

# System SW

Lecture 5 – Basics of C programming language – Part 4

Jarno Tuominen



# Lecture 7 – Basics of C programming language – Part 4

- Review
- Pointers (cont)
- C operators
- Conditional execution



# Review of last lecture

- Data types array & string
- Type qualifiers
- Storage classes
- Type casting
- Pointers



# Review: Arrays

- An array is a container for data which is of all the **same** type
- The data type can be a basic type (integer, char etc) or a derived type (struct, pointer, another array etc)
- Defined by square brackets [ ]
- Static arrays: The size of an array is known at the compile time
- Dynamic arrays involve the use of pointers and malloc() to allocate memory from the heap memory area (will be covered later on)

```
//Declaration
type name[number of elements];
int numbers[6];

//Declaration + initialization
type name[number of elements]=
{comma-separated values};

int point[6]={0,0,1,0,0,0};

//When initializing at declaration,
//size can be omitted
int point[]={0,0,1,0,0,0};

//Multi-dimensional arrays
char two_d[2][3];

int two_d[][] = {
    { 5, 2, 1 },
    { 6, 7, 8 }
};
```

# Review: Strings in C

- There is no string type in C – instead, strings are stored in **character arrays**
- A string is **terminated** by a null-character '\0', which is automatically added to the end by compiler in the first example
- In the latter example null-character needs to be inserted manually
- All the string library functions expect strings to be terminated by a null-character
- String indexing starts from zero
- String handling routines are in string library <string.h> (along with useful memory handling routines)

```
char string[5] = "Blaa";
```

```
char string[5] = {'B','l','a','a','\0'};
```

0	1	2	3	4
B	l	a	a	\0

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char string1[5] = "Blaa";
```

```
    char string2[5] = {'B','l','a','a','\0'};
```

```
    char string3[4] = "Blaa"; //no!!!
```

```
    printf("%s\n",string1);
```

```
    printf("%s\n",string2);
```

```
    printf("%s\n",string3);
```

```
    return 0;
```

```
}
```

```
Blaa
Blaa
Blaa
```

← Oops!

# Review: Type qualifiers / volatile

- All variables are considered "unqualified" by default - Type qualifiers modify the properties of variables in certain ways
- C has 4 type qualifiers:
  - Const
  - volatile (C89)
  - restrict (C99, skipped)
  - \_Atomic (C11, skipped)
- `const`
  - Enables optimization, value cannot change, storage to code memory
- `volatile`
  - Tells the compiler: Do not optimize because variable value can change out of context, any time
  - Use with variables which are used by interrupt handlers

# Review: Storage classes of variables and functions

- Storage Classes are used to describe about the features of a variable/function.
- 4 storage classes:

**static**

**functions** can be called only from the same .c-file

**variables** preserve their value even after they are out of scope

**auto** - a local variable in a function body (default, so you don't have to use this ever)

**extern** - tells compiler that the variable is defined elsewhere (default for globals)

**register** - a hint to compiler to cache the variable in a processors register.

Avoid, if you are not smarter than compiler

**RULE:** don't use keyword **auto** – it's completely useless

**RULE:** use keyword **static** whenever possible

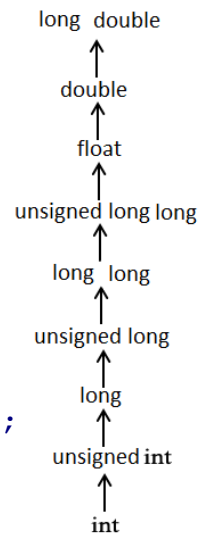
(Barr: C coding standard 2018)

# Review: C type casting

- Type casting is a way to convert a variable from one data type to another explicitly using the **cast operator**  
`(type_name) expression`
- Compiler does automatic conversions from smaller integer types to larger types, this is called **integer promotion**
- Casting is potentially dangerous! If you cast a "large" value to a "smaller" data type, data loss will happen
- RULE:** Each cast shall feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side

```
main() {  
    int sum = 17, count = 5;  
    double mean;  
    mean = (double) sum / count;  
    printf("Value of mean: %f\n", mean);  
}
```

