FUNCTIONS IN C

Topics

1. Functions and Recursion
2. Parameter Passing
    - 2.1 Pass by Value
    - 2.2 Pass by Reference
    - 2.3 Pointers
    - 2.4 Arrays and Strings
    - 2.5 Structures and Arrays of Structures

FUNCTIONS

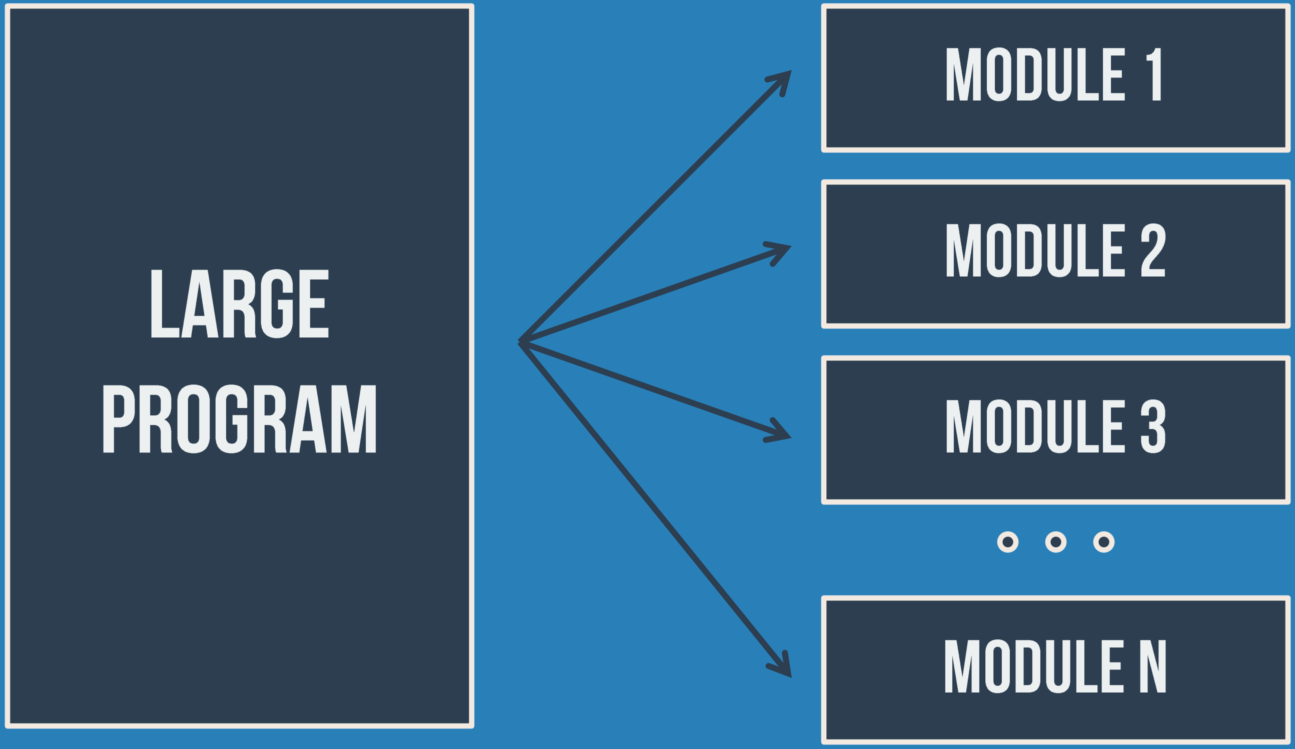
and

RECURSION

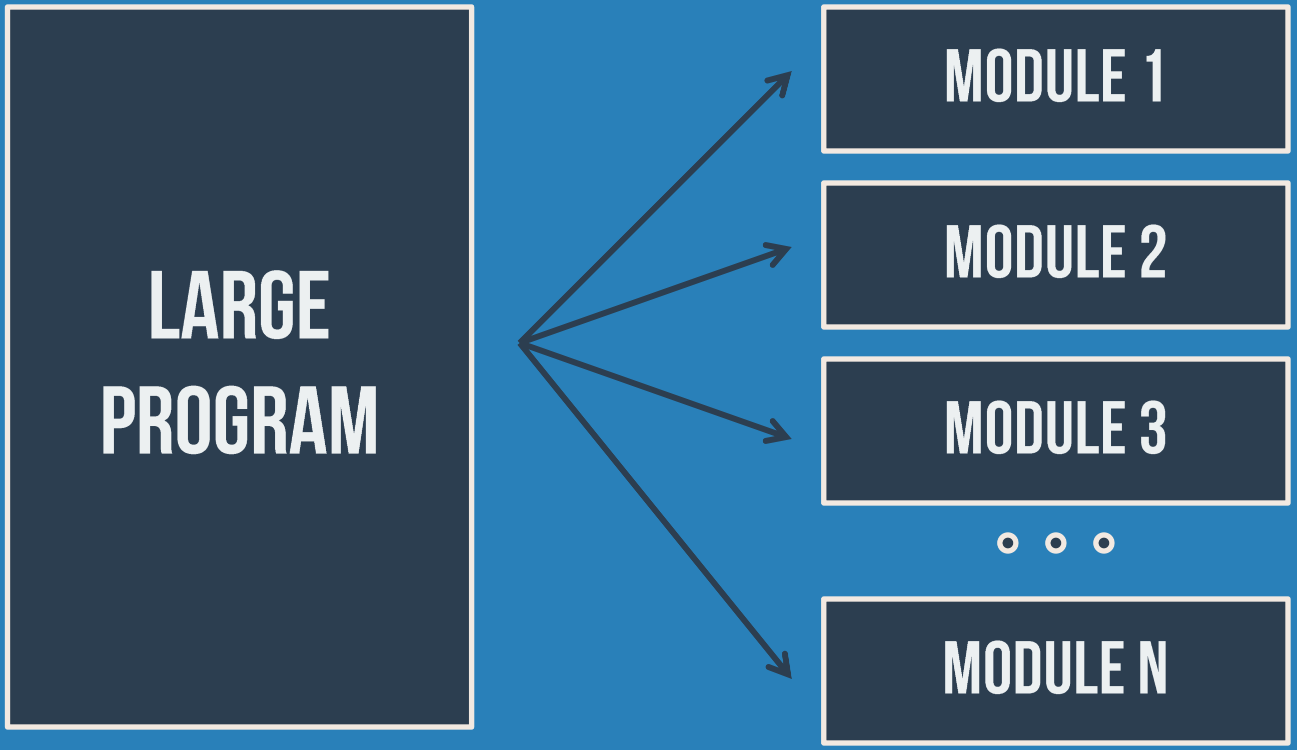
Objective

To learn the fundamentals of functions and recursion.

STRUCTURED PROGRAMMING



We divide the huge problem(program), into smaller parts until each part is solvable.



(And) Each module is a FUNCTION.

FUNCTIONS

Each C program must have at least one function.

```c
int main()
{
    /*
    Program execution starts
    and ends with main()
    */

}
```

```
int main()
{
    /*
    main() can call
    user-defined or
    built-in functions
    */

}
```

```
/* A function can call
    another function. */

void calling_function()
{
    called_function();

}
```

```c
void calling_function()
{
    /* Control transfers from
    the calling function to
    the called function. */
    called_function();

}
```

```c
void calling_function()
{
    called_function();
    /* Control is returned
    when the called function
    finishes its execution. */
}
```

Functions communicate
via
parameter passing.

```
int power(int num, int expo)
{
    int i, result = 1;
    for(i=0; i<expo; i++)
        result *= num;

    return result;
}
```

```
int power(int num, int expo)
{
    int i, result = 1;
    for(i=0; i<expo; i++)
        result *= num;

