{C} Programming

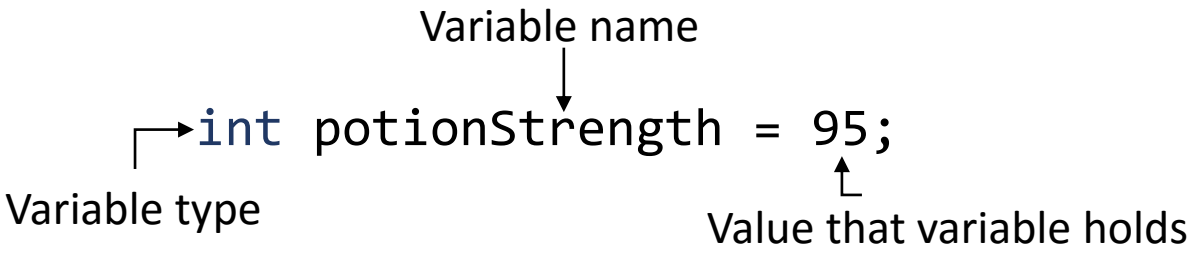
Part 1/2 | Basics

Variables, Conditions, Loops, Arrays, Pointer basics

Variables

A variable is a container (storage area) to hold data.

Eg.



C is strongly typed language. What it means is that the **type of a variable cannot be changed**.

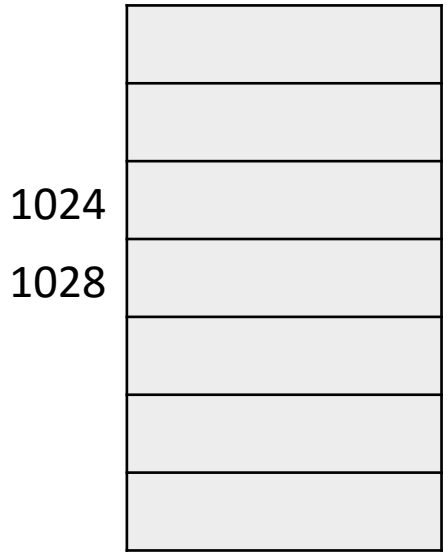
You can use const prefix to declare constant values with specific type:

```
const double PI = 3.14;
```

PI is a constant. That means, that in this program 3.14 and PI is same.

Variables | Example

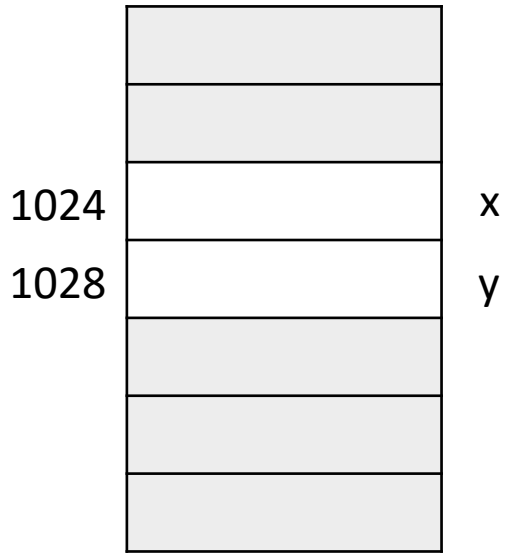
```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

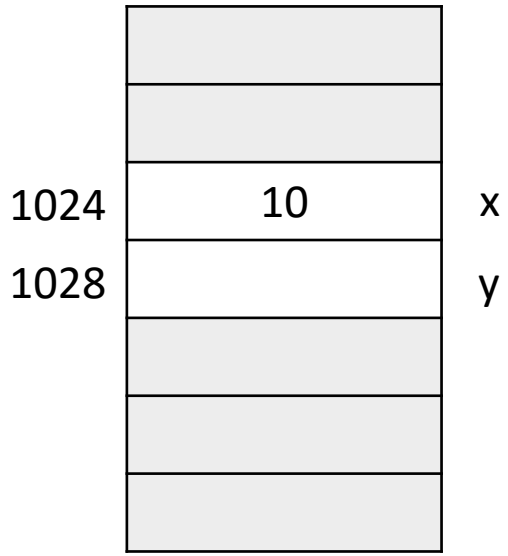
Declare x and y



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

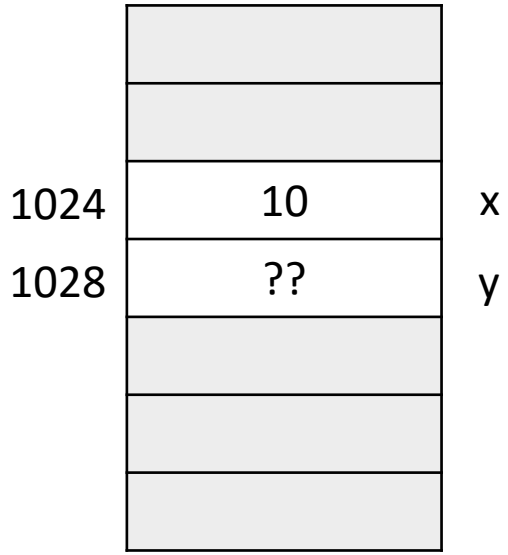
Assign to x the value 10



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

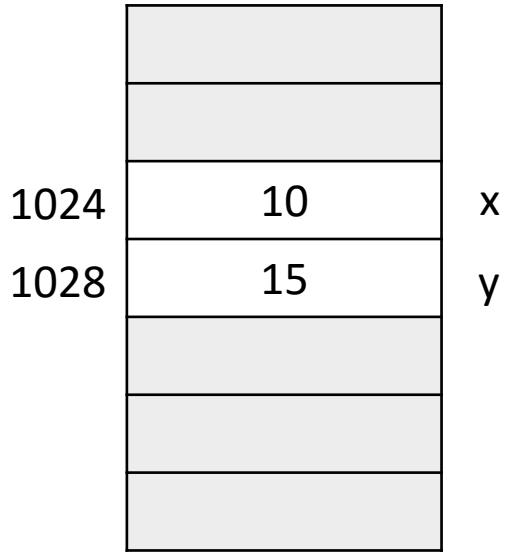
Assign to y the Value x+5



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

Program's output:
x = 10 y = 15



Variable sizes

Type	Size	Comment
integer	4 bytes = 2^32 bits	it can take 2^{32} distinct states as: -2^{31},-2^{31}+1, ...,-2, -1, 0, 1, 2, ..., 2^{31}-2, 2^{31}-1
float	4 bytes	Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.
double	8 bytes	
char	1 byte	
struct	Depends on the structs' fields	

Variable sizes | sizeof();

