Function Abstraction

$\sin x$

We have already seen how to compute an approximate value of $\sin x$ from the following series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \cdots$$

$\sin x$

- The computation of $\sin x$ gives a value like an expression.
- The value of $\sin x$ depends on the parameter x.
- If we can encapsulate the code for this computation as a named object that can be called (with parameters), then it can be used in different parts in a program.

Function Abstraction

- Encapsulated code is called a function in C and many other programming languages.
- A function has a name, it takes zero or more number of parameters [a], it has the type of the returned value[b] and the body of the code for computation.

[a] These are called formal parameters. They are specified with their types.

[b] A function may have a return type void. It does not return any value. The purpose of such a function is abstraction of a computation that causes side-effect.

Code for $\sin x$

```
xRadian = M_PI*x/180.0 ; term = xRadian ;
sineVal = term ; termNo = 1 ;
do {
    double factor ;
    factor = 2.0 * termNo++ ;
    factor = factor * (factor + 1.0) ;
    factor = - xRadian * xRadian / factor ;
    sineVal = sineVal + (term = factor * term) ;
    compError = 100.0*fabs(term/sineVal) ;
} while (compError >= precError) ;
```

Function Interface

The interface of the function to the other part of the program is as follows:

```
double mySin(double, double);
```

- the name is mySin,
- there are two formal parameters, both are of type double; one for the angle in degree and the other for the percentage error,
- the type of the return value is double.

Function Definition mySin()

```
#include <math.h>
#define ABS(X) (((X) < 0.0) ? -(X) : (X))
double mySin(double x, double precError){
  double xRadian, term, sineVal, compError ;
  int termNo ;

  xRadian = M_PI*x/180.0 ; term = xRadian ;
  sineVal = term ; termNo = 1 ;
  do {
     double factor ;
     factor = 2.0 * termNo++ ;
     factor = factor * (factor + 1.0) ;
```

```
    factor = - xRadian * xRadian / factor ;
    sineVal = sineVal + (term = factor * term) ;
    compError = 100.0*ABS(term/sineVal) ;
  } while (compError >= precError) ;
  return sineVal ;
} // sin1.c
```

Note

- The name of the formal parameters are x and precError.
- The variables x, precError, xRadian, term, sineval, termNo are local to the function mySin(). They are not visible to the other parts of the program.
- The variable factor is local to the statement-block of the do-while loop and is not visible outside it.

Note

The body of main() no more contains the code for sine computation. It invokes (calls) the mySin() function with the actual parameters. The first parameter is the angle for which we want the approximate sine value. The second parameter is the prescribed percentage error. Both the actual parameters can be expressions.

Parameter Passing

The value of the first actual parameter is copied to the location of the formal parameter x. Similarly the value of the second actual parameter is copied to the formal parameter precError. The actual computation within the function is done on the content of x and precError.

Return Value

The computed value in the called function (callee) is returned by the return statement and is used in the caller function like any other expression value.

Function main()

```
#include <stdio.h>
double mySin(double, double) ;
int main()
{
 double x, precError ;
 printf("Enter the value of an angle in Degree: ") ;
 scanf("%lf", &x) ;
 printf("\nEnter the Percentage Error: ") ;
 scanf("%lf", &precError) ;
 printf("\nsin(%f) = %f\n", x, mySin(x, precError)) ;
 return 0 ;
} // sin1.c
```

Note

- In the second line '**double mySin(double, double);**' provides the function interface (function prototype) information to the C compiler.
- The variables **x** and **precError** are local to main() and are not visible from other parts of the program. They are different from the **formal parameters** with the same name in mySin().

Note

```
printf("\nsin(\%f)=\%f\n",x,mySin(x,precError)) ;
```

- mySin(x, precError) is the invocation (call) of mySin() with the actual parameters x and precError.
- The value returned by mySin() is used as the 3rd parameter to printf().