    return result;
}
```

```
int main()
{
    int num = 3, expo = 2;
    int ans;

    ans = power(num, expo);
}
```

A function can return at most one value.

A called function can cause data changes in the calling function.

Function Declarations

Functions need to be declared before they are defined.

```
//function prototype
int power(int num, int expo);

int main()
{
    …
}
```

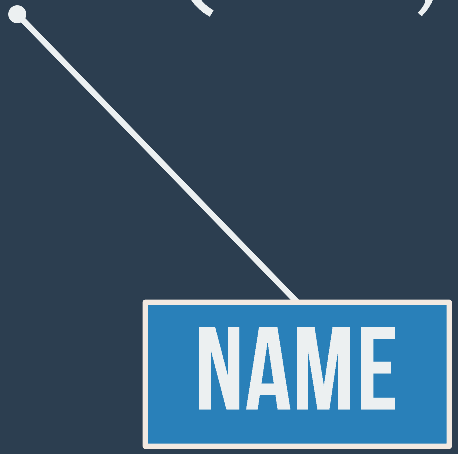
```
//function prototype
int power(int, int);

int main()
{
    …
}
```

```
//function prototype
int power(int, int);
```

RETURN TYPE

```
//function prototype
int power(int, int);
```



NAME

```
//function prototype
int power(int, int);
```

PARAMETERS

```
//declarations
int power(int, int);
void foo(char, int, float);

int main()
{
   …
}
```

Function Definitions

Function header

+

Function body

Function header

```
int power(int num, int expo)
{
      …
}
```

```
int power(int num, int expo)
{
…
}
```

CAN BE ANY DATA TYPE / VOID

```c
int power(int num, int expo)
{
…
}
```

ANY VALID IDENTIFIER IN C

```
int power(int num, int expo)
{
…
}
```

ARE ALSO LOCAL VARIABLES

```
//No parameters
int foo()
{
    …
}
```

```
//No parameters
int foo(void)
{
    …
}
```

Function body

```
int power(int num, int expo)
{
    …
}
```

```
int power(int num, int expo)
{
    …
    //return statement
    return x;
}
```

```
int main(int argc, char* argv[])
{


    return 0;
}
```

Local Variables

Local variables are variables within a function.

