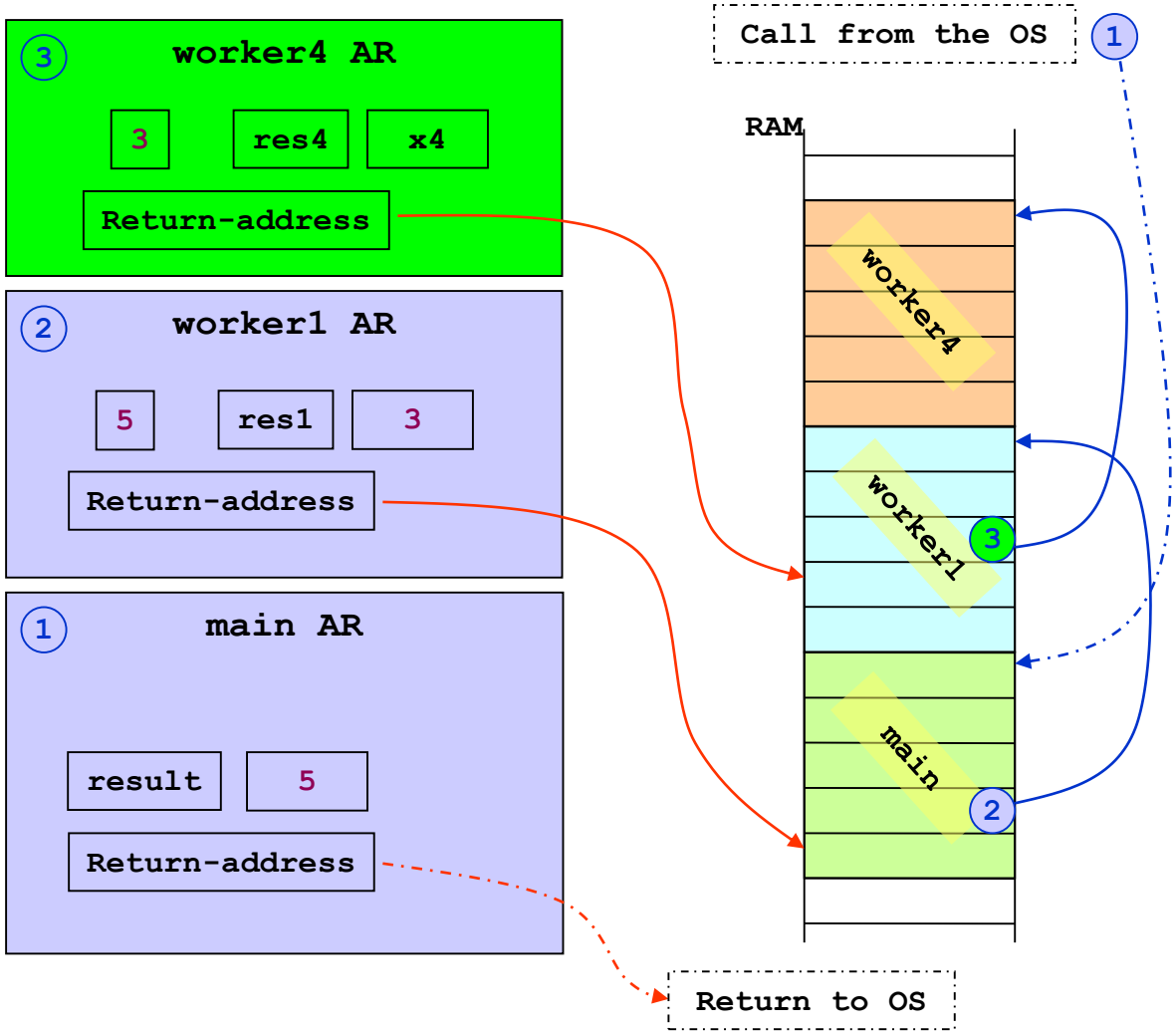
```
/* file: example.c */
#include <stdio.h>
...

int worker4(int par4){
    int res4;
    int x4;
    ...
    return res4;
}

int worker1(int par1){
    int res1;
    int x1 = 3;
    ...
3   res1 = worker4(x1);
    ...
    return res1;
}

int main(){
    int result;
2   int x = 5;
    result = worker1(x);
    ...
}
```

IP



Assembler

University of Reading

- This is our example from Lecture 3
  - Example for adding two numbers: a = 3; b = 4; c = a + b
- But extended by the stack pointer!
  - Actually, all the memory locations are relative to the register base pointer
  - RBP = Pointer to the stack frame!
- Assembler

```
movl    $3, -12(%rbp)
movl    $4, -8(%rbp)
movl    -12(%rbp), %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
movl    %eax, -4(%rbp)
```

All memory locations relative to RBP

Store number 3 into memory -12

Store number 4 into memory -8

Move memory -12 to register EDX

Move memory -8 to register EAX

Add EDX and EAX and store into EAX

Store EAX into memory -4

- Entering/leaving a function updates the stack pointer accordingly

Symbol Table

University of Reading

- The symbol table holds information about symbolic identifiers
- The compiler generates:
  - A table of the identifiers (variables and functions) defined in the source code
  - These are used during the compilation

name	type	Address
myVar1	D	0x000020
myFun1	t	0x000040
myVar2	d	0x000080

In general, the symbol table contains:

- **for each type name, its type definition.**
- **for each variable name, its type, storage class, offset in activation record etc.**
- **for each constant name, its type and value.**
- **for each function, its formal parameter list and its output type.**

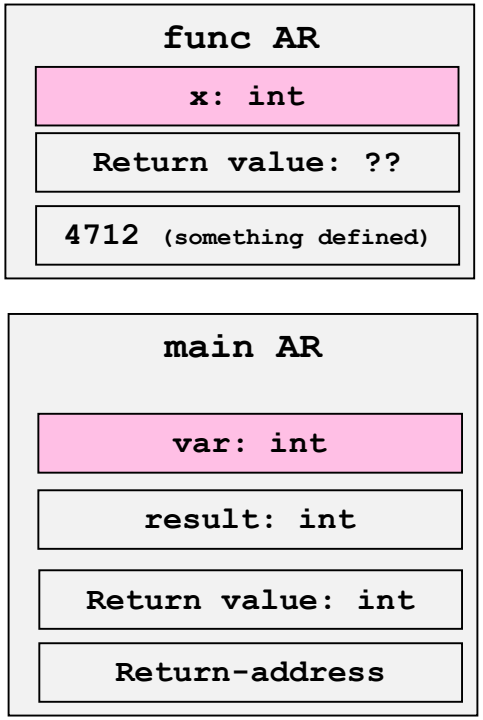
Function Calls: Call by Value

University of Reading

- There are two ways of calling a function:
  - Call by value (we learn today)
  - Call by reference (we'll learn soon!)
- Call by value
  - Copy of the argument passed to the function
  - Changes to the variable in the function cannot affect the original value
    - Avoids accidental change => isolation
- Example function:

```
int func(int x)
```

- In the example on the right:
  - Value of the variable x is copied



Call by Value: Example

University of Reading

- We will exemplify how data is changed in the stack when calling a function

```
// Calculates distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0; // Changes to x or y affect the local scope
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}
```

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    x = 3;
    y = 4;

    result = distanceToMidpoint(x,y);
    printf("\nDistance To Midpoint is: %f", result);

    return 0;
}
```

IP →

Name	Value
x	Uninitialized
y	Uninitialized
result	Uninitialized

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    x = 3;
    y = 4;

    result = distanceToMidpoint(x,y);
    printf("\nDistance To Midpoint is: %f", result);

    return 0;
}
```

IP →

Name	Value
x	3.0
y	4.0
result	Uninitialized

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    x = 3;
    y = 4;

    result = distanceToMidpoint(x,y);
    printf("\nDistance To Midpoint is: %f", result);

    return 0;
}
```

IP

Name	Value
X	3.0
Y	4.0
Distance	Uninitialized

Copied

Name	Value
x	3.0
y	4.0
result	Uninitialized

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    x = 3;
    y = 4;

    result = distanceToMidpoint(x,y);
    printf("\nDistance To Midpoint is: %f", result);

    return 0;
}
```

IP →

Name	Value
x	1.5
y	4.0
distance	Uninitialized

Name	Value
x	3.0
y	4.0
result	Uninitialized

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    x = 3;
    y = 4;

    result = distanceToMidpoint(x,y);
    printf("\nDistance To Midpoint is: %f", result);

    return 0;
}
```

IP →

Name	Value
x	1.5
y	2.0
distance	2.5

Name	Value
x	3.0
y	4.0
result	Uninitialized

Call by Value: Example

University of Reading

```
// Calculates Distance to midway between (0,0) and (x,y)
double distanceToMidpoint(double x, double y)
{
    double distance;
    x /= 2.0;
    y /= 2.0;
    distance = sqrt(x*x + y*y);
    return distance;
}

int main(void)
{
    double x, y, result;
    printf("Input X: ");
    scanf("%lf", &x);
    printf("Input Y: ");
    scanf("%lf", &y);

    result = distanceToMidpoint(x,y);
    printf("\nThe Distance To Midpoint is: %f" ,result);

    return 0;
}
```

IP →

Name	Value
x	3.0
y	4.0
result	2.5

Group Work

University of Reading

Task: Write down the call stack for the following code

- When the instruction pointer is on location 1 and location 2

```
int sqr(int x){
        // location 2 //
        return x*x;
}
int main(){
        int y = 5;
        // location 1 //
        return sqr(y);
}
```

Time: 5 min

Group Work: Solution

University of Reading

Location 1:

main AR

y = 5

return = ??

Location 2:

sqr AR

x = 5

return = ??

main AR

y = 5

return = ??

```
int sqr(int x){
        // location 2 //
        return x*x;
}
int main(){
        int y = 5;
        // location 1 //
        return sqr(y);
}
```

Summary

University of Reading

- The stack is a growing segment on the virtual memory
  - A stack is an abstract data type supporting push() and pop() operations
- The compiler assigns variables to memory locations
  - Relative to the stack pointer
- A function call requires to store information on the stack
  - Instruction pointer of the calling memory location => return address
  - Arguments
  - Space for the return value
  - That is the function activation record