PROGRAMMERING AV INBYGGDA SYSTEM

basic C crash course

Dario Salvi, 2024

Contents

- https://www.tutorialspoint.com/cprogramming/index.htm
- Bilting, Ulf & Skansholm, Jan (2011). *Vägen till C*. 4., [rev. och uppdaterade] uppl. Lund: Studentlitteratur
- Deitel, Paul J. & Deitel, Harvey M. (2013). *C: how to program*. 7. ed. Boston: Pearson
- Kernighan, Brian Wilson & Ritchie, Dennis M. (1988). *The C programming language*. 2. ed. Englewood Cliffs: Prentice Hall
- King, K. N. (2008). *C programming: a modern approach*. 2. ed. New York: W.W. Norton & Company

C is cool!

- C is simple.
  - Minimalistic language.
- C is everywhere.
  - Used for operating systems, compilers, standard libraries, etc.
- C is stable.
  - Very few updates, learn once use it forever.
- C is fast.
  - No (high-level) language beats C in performance.
- But…
  - C is low-level compared to many other languages.
  - C is not memory-safe -> can lead to security bugs.

Types

- **Integers**: whole numbers which can be either positive or negative. Defined using *char*, *int*, *short*, *long* or *long long*.
- **Unsigned integers** - whole numbers which can only be positive. Defined using *unsigned char*, *unsigned int*, *unsigned short*, *unsigned long* or *unsigned long long*.
- **Floating point numbers** - real numbers (numbers with fractions). Defined using *float* and *double*.
- **Structures** - will be explained later, defined using *struct*.

Variables

- Variables are always defined with a **type** and a **name**.
- Optionally, they can be assigned a value.

```
float foo, bar;
int mem = -10;
```

Type sizes in C

- C does not impose a specific size for basic types (for example int)
- The environment (architecture, compiler, libraries) defines the size !
- Use sizeof() to get the number of bytes of a type.
- Example: Arduino
  - On the Arduino Uno an int stores a 16-bit (2-byte) value
  - On the Arduino Due an int stores a 32-bit (4-byte) value
- There are standard libraries to specify the size in bits!
- Example: C99 ***inttypes.h***
  - #include <inttypes.h>
    uint32_t LongVariable;
    uint8_t ByteVariable; /* 0..255 */
    int16_t SignedVariable /*−32,768..32,767 */

https://en.wikipedia.org/wiki/C_data_types

Live coding!

- See document on Canvas to replicate it.
- Needed materials:
  - VSCode
  - C/C++ extension
  - Code Runner extension

Arithmetic operators

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical operators

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Note: in C **0** is false, **any other value** is true.

Bitwise operators

Assume A = 60 (0011 1100) and B = 13 (0000 1101)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment operators

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

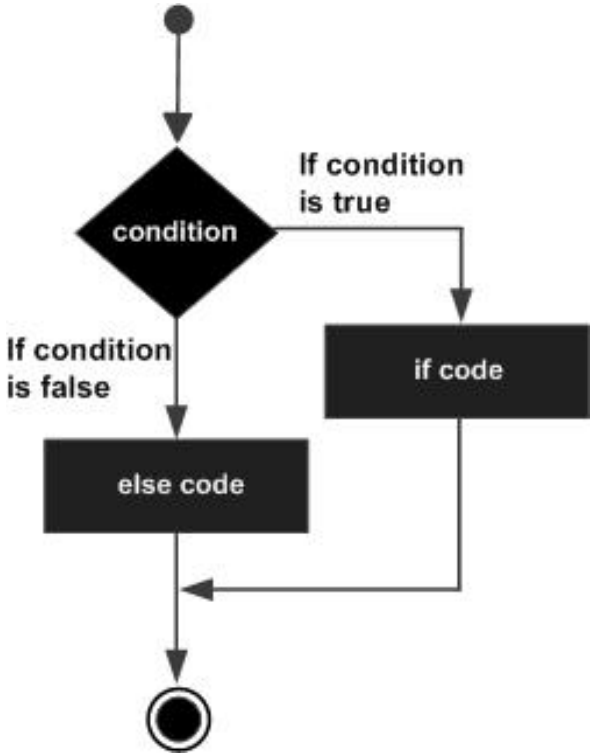
Live coding!