Function parameters are also local variables.

```
int main()
{
/* These are allocated when a
function starts execution… */

    float pi;
    int radius;


}
```

```
int main()
{
    float pi;
    int radius;
/* … and destroyed automatically
as the function terminates.*/
}
```

Can only be accessed within the function

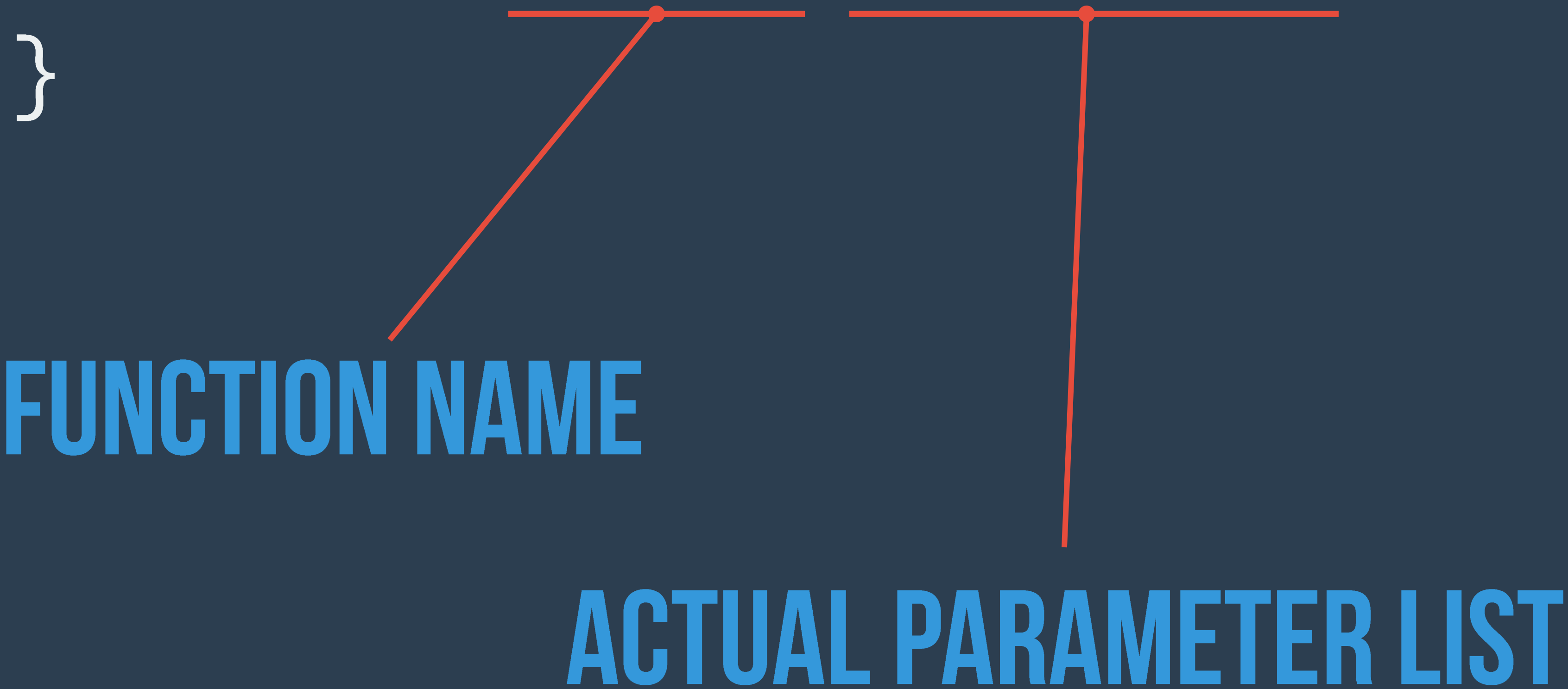
Function Call

Function name

+

Actual parameter list

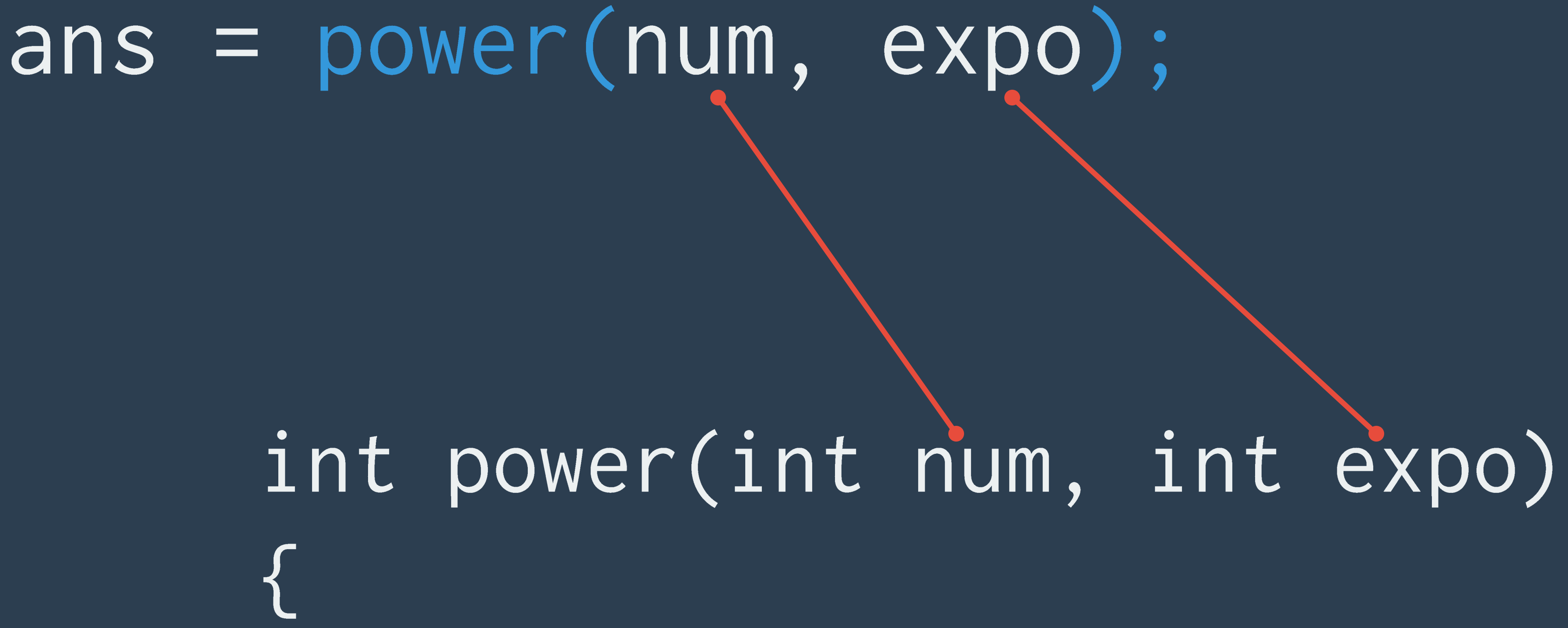
```
int main()
{
    …
    ans = power(num, expo);
}
```



FUNCTION NAME

ACTUAL PARAMETER LIST

Function name should exist within or be included in the program.



```
/* APL must correspond to the
formal parameter list.  */
ans = power(num, expo);