```c
#include <stdio.h>
int main(void){
        int a, e[10];
        float b;
        double c;
        char d;
        printf("Size of int=%lu bytes\n",sizeof(a));
        printf("Size of float=%lu bytes\n",sizeof(b));
        printf("Size of double=%lu bytes\n",sizeof(c));
        printf("Size of char=%lu byte\n",sizeof(d));
        printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
        return 0;
}
```

What should the code above print?

Variable sizes | sizeof();

```c
#include <stdio.h>
int main(void){
        int a, e[10];
        float b;
        double c;
        char d;
        printf("Size of int=%lu bytes\n",sizeof(a));
        printf("Size of float=%lu bytes\n",sizeof(b));
        printf("Size of double=%lu bytes\n",sizeof(c));
        printf("Size of char=%lu byte\n",sizeof(d));
        printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
        return 0;
}
```

Program Output:

```
Size of int=4 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte
Size of integer type array having 10 elements = 40 bytes
```

If statements

The if keyword introduce the concept of handle some predictable yet unknown event:

Eg. In a pc game, what a character will find If he opens some specific door.

A **true** statement is one that evaluates to a **nonzero** number. A **false** statement evaluates to **zero**.

For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1.

Boolean operators: !(not), &&(and), ||(or)

Relational operators: >, <, >=, <=, ==, !=

A. !(1 || 0)	ANSWER: 0
B. !(1 || 1 && 0)	ANSWER: 0 (AND is evaluated before OR)
C. !((1 || 0) && 0)	ANSWER: 1
D. 5>4	ANSWER: 1
E. (5==5)&&(!(5 != 4))	ANSWER: 0

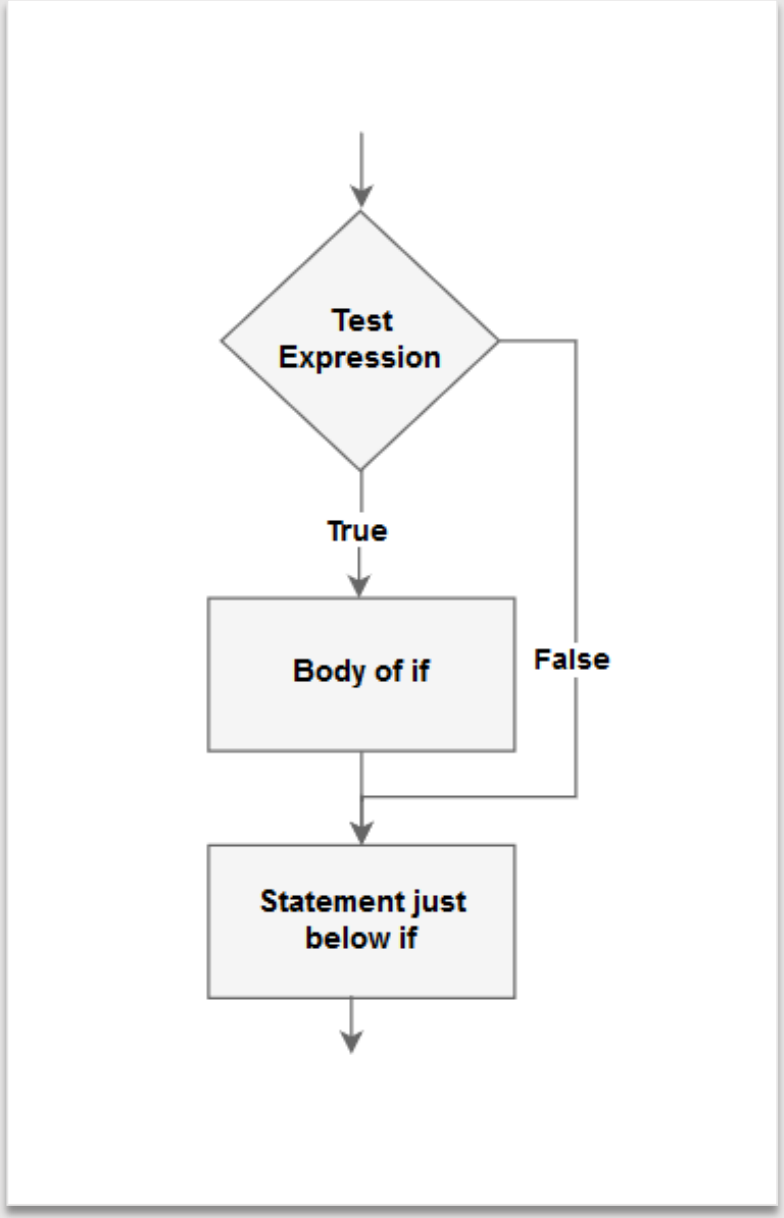
Conditions | if