Value Parameter

Actual parameters

Invocation: **...... mySin(x, precError) ....**

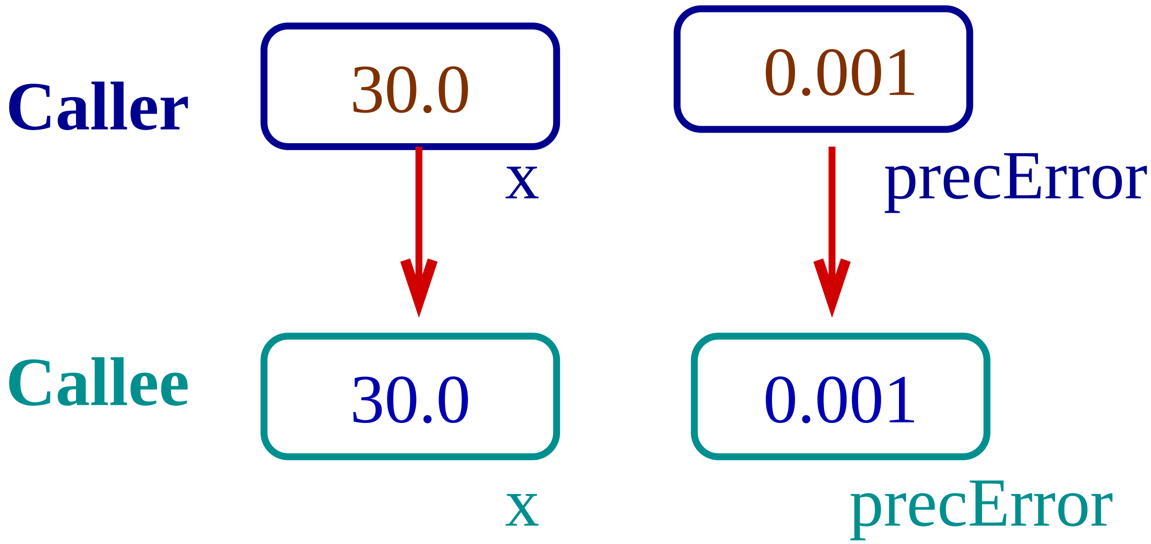
Value copy

Definition:

```
double   mySin(double x, double precError)
{
    .......................................
}
```

Formal parameters

Parameter Passing by Value



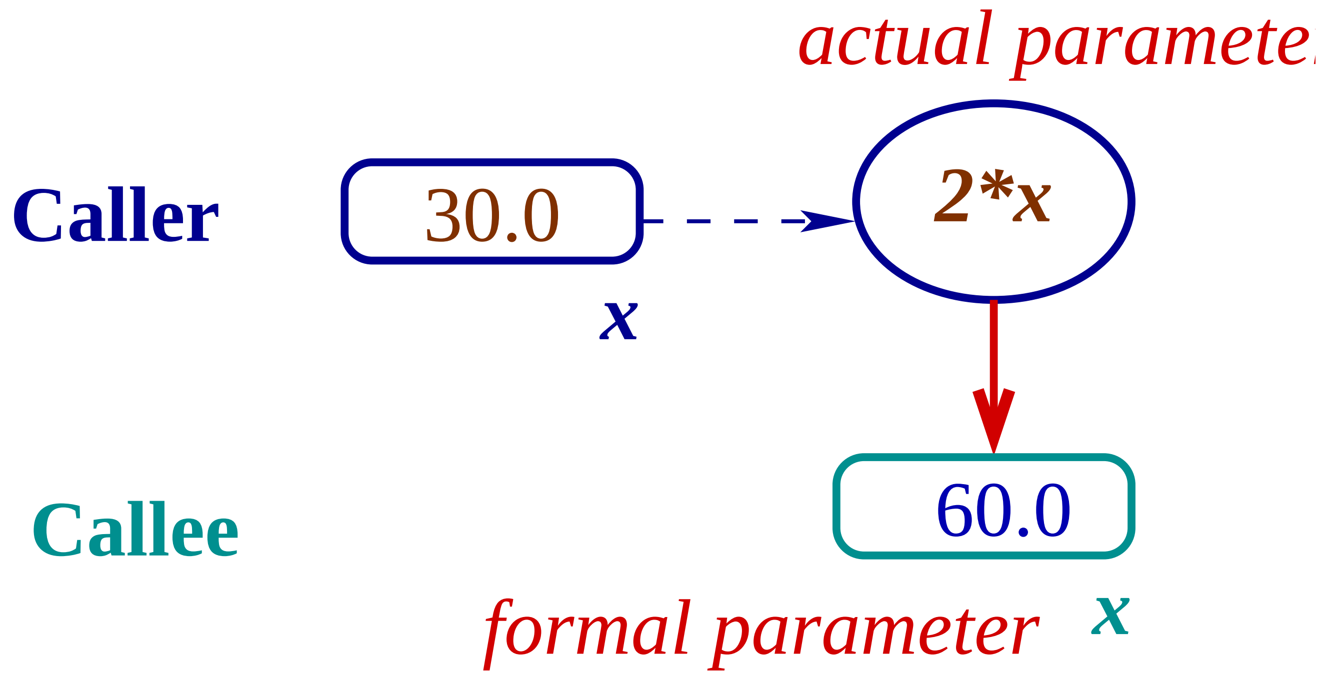
Expression as an Actual Parameter

```
printf(...., mySin(2.0*x, precError));
```

The value of `2.0*x` will be evaluated and passed as the actual parameter to mySin().