     int power(int num, int expo)
     {
```

How do functions communicate?

Parameter passing

+

Use of return values

PARAMETER PASSING

Passing of data as parameters to functions.

PARAMETER PASSING

Pass-by-value /
Pass-by-reference

USE OF RETURN VALUES

Functions may return results of computations.

USE OF RETURN VALUES

A function can return at most one value.

Pass-by-value

Only the actual value of the variable is passed.

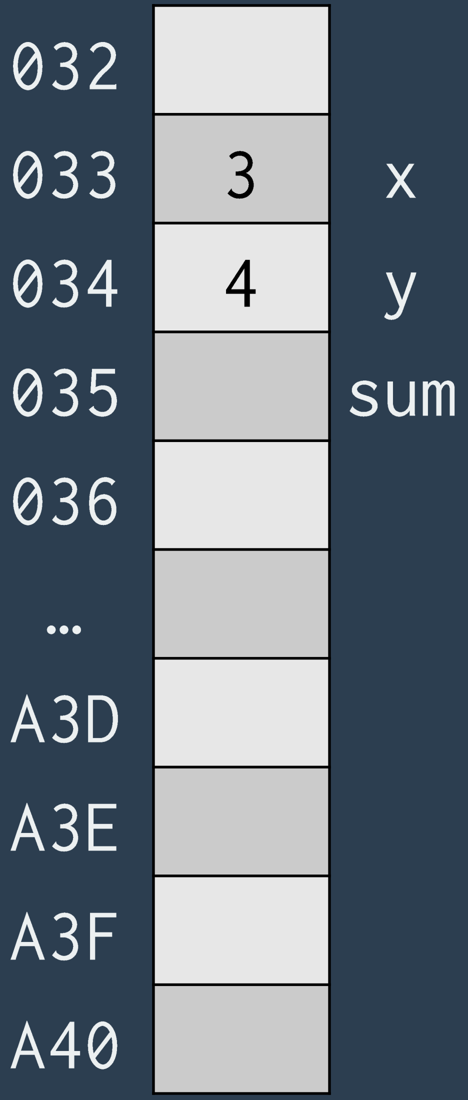
The formal parameters of the called function obtains these values.

```c
#include<stdio.h>

int getSum(int, int);

int main() {
    int x=3, y=4, sum;
    sum = getSum(x, y);
}

int getSum(int a, int b){
   int sum;
   sum = a + b;
   return sum;
}
```