```
if (statement is TRUE){
        Execute all statements inside if body
}
```

The if statement evaluates the expression inside parenthesis.

If test expression is evaluated to true (nonzero), statements inside the body of if are executed.

If test expression is evaluated to false (0), statements inside the body of if are skipped.



If | Example

```c
#include <stdio.h>
int main() {
	int number;
	printf("I am your computer genie!\n");
	printf("Enter a number from 0 to 9:");
	scanf("%d", &number);
	if(number<5){
		printf("That number is less than 5!\n");
	}
	printf("The genie knows all, sees all!\n");
	return(0);
}
```

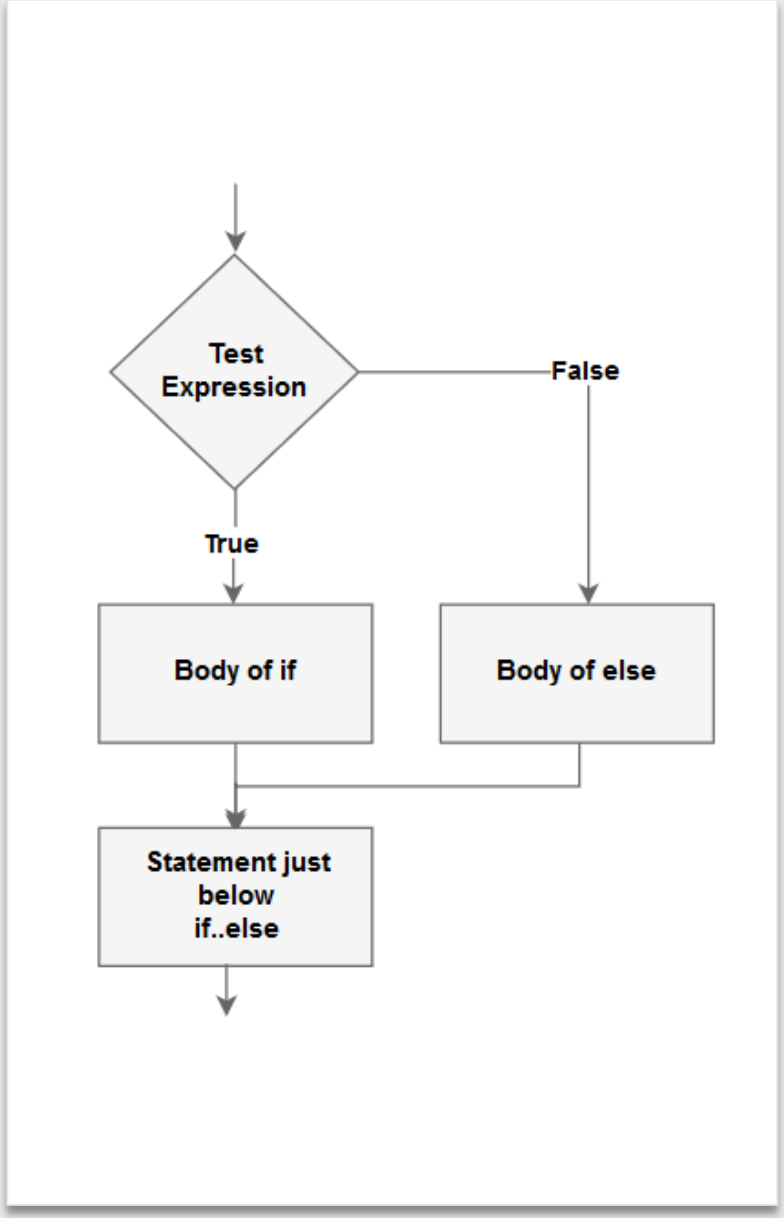
Conditions | if…else