Note that the value of `mySin(2.0*y, precError)` is the actual parameter to the library function printf().

Expression as an Actual Parameter

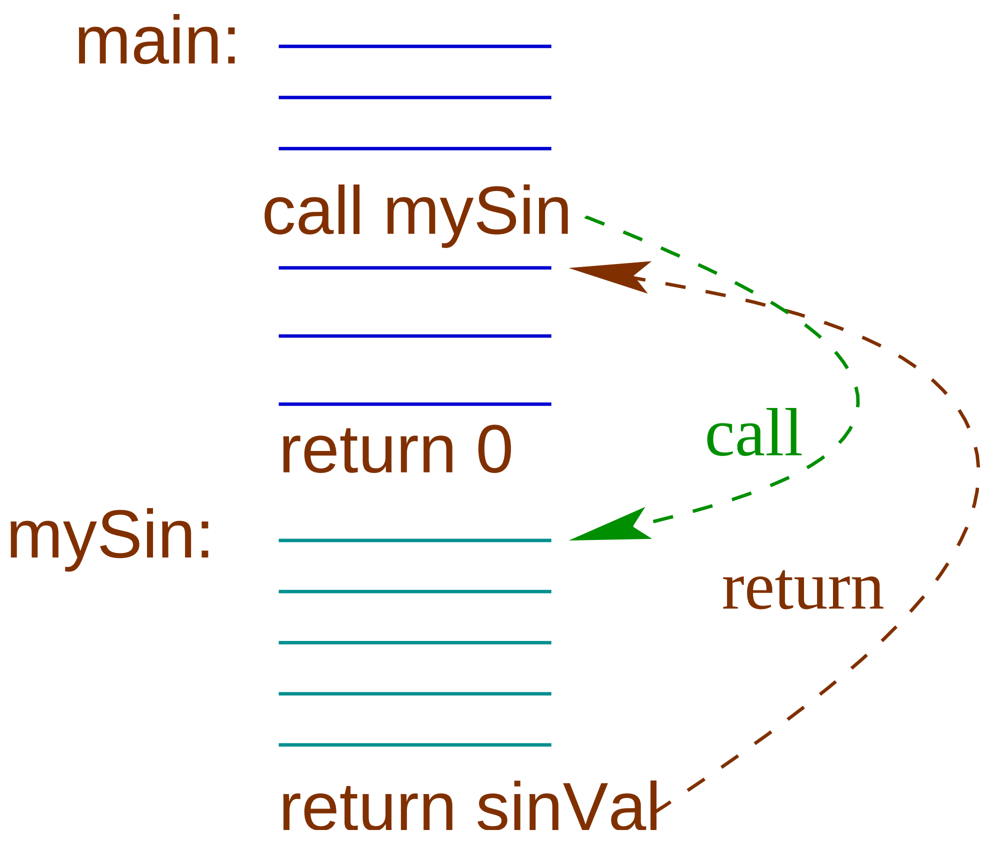


Flow of Control

When a function is called, the continuation of the computation (control is transfered to) is the beginning of the called function. Once the execution of the callee is over, the continuation is (control is transfered back to) the instruction next to the call in the caller[a].

[a]Difficult to show in the high level language.