```
main() {  
    int i = 17;  
    char c = 'c'; //ascii: 99  
    float sum;  
    sum = i + c;  
    printf("Value of sum: %f\n", sum);  
}
```





# Review: Pointers

- Each variable in your program is stored in the memory – in other words, it has a memory location (an address)
- What is a pointer?
  - A **pointer** is a variable whose **value** is the **address** of another variable, i.e., address of a memory location
  - A pointer's value can be also an address of a function – then it is called a **function pointer**
- The variable address &var is stored to a pointer variable i\_p
- And the value of the variable the pointer is pointing to, can be read with \* – this is called **dereferencing**

Pointer variable declaration:

```
data_type * name_p;
```

```
int * i_p;           // pointer to an integer  
my_struct_t * struct_p; /* pointer to my_struct type */
```

Assign a value with address-of &

```
int var = 20; //variable declaration  
int * i_p;    //pointer variable decl.  
i_p = &var;   //store address of var to i_p
```

Dereferencing \*i\_p gives you value-at-address stored in a pointer variable

```
printf("Value of *i_p variable: %d\n", *i_p );
```

# Review: Pointers with structs – an example

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

typedef struct {
    char title[50];
    int book_id;
} Book_t;

#define NUMBER_OF_BOOKS 3

int main()
{
    Book_t library[NUMBER_OF_BOOKS]; //create array of books
    Book_t * my_ptr; //Pointer declaration operator (points to Book_t type)
    my_ptr = &library[0]; // Address-of operator picks the
                          // address of library and assigns it to my_ptr

    //Let's check what's the size of Book_t
    printf("Size of Book_t:\t%d bytes.\n\n",sizeof(Book_t));

    //populate the library with books
    const char title_base[] = "Name of the book ";
    char letter[2] = {'A','\0'};
    for (uint8_t i=0; i<NUMBER_OF_BOOKS; i++) {

        strcpy(&my_ptr->title[0], &title_base[0]);
        strcat(&my_ptr->title[0], &letter[0]);

        my_ptr->book_id = 1000+i;
        letter[0]++; //pick the next letter
        my_ptr++; //increment the pointer to next "book slot" in the library
    }
```

```
//Run inventory for the library

//roll back the pointer to first book in library
my_ptr = &library[0];

printf("Contents of the library:\n\n");
printf("Book Title:\t\tBook id:\n");
for (uint8_t i=0; i<NUMBER_OF_BOOKS; i++) {
    printf("%s\t%d\n",my_ptr->title,my_ptr->book_id);
    my_ptr++; //increment the pointer
              // to next "book slot" in the library
}

return 0;
}
```

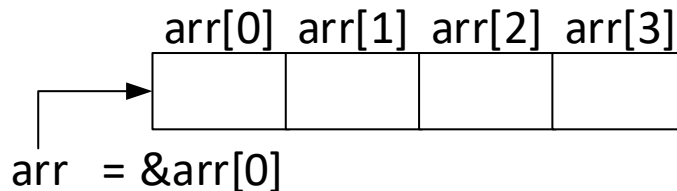
Size of Book\_t: 56 bytes.

Contents of the library:

Book Title:	Book id:
Name of the book A	1000
Name of the book B	1001
Name of the book C	1002

# Recap: Relation between arrays and pointers

- C does not have array variables (or string variables, which are char arrays)
- In fact, the array notation `[]` is actually an alternate way of using pointers
- Consider an array: `int arr[4]`
- Name of the array always points to the first element `[0]` of array
- This means that `arr` is equivalent to `&arr[0]`
- Furthermore, statement  
`arr[2] = 5;` is identical to  
`*(arr+2) = 5;`
- So `arr` is already a pointer!
- BUT, it is a constant pointer, you cannot apply arithmetics to the name of an array!



Note: `strcpy` is a function that take pointer arguments

```
char *strcpy(char *dest, const char *src)
```

```
strcpy(&my_ptr->title[0], &title_base[0]);
```

`my_ptr->title` is an character **array**, so are `title_base` and `letter` – they are already pointers

Therefore, we don't need `&`-operator

```
strcpy(my_ptr->title, title_base);
```

# Lecture 7 – Basics of C programming language – Part 4

- Review
- **Pointers (cont)**
- C operators
- Conditional execution



# Pointers – some coding rules

- At the declaration, pointers should be always initialized to NULL
- This way you can test for possible NULL-pointers before deferencing them (or worse: jumping to that memory address if it was a function pointer)
- Naming convention: use `_p` for all the pointers! This helps you avoid many bugs as you don't accidentally assign a value for the pointer, when the intention was to dereference it
- Coding guideline:
  - When declaring pointers, use whitespace before and after `'*'`
  - When using `&`-operator, tie it to the operand
  - When dereferencing tie `*` to the operand (pointer variable)