```
if (statement is TRUE){
        statements inside the body of if
}else{
        statements inside the body of else
}
```

The if statement evaluates the test expression inside parenthesis.

If test expression is evaluated to true (nonzero), statements inside the body of if are executed and the statements inside the body of else are skipped.

If test expression is evaluated to false (0), statements inside the body of if are skipped and statements inside the body of if are executed.



If…else | Example

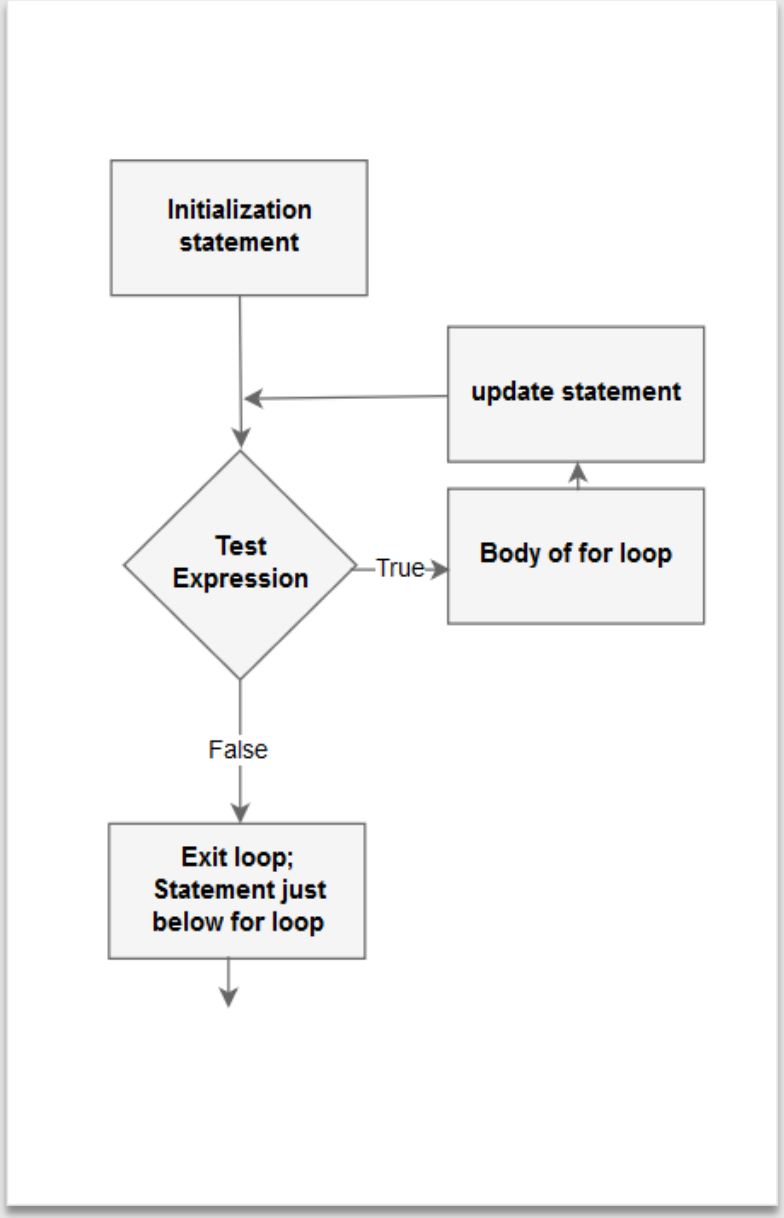
```
#include <stdio.h>
int main(){
        char c;
        printf("Would you like your computer to explode?");
        c=getchar();
        if(c=='Y' || c=='y'){
                printf("OK: Configuring computer to explode now.\n");
                printf("Bye!\n");
        }else{
                printf("Okay. Whew!\n");
        }
        return(0);
}
```

Loops | for

```
for(initStmt; condition; updateStmt){
        stmts to execute while the condition is true;
}
```

The initialization statement is executed once. Then the condition is evaluated. If the condition is true, the code inside of for loop is executed and the update expression is updated.

This process repeats until condition is false.



for | Example

```c
#include <stdio.h>
int main(void) {
     int x;
     for ( x = 0; x < 10; x++ ){
          printf( "%d\n", x );
     }
     return 0;
}
```

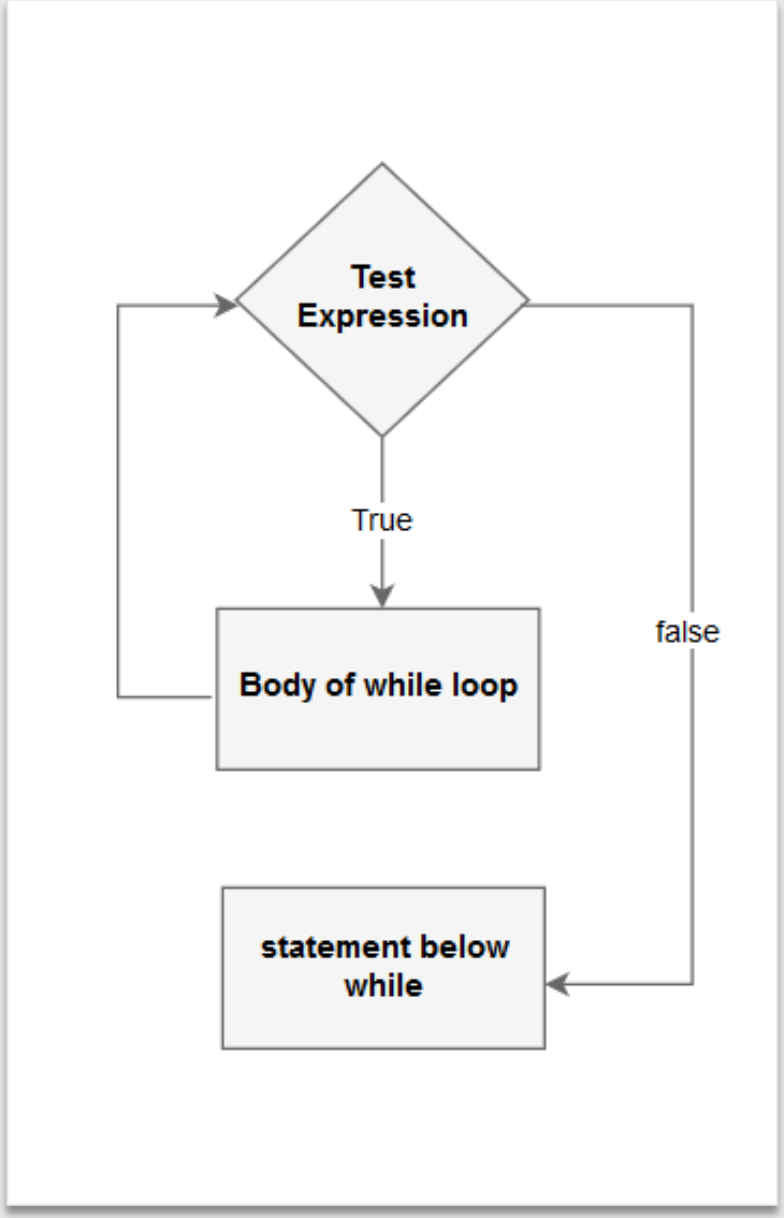
Loops | while