Flow of Control

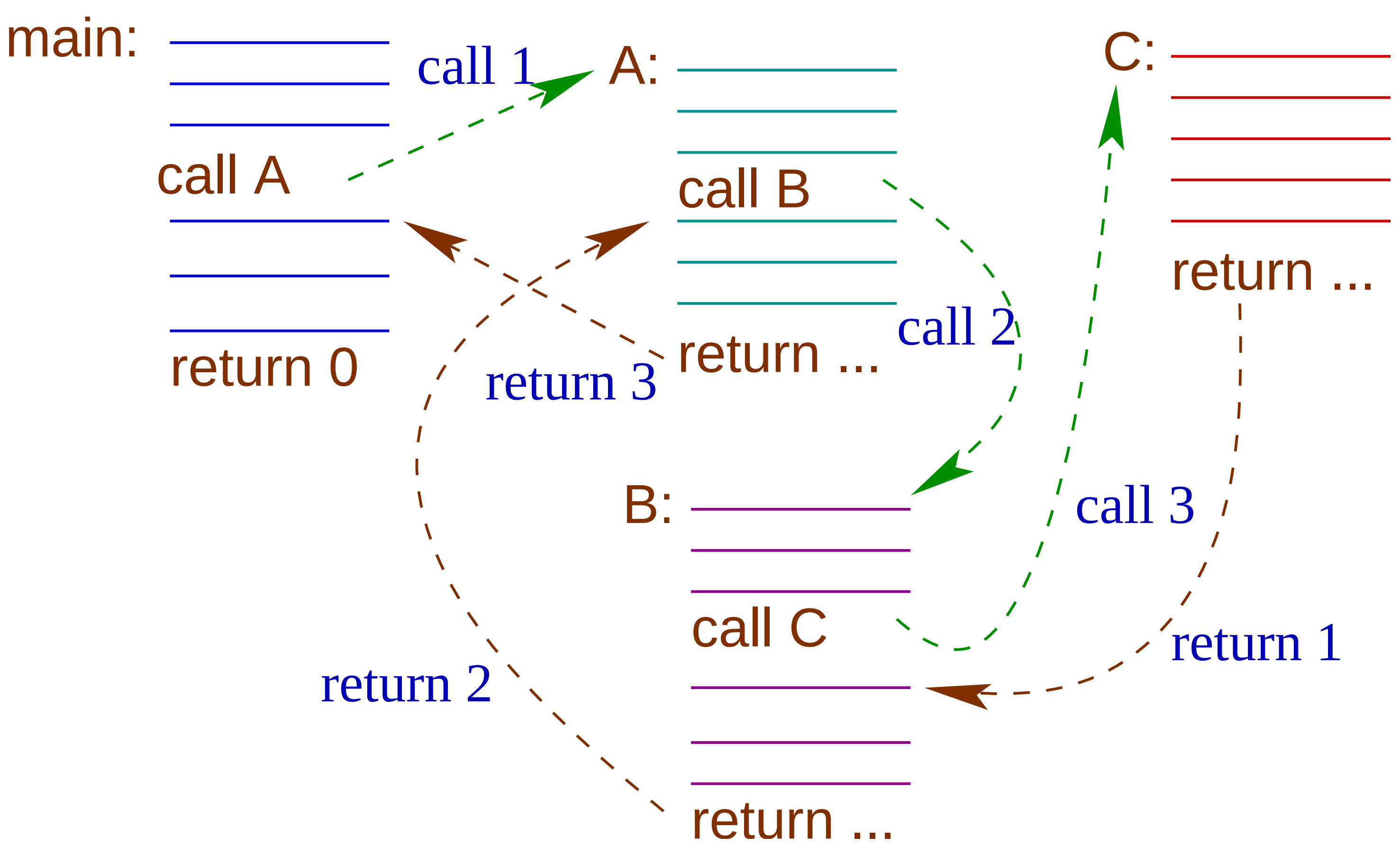


Flow of Control

If there is a sequence of nested function calls i.e. the function `main()` calls the function `A()`, which calls `B()`, which in turn calls `C()`. The function `C()` is completed first, then it is `B()`, then `A()` and finally `main()`.

The last invoked (called) function is completed first - a last in first out (LIFO) order.

LIFO Control Transfer



Flow of Control

As it is necessary for the caller function to have the starting address of the function to call (callee), it is also necessary for the callee to have the return address in the caller (instruction after the call) where the control is transfered on return.

Caller Address

A function is called by its name and its address (address of the first instruction) is often known a priori, at the compilation or linking time[a].

[a]This is not true in case of dynamic linking.

Return Address

But a function may be called from different placess (from different functions) within a program. So the return address from a called function to its caller is different on different call and can only be determined during the time of program execution.

Return Address

The return address is known (often) at the time of call itself - it is the address of the instruction next to the `call` instruction.

Return Address

A CPU provides architectural support to save the return address while processing the call instruction. The place to save the return address may be a CPU register and/or some specific memory area.

In case of nested calls, these addresses are used in LIFO order.

Space for Local Variables

We have already mentioned that both `main()` and mySin() have their local variables.

- `main()`: `x` and `precError`
- `mySin()`: `x`, `precError`, `xRadian`, `term`, `sineVal`, `compError`, `termNo` and `factor`.

Space for Local Variables

Local variables of one function is not visible from another function. Local variables (non-static) of a function get bound to memory only when the function is invoked. Following our previous example of the call sequence,

$$\texttt{main()} \xrightarrow{\texttt{calls}} \texttt{A()} \xrightarrow{\texttt{calls}} \texttt{B()} \xrightarrow{\texttt{calls}} \texttt{C()}$$

Space for Local Variables

Local variables of function

- `C()` are created last and are destroyed first,
- the variables of `main()` are created first and they live longest.

