

System SW

Lecture 6 – Basics of C programming language – Part 5

Jarno Tuominen



Lecture 8 – Basics of C programming language – Part 5

- Review
- Loop statements
- Functions



Review of last lecture

- Pointers (cont)
- C operators
- Conditional execution



Review: Pointers

- Coding rules
 - Initialize to NULL to enable testing of uninitialized pointer
 - Use some meaningful prefix/suffix for pointer variables to prevent accidental misuse of pointer variables
- NULL pointers
 - Can be used to check if malloc() succeeded
 - Can indicate end of linked list
- Void pointers
 - Used to implement general functions, which support multiple data types as arguments or as return values
 - Can be type casted to any type
- Pointer arithmetics
 - 4 arithmetic ops: ++, --, +, -
 - Comparison possible
 - Arithmetic according to data type of a pointer
- Pointers to pointers
 - ** indicates double indirection

// pointer to an integer, initialized to NULL

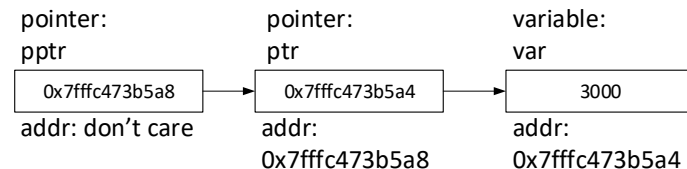
```
int *i_p = NULL;
```

```
if(i_p) {"not NULL, move on"}
```

```
if(!i_p) {"error, do something"}
```

```
int *x_p = malloc(32);
```

```
if(!i_p) {"memory allocation failed! "}
```



```
var = *ptr = **pptr
```

Review: C operators

Type	Operators	
Logical	, &&, !	= logical OR && = logical AND ! = logical NOT
Bitwise	<<, >>, , &, ^, ~	<< = Left Shift >> = Right Shift = Bitwise OR & = Bitwise AND ^ = Bitwise XOR ~ = One's Complement
Arithmetic	+, -, /, *, ++, --, %	+ = Addition - = Subtraction / = Divide * = Multiply ++ = Increment -- = Decrement % = Modulus
Relational	<, <=, >, >=, ==, !=	< = Less-Than <= = Less-Than Or Equal to > = Greater-Than >= = Greater-Than or Equal to == = Equal to != = Not Equal To

Review: Compound operators and ?:

Addition assignment	<code>a += b</code>	<code>a = a + b</code>
Subtraction assignment	<code>a -= b</code>	<code>a = a - b</code>
Multiplication assignment	<code>a *= b</code>	<code>a = a * b</code>
Division assignment	<code>a /= b</code>	<code>a = a / b</code>
Modulo assignment	<code>a %= b</code>	<code>a = a % b</code>
Bitwise AND assignment	<code>a &= b</code>	<code>a = a & b</code>
Bitwise OR assignment	<code>a = b</code>	<code>a = a b</code>
Bitwise XOR assignment	<code>a ^= b</code>	<code>a = a ^ b</code>
Bitwise left shift assignment	<code>a <<= b</code>	<code>a = a << b</code>
Bitwise right shift assignment	<code>a >>= b</code>	<code>a = a >> b</code>