- Examples of operators.
- See document on Canvas to replicate it.

Conditions

- If .. Else are used to execute a piece of code depending on the value of a variable (condition).

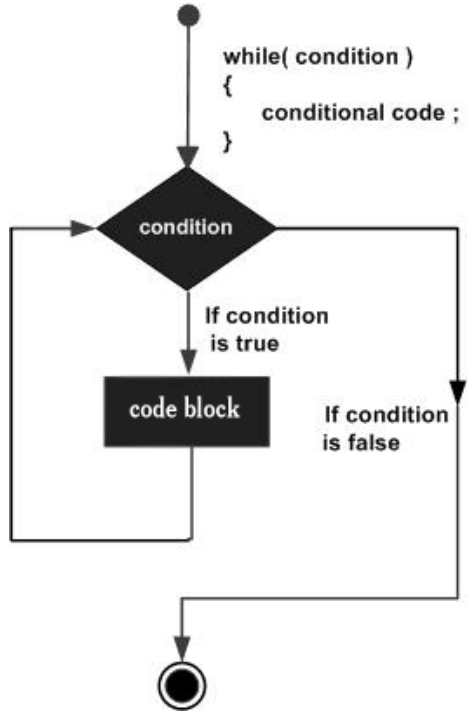
```
if (boolean_expression) {
/* statement(s) will execute if the boolean expression is true */
} else {
/* statement(s) will execute if the boolean expression is false */
}
```



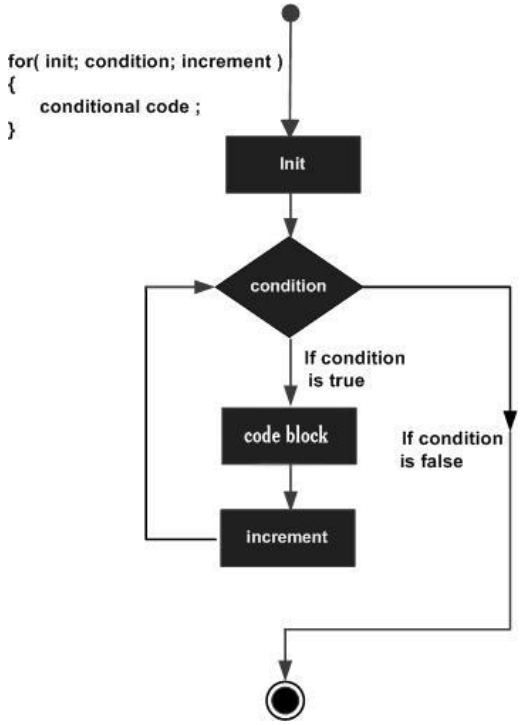
- More conditions can be chained: if … else if … else …
- If you have many conditions, you can use switch … case

Loops

- While: executes statements as long as a condition is true.




```
while(condition) {
  statement(s);
}
```



- For: executes a statement a number of times.

```
for ( init; condition; increment ) {
  statement(s);
}
```

Live coding!

- Examples of conditions and loops.
- See document on Canvas to replicate it.

Pointers

- Every variable is a memory location and every memory location has an address which can be accessed using ampersand (&) operator, which denotes an address in memory.
- A **pointer** is a variable whose value is a memory, for example the address of another variable. It is declared with
  type *var-name;
  - **type** is the pointer's base type
  - **var-name** is the name of the pointer variable.
  - asterisk * is used to declare a pointer
- A non-defined pointer is usually assigned the value **NULL** (=0)

Pointers operators

Pointers, Parenthesis and Math

Pointer Thing	Memory Address	Memory Contents
`p`	Yep	Nope
`*p`	Nope	Yep
`*p++`	Incremented after value is read	Unchanged
`*(p++)`	Incremented after value is read	Unchanged
`(*p)++`	Unchanged	Incremented after it's used
`*++p`	Incremented before value is read	Unchanged
`*(++p)`	Incremented before value is read	Unchanged
`++*p`	Unchanged	Incremented before it's used
`++(*p)`	Unchanged	Incremented before it's used
`p*++`	Not a pointer	Not a pointer
`p++*`	Not a pointer	Not a pointer

The `++` operator is used above, though any math operation can be substituted.