Here too we see the creation and the destruction in LIFO order like the return addresses from the called function.

Stack Region of Memory

Often the local variables of functions and return addresses live in a memory region maintained in LIFO order. This region is called stack[a].

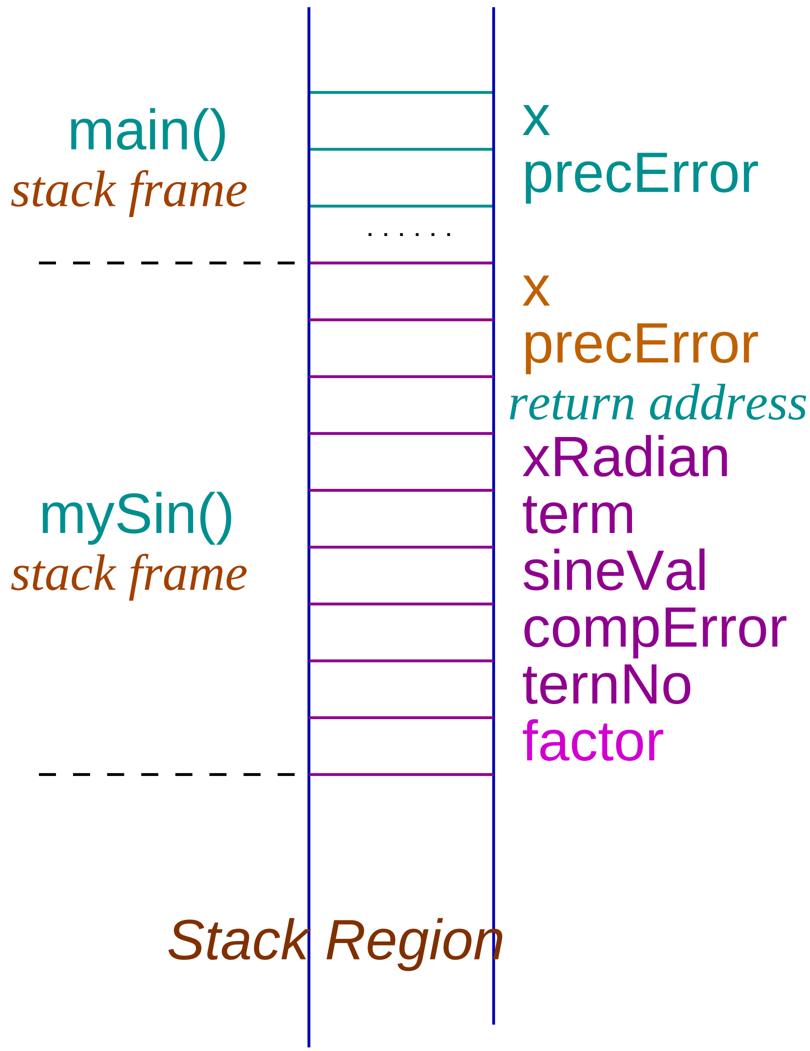
[a] We shall see that a stack is a data type (structure) where entry and exit of data follow LIFO order. This memory region behaves like a stack.

Use of Stack Region

The stack space is used for parameter passing[a], binding local variables to memory, storing return addresses, storing the value returned by the function[b] etc. The stack space used by a function call is known as the activation record or stack frame of the call.

[a] Some systems pass parameters through the CPU registers.

[b] This too can be done through CPU register.