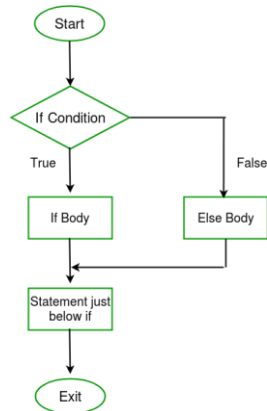
```
res = (a > b) ? x : y;
```

evaluates to

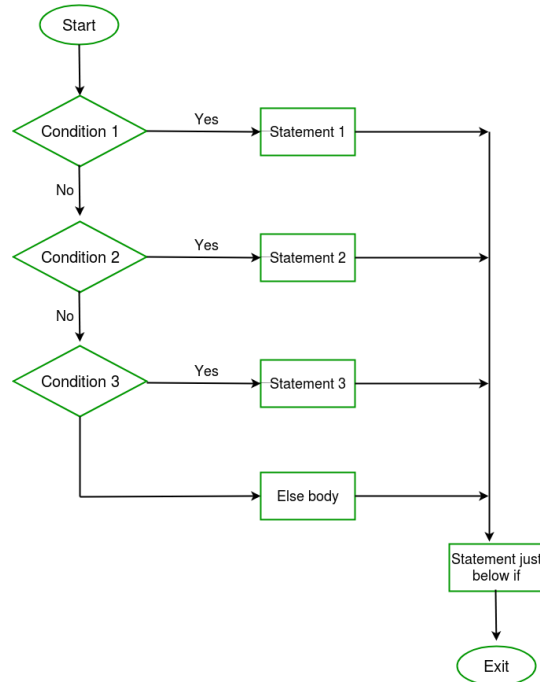
```
if (a>b) {  
    res = x;  
} else {  
    res = y;  
}
```

Review: Conditional execution

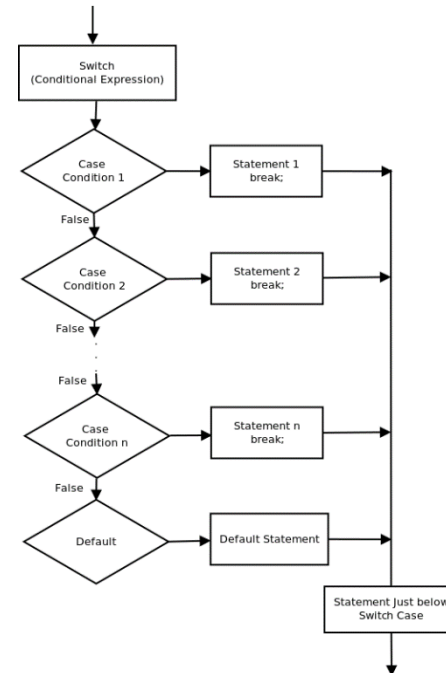
if-else



if-else if -ladder



switch-case



Lecture 8 – Basics of C programming language – Part 5

- Review
- Loop statements
- Functions

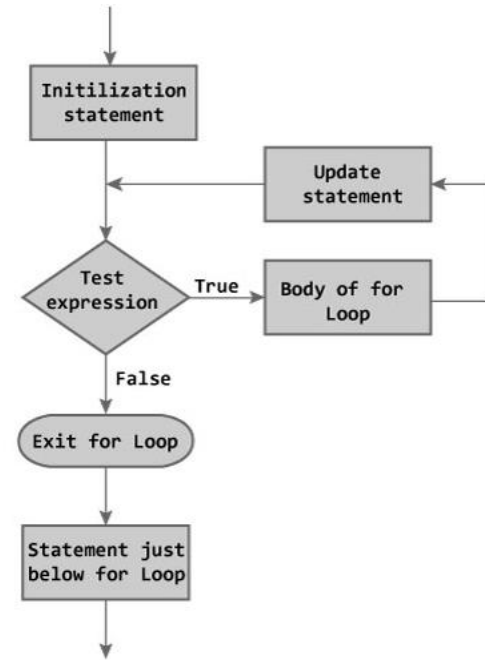


For - loops

- A block of code is repeated based on condition expression
 - Preloop condition check
- Loop counter can be declared and initialized at the for-loop statement

```
#include <stdio.h>
int main()
{
    int sum = 0;
    int num = 10;

    for(int count = 1; count <= num; count++) {
        sum += count;
    }
    printf("Sum = %d", sum);
    return 0;
}
```



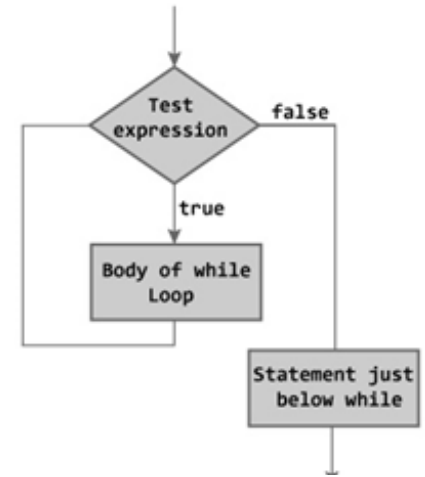
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