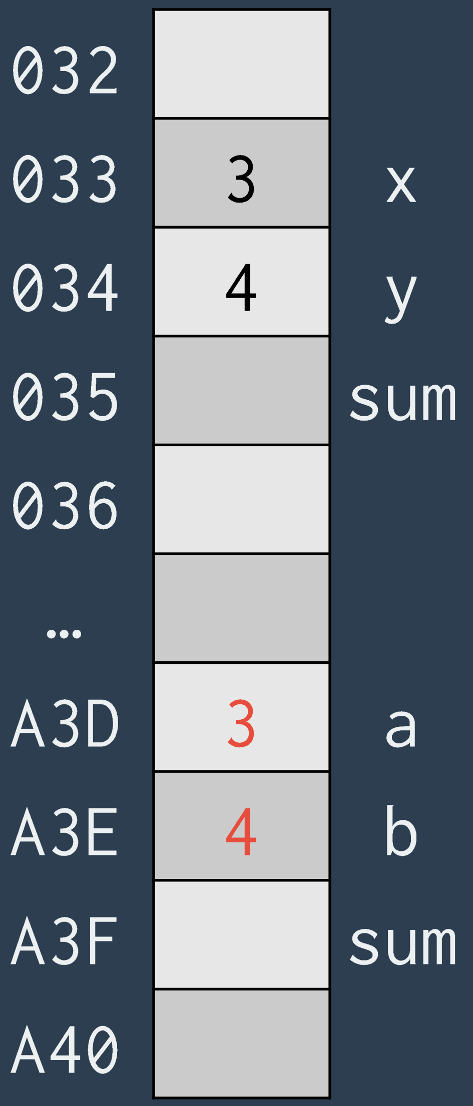
032		
033	3	x
034	4	y
035		sum
036		
…		
A3D		
A3E		
A3F		
A40		

```c
#include<stdio.h>

int getSum(int, int);

int main() {
    int x=3, y=4, sum;
    sum = getSum(x, y);
}

int getSum(int a, int b){
   int sum;
   sum = a + b;
   return sum;
}
```



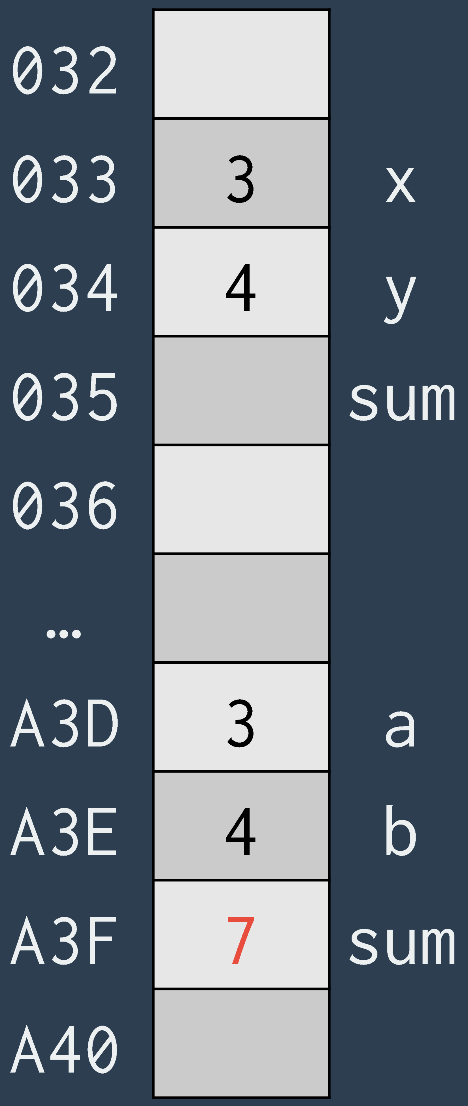
032		
033	3	x
034	4	y
035		sum
036		
…		
A3D	3	a
A3E	4	b
A3F		sum
A40		

```c
#include<stdio.h>

int getSum(int, int);

int main() {
    int x=3, y=4, sum;
    sum = getSum(x, y);
}

int getSum(int a, int b){
   int sum;
   sum = a + b;
   return sum;
}
```



032		
033	3	x
034	4	y
035		sum
036		
…		
A3D	3	a
A3E	4	b
A3F	7	sum
A40		

```c
#include<stdio.h>

int getSum(int, int);

int main() {
   int x=3, y=4, sum;
   sum = getSum(x, y);
}

int getSum(int a, int b){
  int sum;
  sum = a + b;
  return sum;
}
```

032		
033	3	x
034	4	y
035	7	sum
036		
…		
A3D	3	a
A3E	4	b
A3F	7	sum
A40		

POINTERS

A pointer is a variable that stores the address of another variable.

Declared as:

```
<data type> * <var_name>;
```

```
int * p;
float *q;
```

Associated with two unary operators:

&

the address operator

*

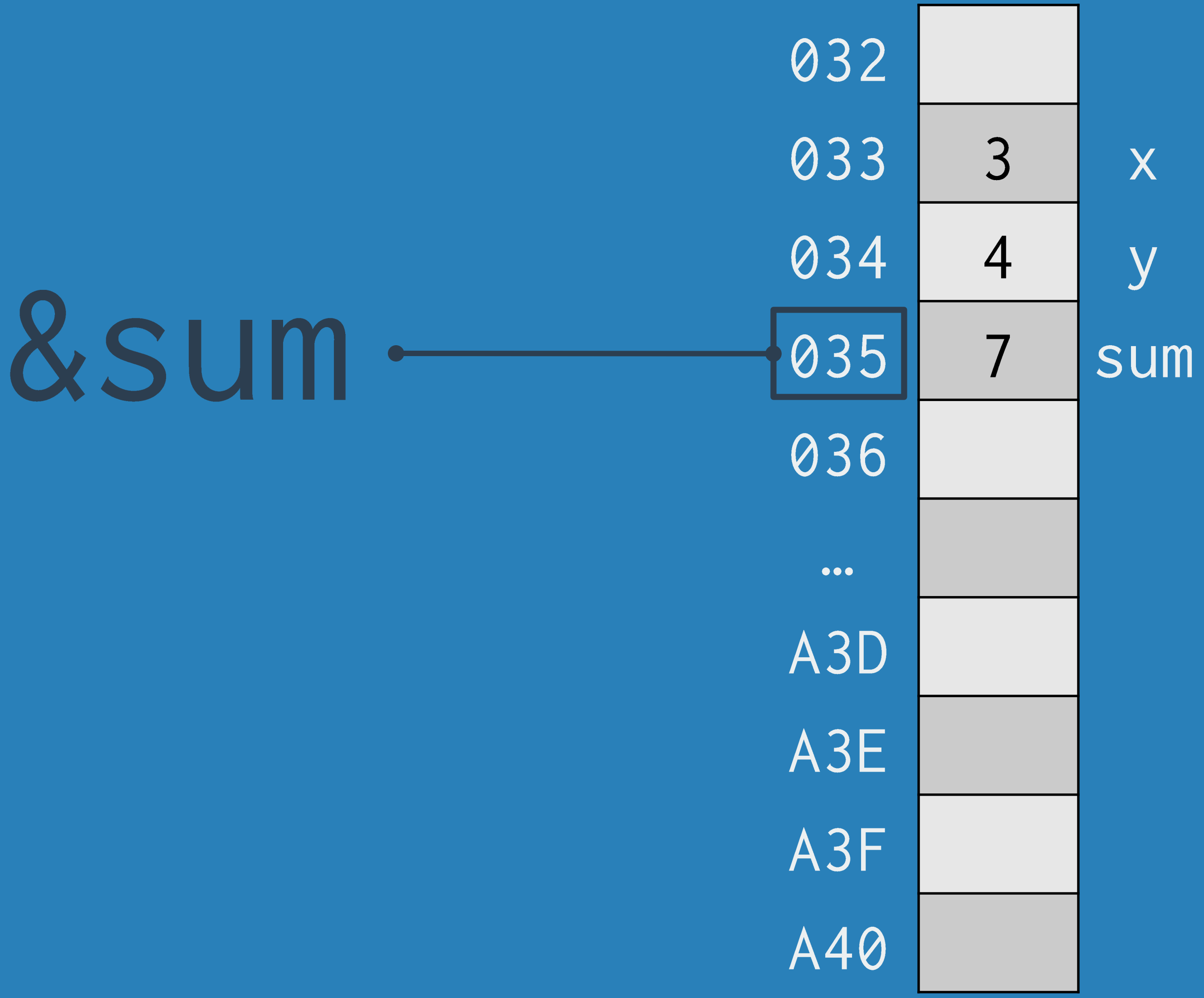
the indirection operator

&

the address operator

```
int main() {
   int x=3, y=4, sum;
   sum = getSum(x, y, &sum);
}
```

&sum is read as “the address of variable sum”



&sum

032		
033	3	x
034	4	y
035	7	sum
036		
…		
A3D		
A3E		
A3F		
A40		

*

the indirection operator

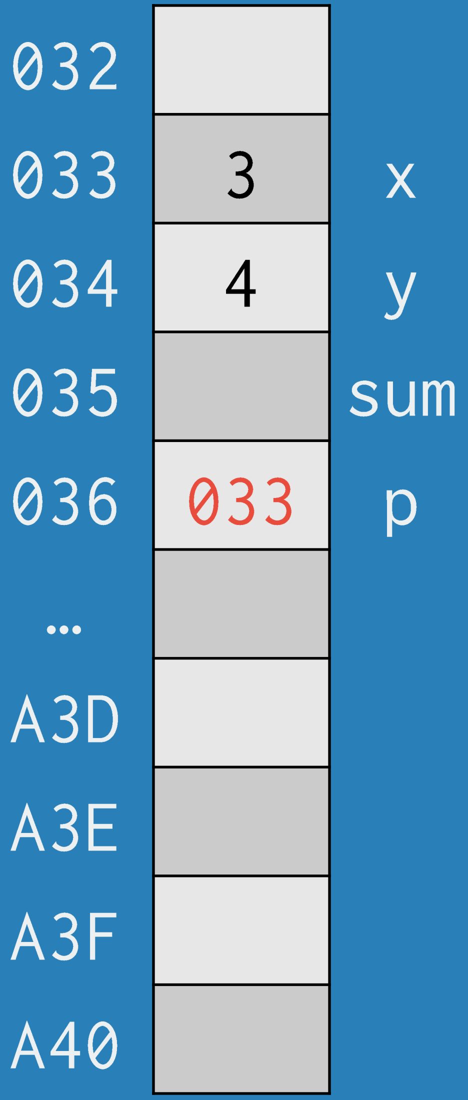
```
int main() {
   int x=3, y=4, sum;
   int *p;
   p = &x;
   sum = y + (*p);
}
```

*p is read as
“the value/variable at
the address held by p”

```
int main() {
   int x=3, y=4, sum;
   int *p;
   p = &x;
   sum = y + (*p);
}
```