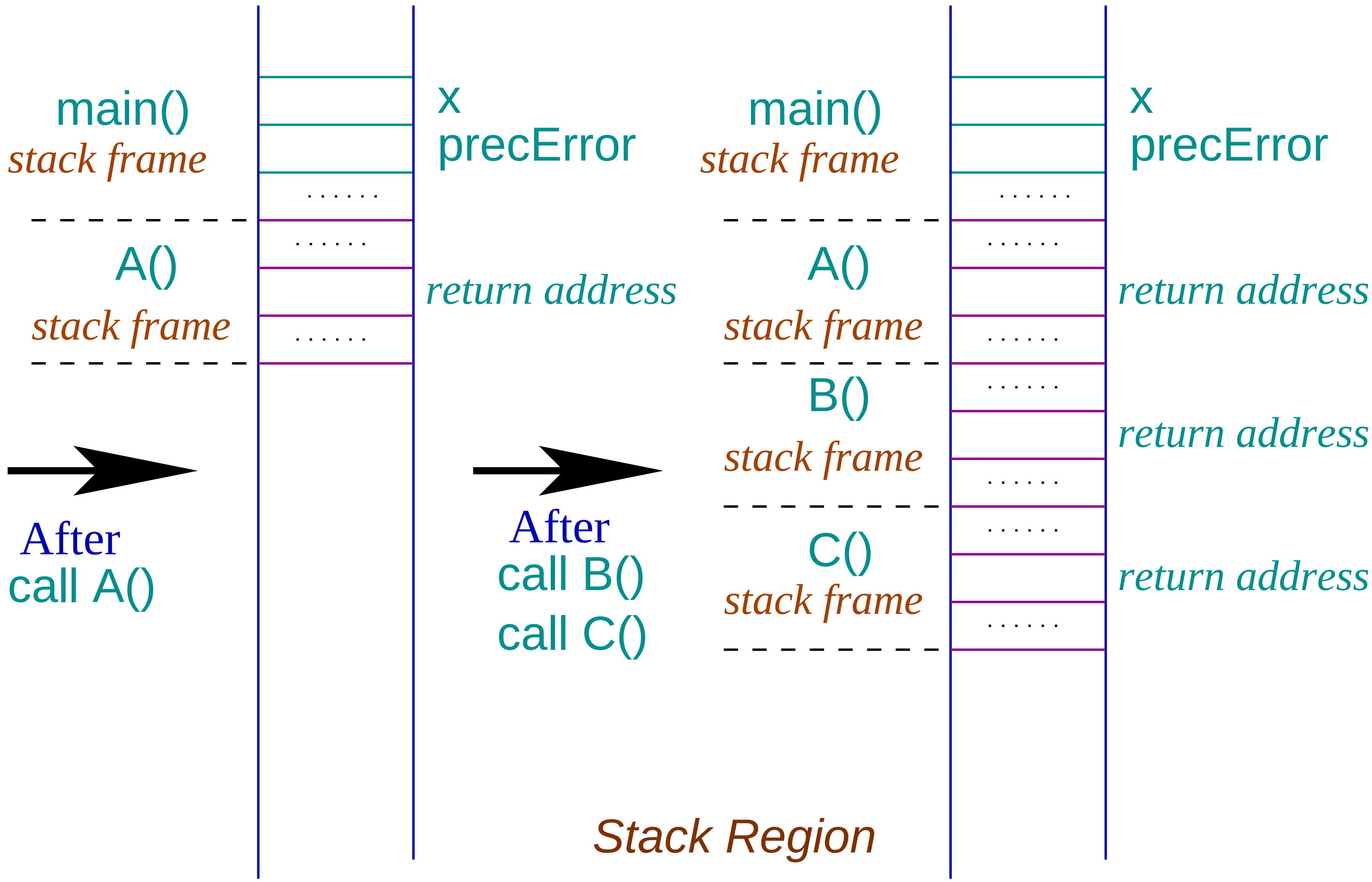

main()
stack frame
x
precError
......
x
precError
return address
xRadian
term
sineVal
compError
ternNo
factor
mySin()
stack frame
Stack Region

Stack Region for Nested Calls

If we consider the following calls,

$$\texttt{main()} \xrightarrow{\texttt{calls}} \texttt{A()} \xrightarrow{\texttt{calls}} \texttt{B()} \xrightarrow{\texttt{calls}} \texttt{C()}$$

the stack frames may look as follows.


main()
stack frame
x
precError
......
......
A()
return address
stack frame
......
After
call A()
main()
stack frame
x
precError
......
......
A()
return address
stack frame
......
......
B()
return address
stack frame
......
......
C()
return address
stack frame
......
After
call B()
call C()
Stack Region

Problem of Output Parameter

It may be necessary for a function to update one or more memory locations other than returning a value.

Consider the call `scanf("%d", &n)`. If the value is read successfully, the function returns one (1), but it also updates the location of `n`. The parameter passing by value in C language creates some problem.

Function `inc()`

We want a function `inc()` that returns the value of its argument and also increments the content of the actual parameter (a variable). The following C code does not work due to call-by value semantics.

```
int inc(int n){
    return n++ ;
}
```

Invocation of inc()