```
while(condition){
	Code to execute while the condition is true
}
```

If the condition is true, the body of the while is executed.

This process repeats until condition is false.

What’s the difference of do..while(); ?

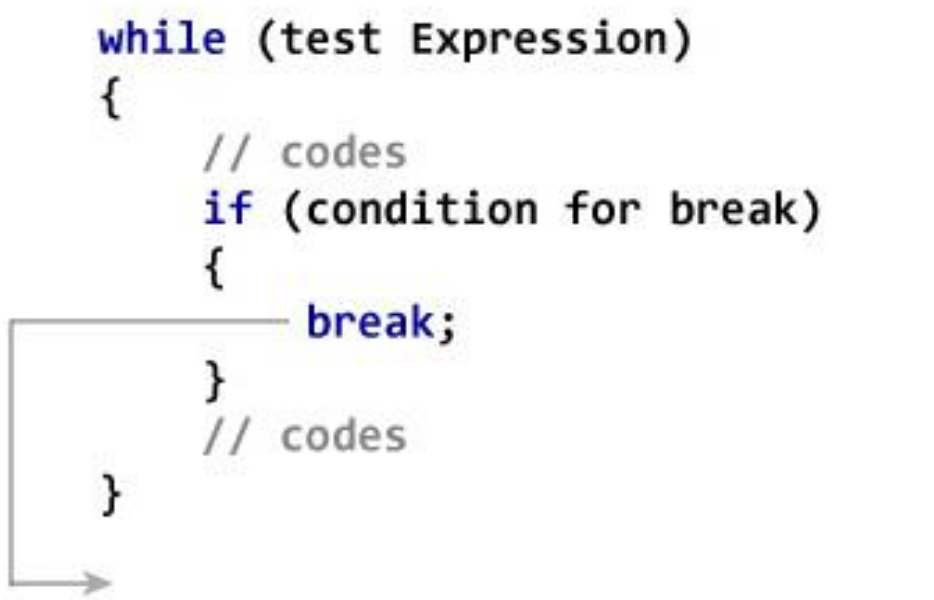


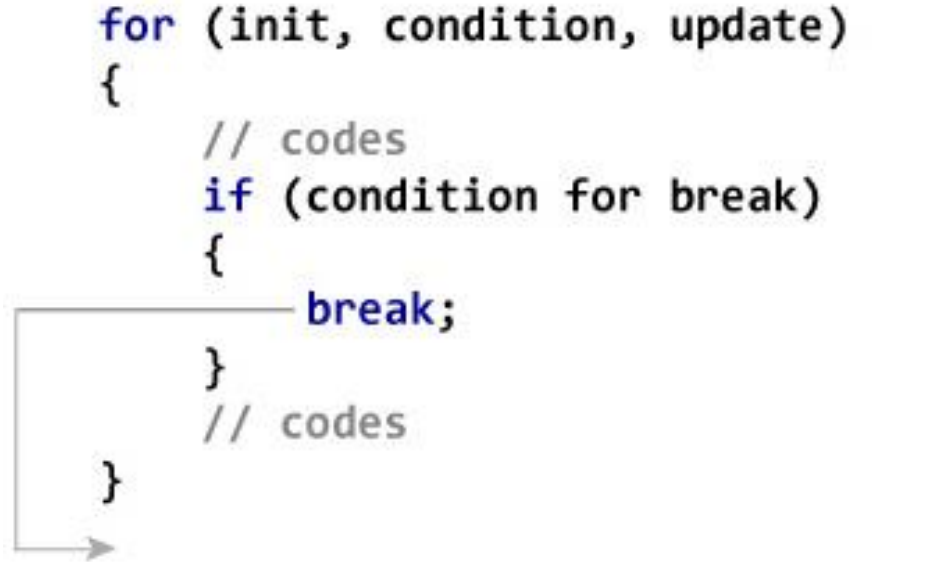
Break & continue I

The break statement **terminates the loop**.