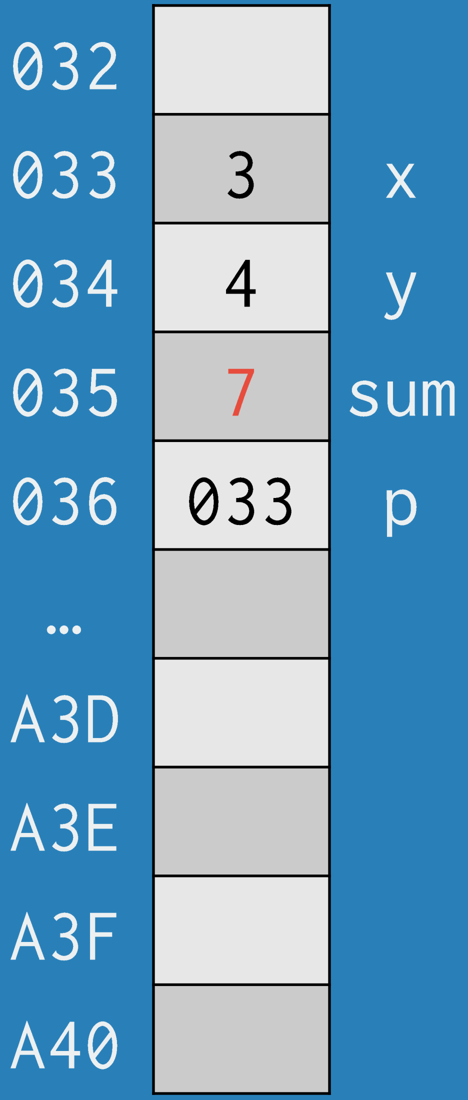
032		
033	3	x
034	4	y
035		sum
036		p
…		
A3D		
A3E		
A3F		
A40		

```c
int main() {
  int x=3, y=4, sum;
  int *p;
  p = &x;
  sum = y + (*p);
}
```



032		
033	3	x
034	4	y
035		sum
036	033	p
…		
A3D		
A3E		
A3F		
A40		

```
int main() {
  int x=3, y=4, sum;
  int *p;
  p = &x;
  sum = y + (*p);
}
```



032		
033	3	x
034	4	y
035	7	sum
036	033	p
…		
A3D		
A3E		
A3F		
A40		

Pass-by-reference

The reference to the variable is passed to the function.

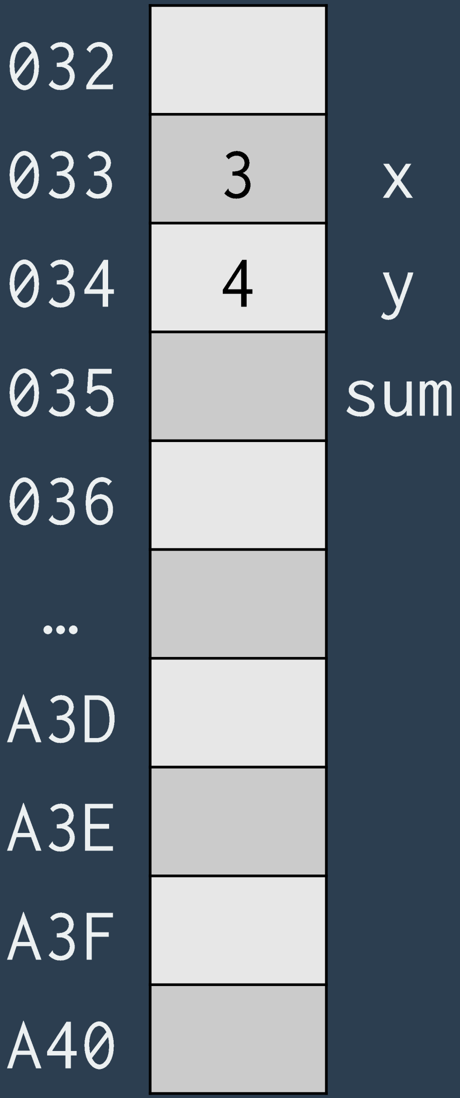
reference == address

```c
#include<stdio.h>

void getSum(int, int, int *);

int main() {
    int x=3, y=4, sum;
    getSum(x, y, &sum);
}

void getSum
(int a, int b, int *sum){
  *sum = a + b;
}
```