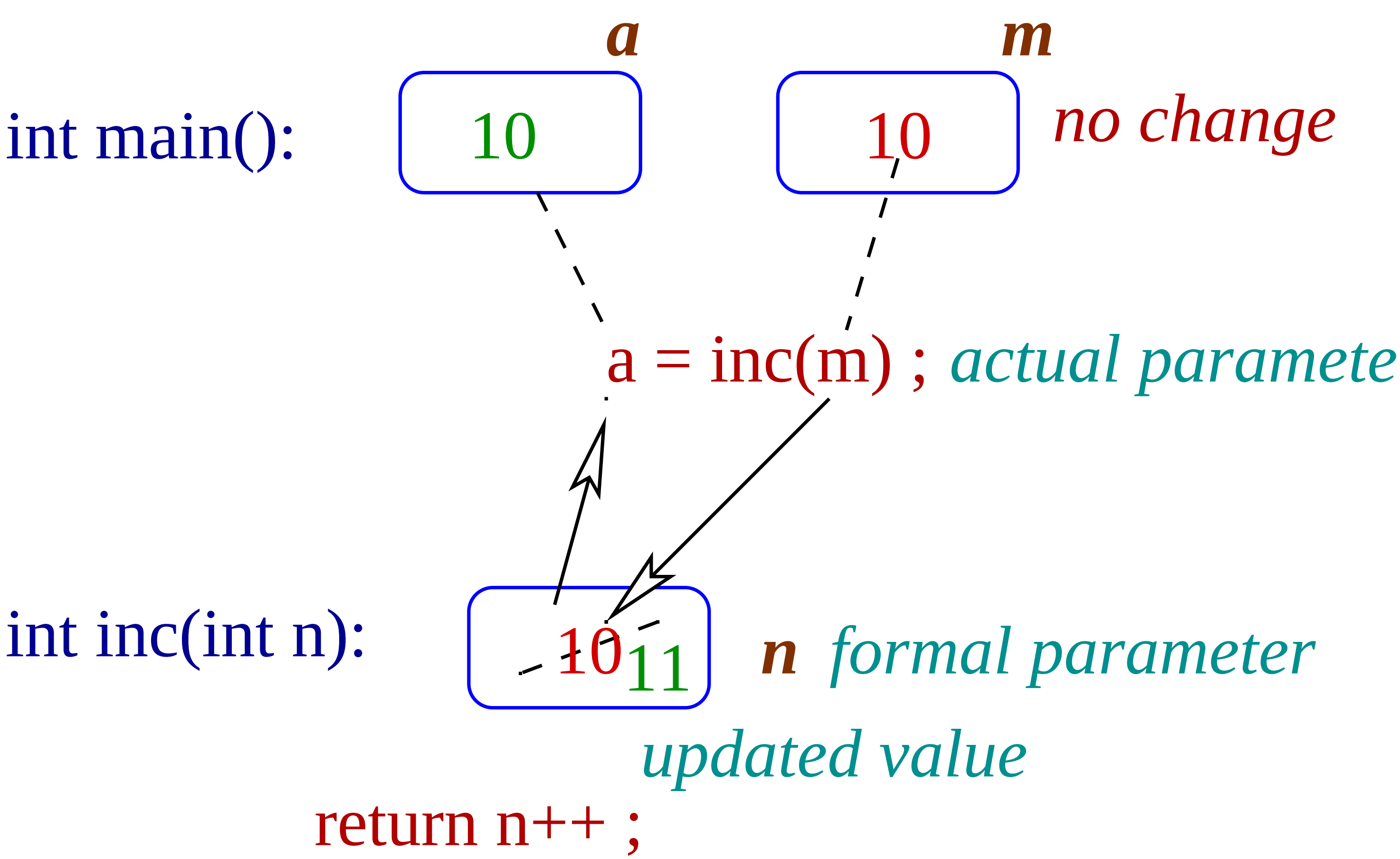
```
#include <stdio.h>
int inc(int n){ // outParam1.c
    return n++ ;
}
int main() {
    int m = 10, a ;
    printf("a: %d\n", a = inc(m)) ;
    printf("m: %d\n", m) ;
    return 0 ;
}
```

Invocation of `inc()`

```
$ cc -Wall outParam1.c
$ ./a.out
a:  10
m:  10
```

Note

- The value of the actual parameter m is copied to the formal parameter n.
- The function inc() returns the content of n and then increments it.
- The return value is assigned to a in main(), but the content of m is unchanged as there is no data copy from the formal to actual parameter.


a
m
int main():
10
10
no change
a = inc(m) ; actual paramete
int inc(int n):
10
11
n formal parameter
updated value
return n++ ;