While - loops

- A block of code (body of while loop) is repeated based on condition check
- Preloop condition check: executed zero or more times

```
int number = 10;  
int factorial = 1;
```

```
// loop terminates when number is less than or equal to 0  
while (number > 0) {  
    factorial *= number; // factorial = factorial*number;  
    number--;  
}  
printf("Factorial= %lld", factorial);
```



auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

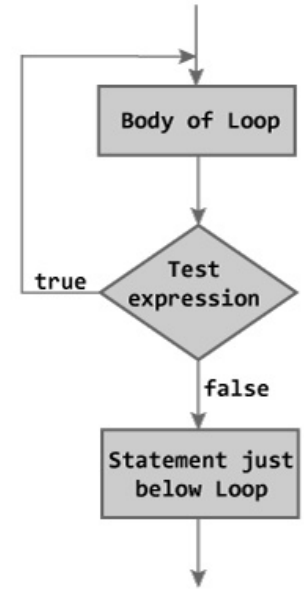
Do – While Loops

- A block of code (body of while loop) is repeated based on condition check
- Postloop condition check: executed **at least once**

```
int number, sum = 0;

// loop body is executed at least once
do {
    printf("Enter a number: ");
    scanf("%d", &number);
    sum += number;
} while(number != 0);

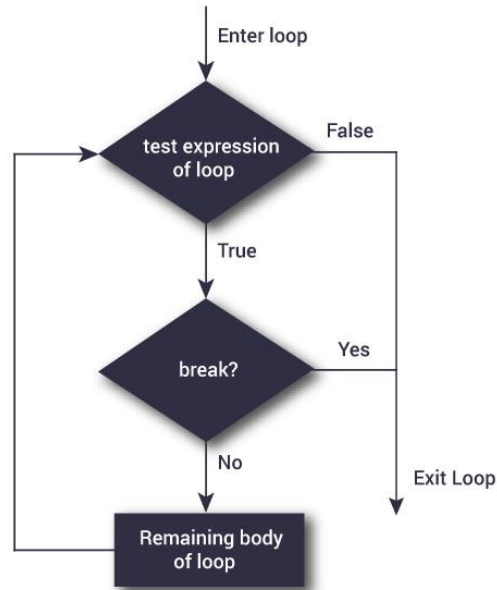
printf("Sum = %d", sum);
```



auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

break in loops

- **break** causes exit from the loop
- In case of nested loops, exits only the inner loop

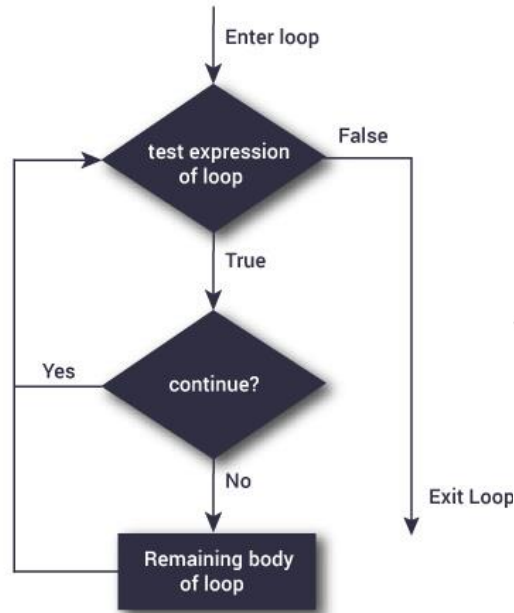


```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

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

continue in loops

- continue** stops the current iteration and moves to next iteration (and checks the condition)



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

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

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

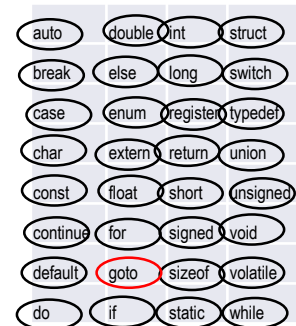
goto – for a spaghetti code

- “goto” allows you to jump unconditionally to arbitrary part of your code (within the same function)
 - the location is identified using a label
 - a label is a named location in the code. It has the same form as a variable followed by a ':'
- Generally, avoid using goto
 - Dijkstra: Go To Statement Considered Harmful. Communications of the ACM 11(3),1968
 - Excess use of goto creates spaghetti code
 - Using goto makes code harder to read and debug
 - Any code that uses goto can be written without using one

start:

```
{  
    if (cond )  
        goto outside;  
    /* some code here */  
    goto start;  
}
```

outside:



goto – ok, for error handling

- Languages like C++ and Java provide exception mechanism to recover from errors. In C, goto provides a convenient way to exit from nested blocks.