// 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 * i_p = NULL;
```

```
i_p = &var;
```

```
*i_p = 50;
```

# NULL pointers

- On most of the operating system, programs are not permitted to access memory at address 0, because the memory location is reserved by the OS
- Memory address 0 has a special significance: it signals that the pointer is not intended to point to an accessible memory location
- You can use null pointers as markers, for example to indicate the end of a linked list, or to test if memory allocation failed

```
int *x_p = malloc(32);  
if(!i_p) {"memory allocation failed! "}
```

# Void pointers

- A void pointer is a pointer that
  - has no associated data type with it
  - can hold address of any type
  - can be typecasted to any type
- Used to implement generic functions
- For example malloc() returns a void pointer to the allocated memory location
  - Can be typecasted to any type
  - In C, automatic casting takes place
  - In C++, you have to do it explicitly!

```
int a = 10;  
char b = 'x';
```

```
void *x_p = &a; // void pointer  
holds address of int 'a'  
x_p = &b; // void pointer holds  
address of char 'b'
```

```
int *x_p = malloc(sizeof(int) * 16);
```

```
int *x_p = (int *) malloc(sizeof(int) * 16);
```

# Pointer arithmetics

- C pointer is a variable that holds a numeric value (an address), which means that you can perform arithmetic operations to it
  - Four arithmetic operators can be used with pointers: +, -, ++, --
- At the declaration of the pointer variable, the data type is defined. Therefore, when incrementing the value of the pointer, it takes the size of the data type into account

```
typedef struct {char title[50];  
int book_id; } Book_t;
```

```
uint64_t delta_addr;  
Book_t library[5];  
Book_t * my_ptr, * my_ptr2;  
my_ptr = &library[0];  
my_ptr2 = my_ptr;  
my_ptr2++;  
//Let's check what's the size of Book_t  
printf("Size of Book_t:\t%lu bytes.\n\n", sizeof(Book_t));  
printf("ptr1: 0x%lx\nptr2: 0x%lx\n", my_ptr, my_ptr2);  
  
delta_addr = my_ptr2 - my_ptr; //delta in book_t's  
printf("ptr2-ptr1: %lu\n", delta_addr);  
//cast pointers to integers  
delta_addr = (uint64_t)my_ptr2 - (uint64_t)my_ptr;  
//now we see delta in bytes  
printf("ptr2-ptr1: %lu\n", delta_addr);
```

Size of Book\_t: 56 bytes.

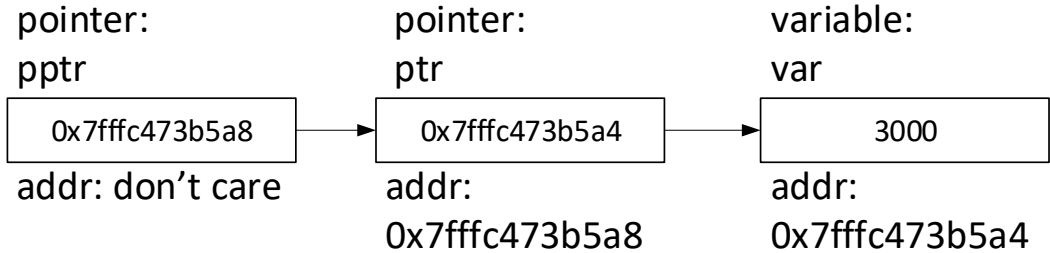
```
ptr1: 0x7ffc9dcff370  
ptr2: 0x7ffc9dcff3a8  
ptr2-ptr1: 1  
ptr2-ptr1: 56
```



# Pointers to pointers

- A pointer to pointer is a form of multiple indirection
- Normally, a pointer contains the address of a variable, but it can also contain an address of another pointer variable

```
Address of var = 7fffc473b5a4
Address of ptr = 7fffc473b5a8
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```



```
int var;
int * ptr;
int ** pptr;
var = 3000;
ptr = &var; // take the address of var
pptr = &ptr; // take the address of ptr
printf("Address of var = %lx\n", &var );
printf("Address of ptr = %lx\n", &ptr );
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
```

# Pointers are so complex – why all the hassle?

- Instead of passing large amounts of data to functions (and returning it back), functions can take pointers as argument allowing the function to manipulate data directly in that memory location
  - Less memory transfer/copy operations -> faster code!
- By default, C uses "call by value" to pass arguments to functions – in this case changes made to the parameter inside function has no effect on the argument
- "Call by reference" means that the address of the argument (i.e. the pointer) is passed to the function, which can directly alter the value of the argument
  - We'll revisit this when covering functions

# Lecture 7 – Basics of C programming language – Part 4

- Review
- Pointers (cont)
- **C operators**
- Conditional execution



# C operators

- Used to assign, compare and modify data
- Unary, binary and ternary operators (i.e. takes one, two or three operands)
- Some operators can be combined with assignment -> compound operators
- Precedence rules (order of operations) are not very clear – use parenthesis () when in doubt!