```
for(i=1; i <= 10; ++i){
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);
        if(number < 0.0) {
                break;
        }
}
```

- The for loop stops when a negative number is given by the user





Break & continue II

The continue statement **skips** some statements inside the loop.

```
for(i=1; i <= 10; ++i){
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);
        if(number < 0.0) {
                continue;
        }
        sum+= number;
}
```

- The negative values will not be added to the sum. Why?

```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Switch

```
switch (n)
{
    case constant1:
        // code to be executed if n is equal to constant1;
        break;

    case constant2:
        // code to be executed if n is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if n doesn't match any constant
}
```

Switch | example: main.c (hy240b)

```
switch (event)
{
    case 'A' :
         add_new_movie(movieID,category, year);
         break;
    case ‘R' :
        rate_movie(userID, movieID, score);
        break;
        .
        //other events
        .
    default:
        printf(“Not recognizable event”);
        break;
}
```

Functions

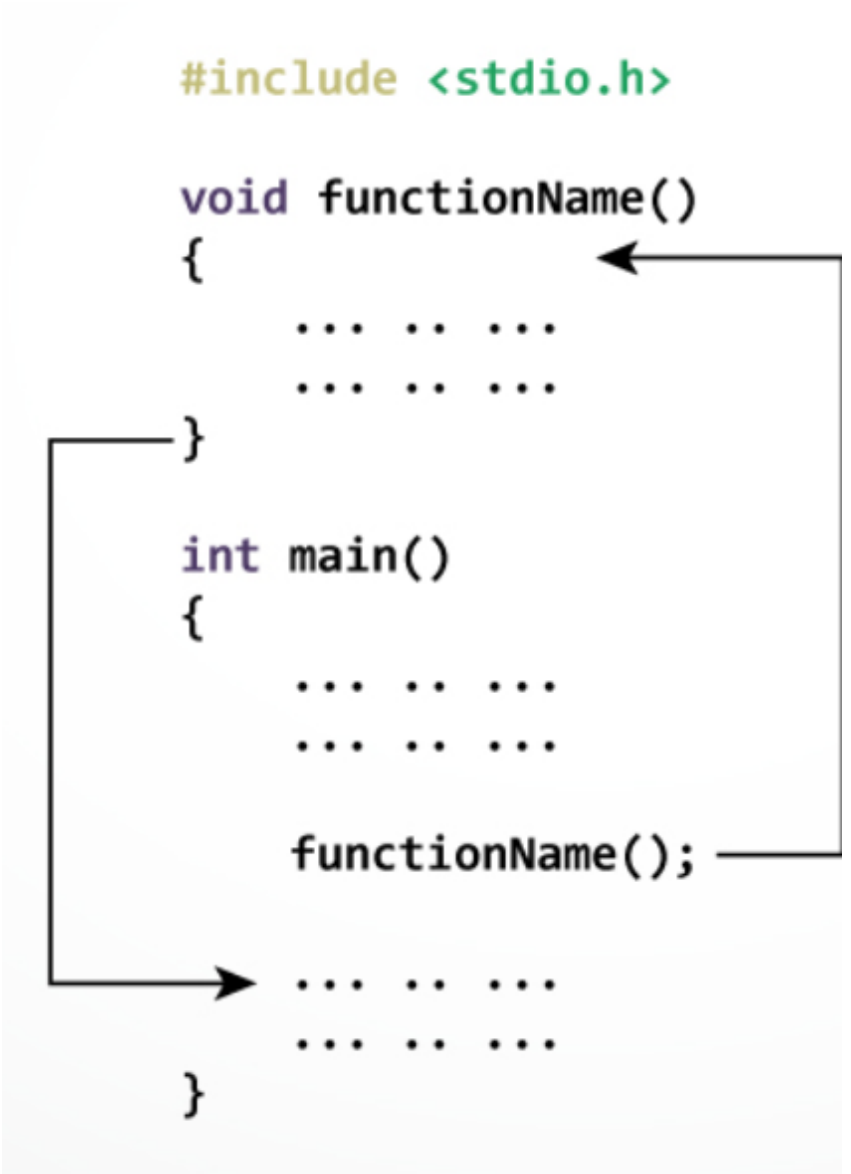
A function is a block of code that performs a specific task

There are two types of functions:

- **Standard library functions:** built-in functions to handle some tasks. Some of them are: printf, scanf etc
- **User-defined functions:** Functions defined by user to make the program easier to understand, reuse the same code or devide the program in smaller modules.