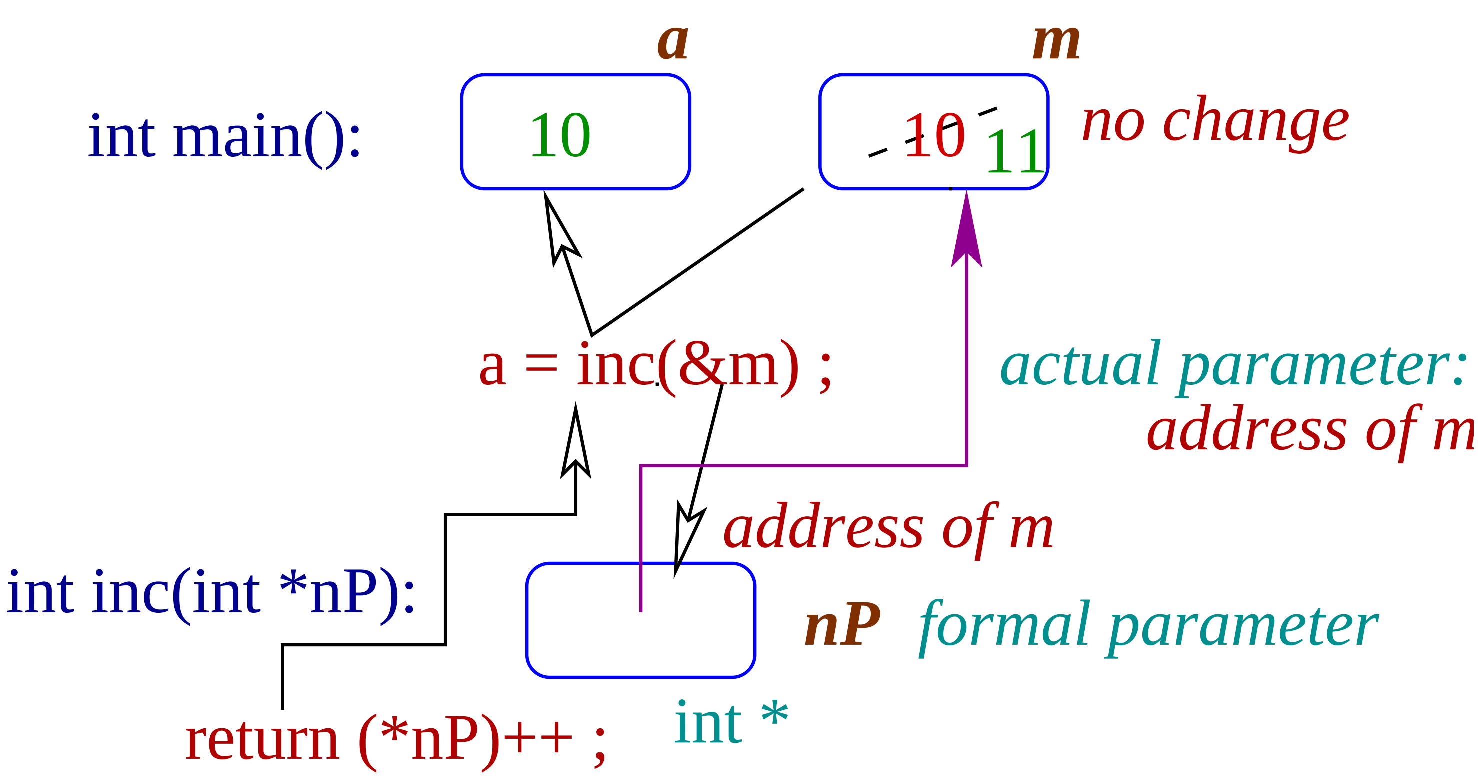
Modified `inc()`

We modify the function as follows:

```
int inc(int *nP){
    return (*nP)++ ;
}
int main() { // outParam2.c
    int m = 10, a ;
    printf("a: %d\n", a = inc(&m)) ;
    printf("m: %d\n", m) ;
    return 0 ;
}
```

Invocation of `inc()`

```
$ cc -Wall outParam2.c
$ ./a.out
a:  10
m:  11
```


a
m
int main():
10
10
11
no change
a = inc(&m) ;
actual parameter:
address of m
address of m
int inc(int *nP):
nP
formal parameter
return (*nP)++ ;
int *

Note

- The actual parameter &m is the address of m copied to the pointer location `int *nP`.
- The return value is the content of the location pointed by `nP` i.e. `*nP`.
- Then the location `*nP` i.e. `m` is incremented.

scanf()

This explains why in the C library function scanf("%d", &n), we have to use the unary operator '&'.

Another Example

The following function is suppose to exchange the content of two memory locations, but it does not work.

```
void exchange(int m, int n){
     int temp = m ;
     m = n ;
     n = temp ;
}
```

Modify to make it work.

exchange()

```
void exchange(int *mP, int *nP){
    int temp = *mP ;
    *mP = *nP ;
    *nP = temp ;
}
```