Type	Operators
Logical	, &&, !
Bitwise	<<, >>,  , &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, >=, ==, !=

# Logical operators

Type	Operators
Logical	, &&, !
Bitwise	<<, >>,  , &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, >=, ==, !=

Boolean conditions:

False = any condition that is **zero**

True = any condition that is **non-zero**

`||` = logical OR

`&&` = logical AND

`!` = logical NOT

```
if ( condition1 || condition2 )
```

```
if ( condition1 && condition2 )
```

```
while ( ! condition )
```

# Bitwise operators

Type	Operators
Logical	, &&, !
Bitwise	<<, >>,  , &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, >=, ==, !=

<< = Left Shift

>> = Right Shift

| = Bitwise OR

& = Bitwise AND

^ = Bitwise XOR

~ = One's Complement

```
foo = varA >> 4;  
foo = varA | varB;  
foo = varA & varB;  
foo = varA ^ varB;  
foo = ~varA;
```

## Right-Shift (logical)

varA >> 4 → Right Shift 4 bits

Ex: 0b10010110 >> 4

= 0b00001001 (zeros shifted in)  
(lower bits truncated)

## AND

foo = varA & varB;

varA = 0xF5;	→	1	1	1	1	0	1	0	1
varB = 0x5B;	→	0	1	0	1	1	0	1	1
foo = 0x51	→	0	1	0	1	0	0	0	1

# Arithmetic operators

Type	Operators
Logical	, &&, !
Bitwise	<<, >>,  , &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, >=, ==, !=

```
foo = varA + varB;  
foo = varA - varB;  
foo = varA * varB;  
foo = varA % 2;  
foo = varA++;
```

**+** = Addition  
**-** = Subtraction  
**/** = Divide  
**\*** = Multiply  
**++** = Increment  
**--** = Decrement  
**%** = Modulus

## Modulus Operator (Remainder)

15 % 4 -> 3

## Increment/Decrement

varA++ -> varA = varA + 1;  
varA-- -> varA = varA - 1;

# Relational operators

Type	Operators
Logical	, &&, !
Bitwise	<<, >>,  , &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, >=, ==, !=

< = Less-Than

<= = Less-Than Or Equal to

> = Greater-Than

>= = Greater-Than or Equal to

== = Equal to

!= = Not Equal To



# Concerning equivalence checks

//Beware of these nasty mistakes:

if (a=0) {...}

This will set **a** to zero, instead of testing it against zero, the result is always true! Difficult to catch, but at least most compilers or even editors give you a warning, but not all.

//Place constant to left side instead:

if (0=a) {...}

Now we try to assign a value to constant, which will immediately cause compiler error

//Preferably, don't make mistakes ;)

if (0==a) {...}



# Conditional operator ?:

- The only ternary operator in C
- Also known as inline if (iif) or ternary if
- Just a shorter notification, does not make your code any better
- You cannot set a breakpoint to either case
- Think twice before showing off with ?:

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

evaluates to

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

# Compound assignment operators

- Syntactically, it is possible to combine assignment with bitwise or arithmetic operators
- Think: Do they really make your code more readable and less prone to errors?

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 &amp;= b</code>	<code>a = a &amp; 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 &lt;&lt;= b</code>	<code>a = a &lt;&lt; b</code>
Bitwise right shift assignment	<code>a &gt;&gt;= b</code>	<code>a = a &gt;&gt; b</code>