- Syntax:

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
}
```



Functions | example

```c
#include <stdio.h>
int mult ( int x, int y );

int main(void){
        int x = 3;
        int y = 5;
        printf( "The product of %d and %d is %d\n", x, y, mult( x, y ) );
        return 0;
}

int mult (int x, int y){
        return x * y;
}
```

Recursion

if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

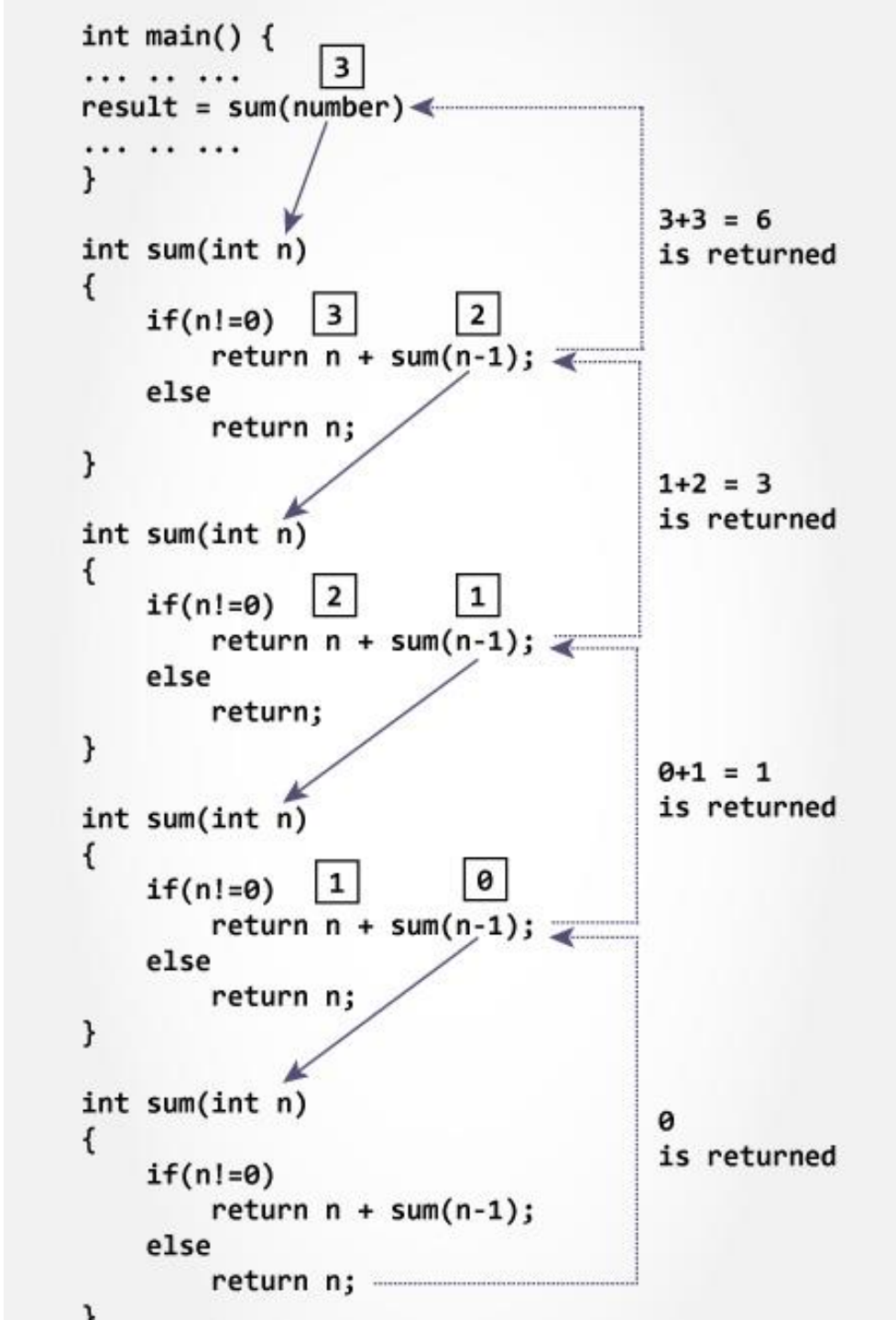
```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement can be used where one branch makes the recursive call and other doesn't.

Recursion | Example

```c
#include <stdio.h>
int sum(int n);
int main(){
        int number, result;
        printf("Enter a positive integer: ");
        scanf("%d", &number);
        result = sum(number);
        printf("sum=%d", result);
}
int sum(int n){
        if (n!=0)
                return n + sum(n-1);
        else
                return n;
}
```



Recursion | factorial

```c
#include <stdio.h>
int factorial(unsigned int i) {
	if(i <= 1) {
		return 1;
	}
	return i * factorial(i - 1);
}

int main() {
	int i = 3;
	printf("Factorial of %d is %d\n", i, factorial(i));
	return 0;
}
```

Arrays | 1D

Arrays is a kind of data structure that can store a **fixed-size sequential collection** of elements of the **same type.**

```
data_type array_name[array_size];
```

Eg. float temperature_in_crete[10]; , int age[5]; , char letters[24];

You can initialize an array one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

If you omit the size of the array, an array just big enough to hold the initialization is created.