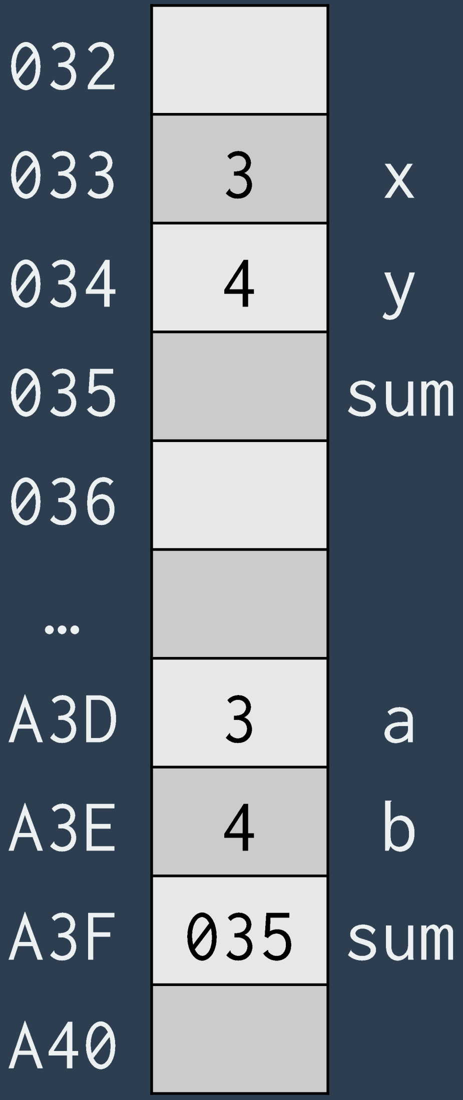
032		
033	3	x
034	4	y
035		sum
036		
…		
A3D		
A3E		
A3F		
A40		

```c
#include<stdio.h>

void getSum(int, int, int *);

int main() {
   int x=3, y=4, sum;
   getSum(x, y, &sum);
}

void getSum
(int a, int b, int *sum){
  *sum = a + b;
}
```



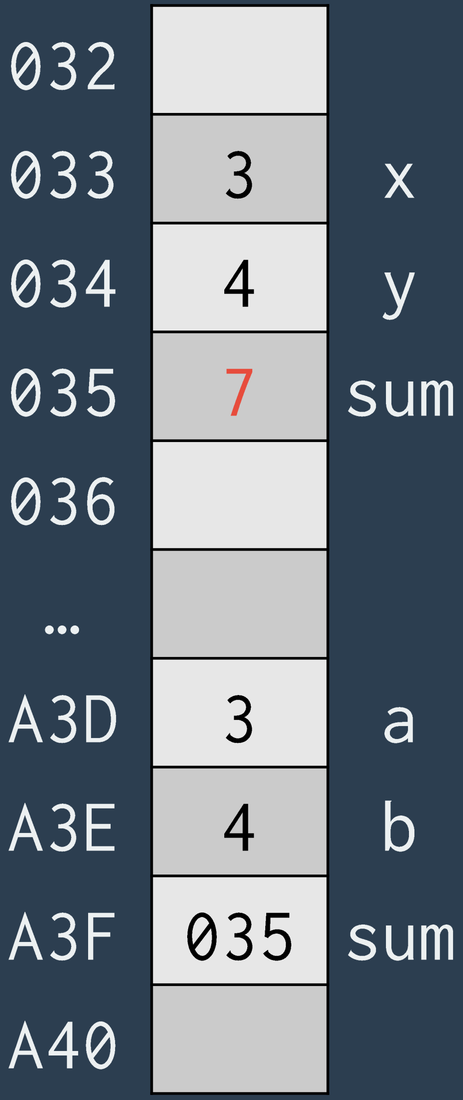
032		
033	3	x
034	4	y
035		sum
036		
…		
A3D	3	a
A3E	4	b
A3F	035	sum
A40		

```c
#include<stdio.h>

void getSum(int, int, int *);

int main() {
   int x=3, y=4, sum;
   getSum(x, y, &sum);
}

void getSum
(int a, int b, int *sum){
  *sum = a + b;
}
```



032		
033	3	x
034	4	y
035	7	sum
036		
…		
A3D	3	a
A3E	4	b
A3F	035	sum
A40		

RECURSION

A recursive function is a function that calls itself.

Base case

+

Recursive call/case

$$\sum_{i=0}^{n} i$$

n=6

0 1 3 6 10 15 21

```
//summation of numbers 0 to n
int summ(int n)
{
    if (n == 1)
        return 1;

    return summ(n-1) + n;
}
```

The Base Case

```
int summ(int n)
{
    if (n == 1)
        return 1;

    return summ(n-1) + n;
}
```

aka the

stopping condition

Usually returns a constant.

```
int summ(int n)
{
    /* What if there is no
       base case? */

    return summ(n-1) + n;
}
```

The Recursive Call

aka the

general/recursive

case

```
int summ(int n)
{
    if (n == 1)
        return 1;

    return summ(n-1) + n;
}
```

Making progress towards a base case by reducing the problem.

Defining recursive functions

$$\sum_{i=0}^{n} i$$

```
summ(n) =
  n + (n-1) + (n-2) + …
                    + 2 + 1 + 0
```

*where n >= 0

Base case:

```
summ(0) = 0, if n=0
```

Recursive call:

```
summ(n) = n + summ(n-1),
                  if (n >= 0)
```

```
//summation of numbers 0 to n
int summ(int n)
{
    if (n == 0)
        return 0;

    return summ(n-1) + n;
}
```

Limitations of recursion

Extensive overhead due to numerous function calls.

A called function requires memory.

Can you turn a recursive function into an iterative one?

```
//dynamic programming version
int sum=0, i, n;
 …
for(i=1; i<n; i++)
{
    sum += i;
}
```