# Lecture 7 – Basics of C programming language – Part 4

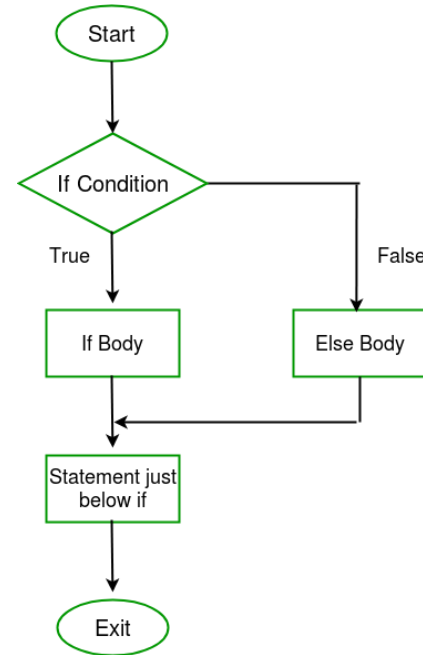
- Review
- Pointers (cont)
- C operators
- Conditional execution



# if-else statements

- An **if statement** consists of a boolean expression followed by one or more statements
- An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false

```
int main() {  
    int i = 20;  
    if (i < 15) {  
        printf("i is smaller than 15\n");  
    } else {  
        printf("i is greater than 15\n");  
    }  
    printf("Always prints this\n");  
    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

# Practical tip

```
if (a==b) c++;
```

Works, but you cannot set a breakpoint when the condition is true -> difficult to debug -> put the statement on it's own code line

```
if (a==b)
```

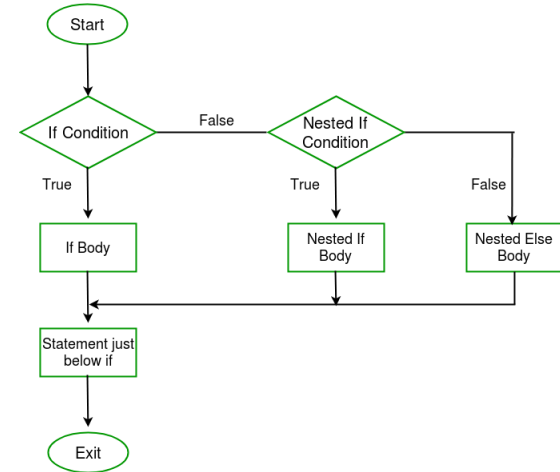
```
    c++;
```

Now we can set a breakpoint on this line!

# Nested if-else

- You can use one **if** or **else if** statement inside another **if** or **else if** statement(s)
- Gets easily quite complex to follow – consider switch-case statements instead

```
int main() {  
    int i = 10;  
  
    if (i == 10) {  
        if (i < 15)  
            printf("i is smaller than 15\n");  
  
        if (i < 12)  
            printf("i is smaller than 12 too\n");  
        else  
            printf("i is greater than 15\n");  
    }  
    return 0;  
}
```

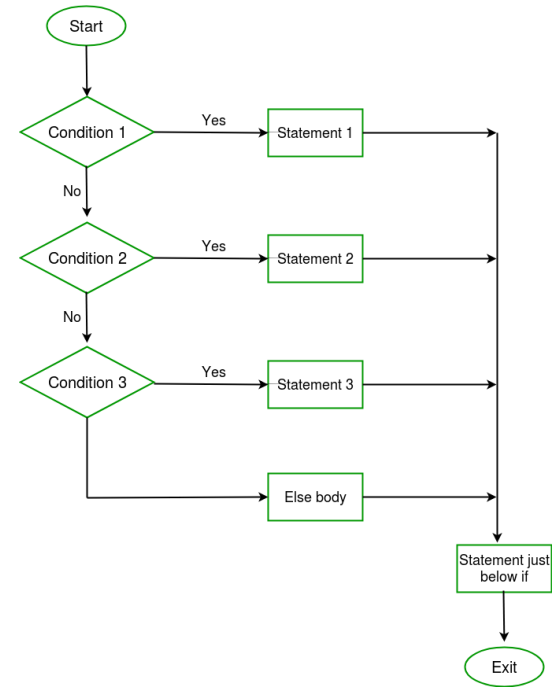


# if-else if -ladder

- In case you need to decide among multiple options
- Only one statement will be executed
- This is clearly a place for switch-case

```
int main()
{
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is something else");
}
```





# switch – case

- Switch case statements are a substitute for long if statements that compare a variable to several integral values
- Note: only constant cases allowed – (enumerated values are constants)
- Remember "break" at the end of each case
- Default case: if no other cases matched

```
switch (x){  
    case 1: printf("Choice is 1");  
            break;  
    case 2: printf("Choice is 2");  
            break;  
    default: printf("Choice was other than 1 or 2");  
            break;  
}
```