```
for (..) {  
    for (..) {  
        if (error_cond)  
            goto error; /* skips 2 blocks */  
    }  
}  
error:
```

```
cont_flag=1;  
for (..) {  
    for (init; cont_flag ;iter) {  
        if (error_cond) {  
            cont_flag =0;  
        }  
        //inner loop  
    }  
    if (!cont_flag) break;  
    //outer loop  
}
```

Lecture 8 – Basics of C programming language – Part 5

- Review
- Loop statements
- **Functions**



Function declaration / prototype

- A C program must have at least one function: `main()`, which is the only one not requiring the declaration
- A function **declaration** tells the compiler about a function's name, return type, and parameters.
 - A function needs to be declared before usage
 - Declarations/prototypes typically in `.h` -file
 - Specifies the type, unique name and parameter list of a function
 - Not needed if the function is before the calling function, in a same `.c`-file.
 - Parameter names are not important in function declaration only their type is required

```
return_type function_name( parameter list );
```

```
int max(int num1, int num2);
```

```
int max(int, int);
```

Function definition

- A function **definition** provides the actual body of the function
 - Return data type and parameters data types must match the function prototype
 - Return value can be of any data type, even a pointer – or nothing (**void**)
 - A function may be distributed as a part of a library (even encrypted) – in this case the prototype is still required.

```
return_type function_name( parameter list ) {  
    body of the function  
}
```

```
/* function returning the max between two  
numbers */  
int max(int num1, int num2) {  
  
    /* local variable declaration */  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

Static functions

- By default, all the functions are global – they have “**extern**” storage class
- In order to suppress the visibility of function to a module (file) scope, use **static**
- This prevents the usage of the function from the external modules
 - decreases the coupling between your modules
 - helps maintain your software architecture and reduces bugs

RULE: The **static** keyword shall be used to declare all functions and variables that do not need to be visible outside of the module in which they are declared.
(Barr: C coding standard 2018)

File: clkgen.c

```
static void clk_high() {  
    set_io(PIN_NUMBER);  
}  
  
static void clk_low() {  
    clear_io(PIN_NUMBER);  
}  
  
void create_8clocks() {  
    for (uint8_t i=0; i<7; i++) {  
        clk_high();  
        delay_us(10);  
        clk_low();  
        delay_us(10);  
    }  
}
```

} “private” functions

Calling a function

- When a program calls a function, the program control is transferred to the called function
- A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program
- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);

int main () {
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
    /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
}
```

```
/* function returning the max of two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Passing function arguments

- The parameters (or arguments) of a function, that are given in function call are **actual parameters**
- The parameters listed in function declaration are **formal parameters**
- Formal parameters behave like other **local** variables inside the function and are created upon entry into the function and destroyed upon exit
- While calling a function, there are two ways in which arguments can be passed to a function
 1. Call by value (the default, as in earlier example)
 2. Call by reference

Function call by value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function
- In this case, changes made to the parameter inside the function have no effect on the argument

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

```
void swap(int x, int y) {
    int temp;
    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put temp into y */
    return;
}

int main () {
    /* local variable definition */
    int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
    /* calling a function to swap the values */
    swap(a, b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
```

Function call by reference

- The **call by reference** method of passing arguments to a function copies the **address of an argument** into the formal parameter
- Inside the function, the address is used to access the actual argument used in the call
- It means the changes made to the parameter affect the passed argument

```
void swap(int *x, int *y) {  
    int temp;  
    temp = *x;      /* save the value at address x */  
    *x = *y;        /* put y into x */  
    *y = temp;      /* put temp into y */  
    return;  
}  
  
int main () {  
    /* local variable definition */  
    int a = 100;  
    int b = 200;  
    printf("Before swap, value of a : %d\n", a );  
    printf("Before swap, value of b : %d\n", b );  
    swap(&a, &b);  
    printf("After swap, value of a : %d\n", a );  
    printf("After swap, value of b : %d\n", b );  
    return 0;  
}
```

```
Before swap, value of a : 100  
Before swap, value of b : 200  
After swap, value of a : 200  
After swap, value of b : 100
```