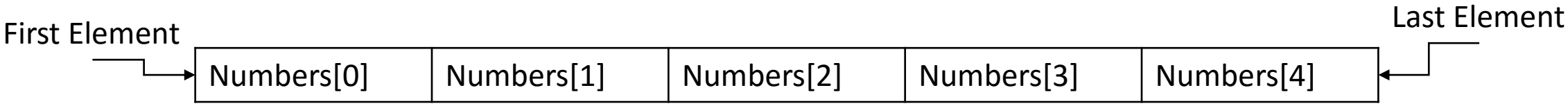
```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Size of array defines the number of elements in an array.

If you have you declare an array int numbers[10]; you can use the array members from numbers[0] to numbers[9]

Arrays | 1D

int Numbers[5];



Suppose, the starting address of Numbers[0] is 1020 and the size of int be 4 bytes. Then, the next address (address of Numbers[1]) will be 1024, address of Numbers[2] will be 1028 and so on.

1020	Numbers[0]
1024	Numbers[1]
1028	Numbers[2]
1032	Numbers[3]
1036	Numbers[4]

Multidimensional Arrays

C programming language allows programmer to create arrays of arrays known as multidimensional arrays.

```
data_type array_name[size_1][size_2]…[sizeN];
```

For example, int c[2][3]={{1,3,0}, {-1,5,9}};

```
#include <stdio.h>
int main () {
        int a[5][2] = {{0,0}, {1,2}, {2,4}, {3,6},{4,8}};
        int i, j;
        for ( i = 0; i < 5; i++ ) {
                for ( j = 0; j < 2; j++ ) {
                        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
                }
        }
        return 0;
}
```

Pointers

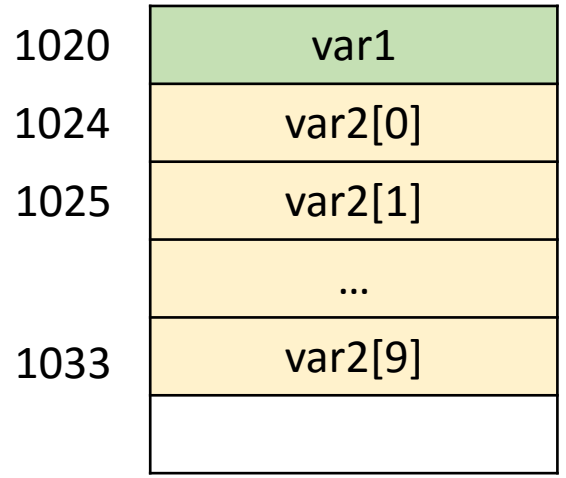
- Every variable is a **memory location** and every memory location has its **address**.
- Address can be accessed using ampersand (&) operator, which denotes an address in memory.

```c
#include <stdio.h>
int main(){
      int var=5;
      printf("Value: %d\n",var);
      printf("Address: %d\n",&var);
      return 0;
}
```

```
sh-4.3$ main
Value: 5
Address: 1278054348
```

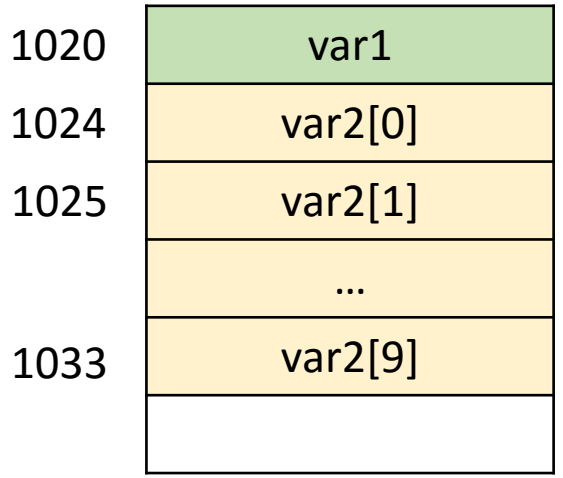
Pointers

```c
#include <stdio.h>
int main () {
        int var1;
        char var2[10];
        printf("Address of var1 variable: %x\n", &var1 );
        printf("Address of var2 variable: %x\n", &var2 );
        return 0;
}
```



Pointers

```c
#include <stdio.h>
int main () {
        int var1;
        char var2[10];
        printf("Address of var1 variable: %x\n", &var1 );
        printf("Address of var2 variable: %x\n", &var2 );
        printf("Address of var2[0] variable: %x\n", &var2[0] );
        printf("Address of var2[1] variable: %x\n", &var2[1] );
        printf("Address of var2[2] variable: %x\n", &var2[2] );
        printf("Address of var2[3] variable: %x\n", &var2[3] );
        printf("Address of var2[4] variable: %x\n", &var2[4] );
        printf("Address of var2[5] variable: %x\n", &var2[5] );
        printf("Address of var2[6] variable: %x\n", &var2[6] );
        printf("Address of var2[7] variable: %x\n", &var2[7] );
        printf("Address of var2[8] variable: %x\n", &var2[8] );
        printf("Address of var2[9] variable: %x\n", &var2[9] );
        return 0;
}
```




```
sh-4.3$ main
Address of var1 variable: a6d93a2c
Address of var2 variable: a6d93a20
Address of var2[0] variable: a6d93a20
Address of var2[1] variable: a6d93a21
Address of var2[2] variable: a6d93a22
Address of var2[3] variable: a6d93a23
Address of var2[4] variable: a6d93a24
Address of var2[5] variable: a6d93a25
Address of var2[6] variable: a6d93a26
Address of var2[7] variable: a6d93a27
Address of var2[8] variable: a6d93a28
Address of var2[9] variable: a6d93a29
```