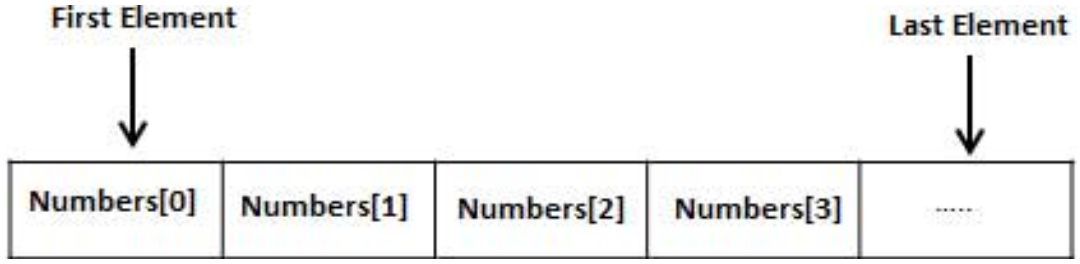
A tip: Use parenthesis to isolate part of the pointer problem and the answer will always work out the way you intended.

Live coding!

- See document on Canvas to replicate it.

Arrays

- Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
- Arrays are declared with:
  type arrayName [arraySize];
- Accessed with: arrayName[index]



Live coding!

- Examples of arrays in C.
- See document on Canvas to replicate it.

Functions

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is **main().**
- A function *declaration* tells the compiler about a function's name, return type, and parameters. A function *definition* provides the actual body of the function.

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

Function Arguments

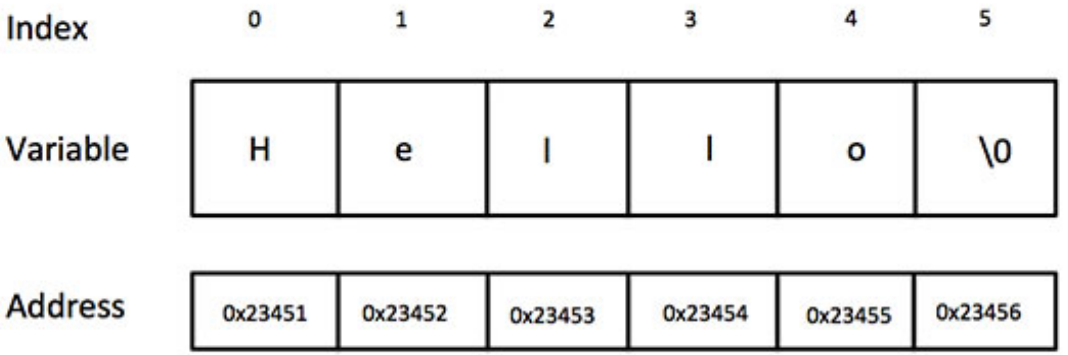
- If a function is to use *arguments*, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- Formal parameters behave **like other local variables** inside the function and are created upon entry into the function and destroyed upon exit.
- Call by value:
  - This method 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.
- In C functions are always called by value!
- Emulated call by reference:
  - This method 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. This means that changes made to the parameter affect the argument.

Live coding!

- Examples of functions in C.
- See document on Canvas to replicate it.

Strings

- Strings are actually one-dimensional array of characters terminated by a **null** character '\0’.
- These two are equivalent:
  - char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0’};
  - char greeting[] = "Hello";



- C provides a number of functions to manipulate strings in string.h
  - **strcpy(s1, s2)** copies s2 into s1
  - **strcat(s1, s2)** concatenates string s2 onto the end of string s1
  - **strcmp(s1, s2)** returns 0 if s1 and s2 are the same
  - **strstr(s1, s2)** returns a pointer to the first occurrence of string s2 in string s1

Live coding!

- Examples of string manipulation.
- See document on Canvas to replicate it.

Structs

- **structure** is another user defined data type available in C that allows to combine data items of different kinds.
- Structures are used to represent a record with different fields.
- To access any member of a structure, we use the **member access operator (.)**.
- You would use the keyword **struct** to define variables of structure type.
- If you have a **pointer** to a struct you can use the **arrow operator (->)** to access its members.

```
struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} book;

struct Books Book1;
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "Tutorial");
Book1.book_id = 6495407;
```

Unions

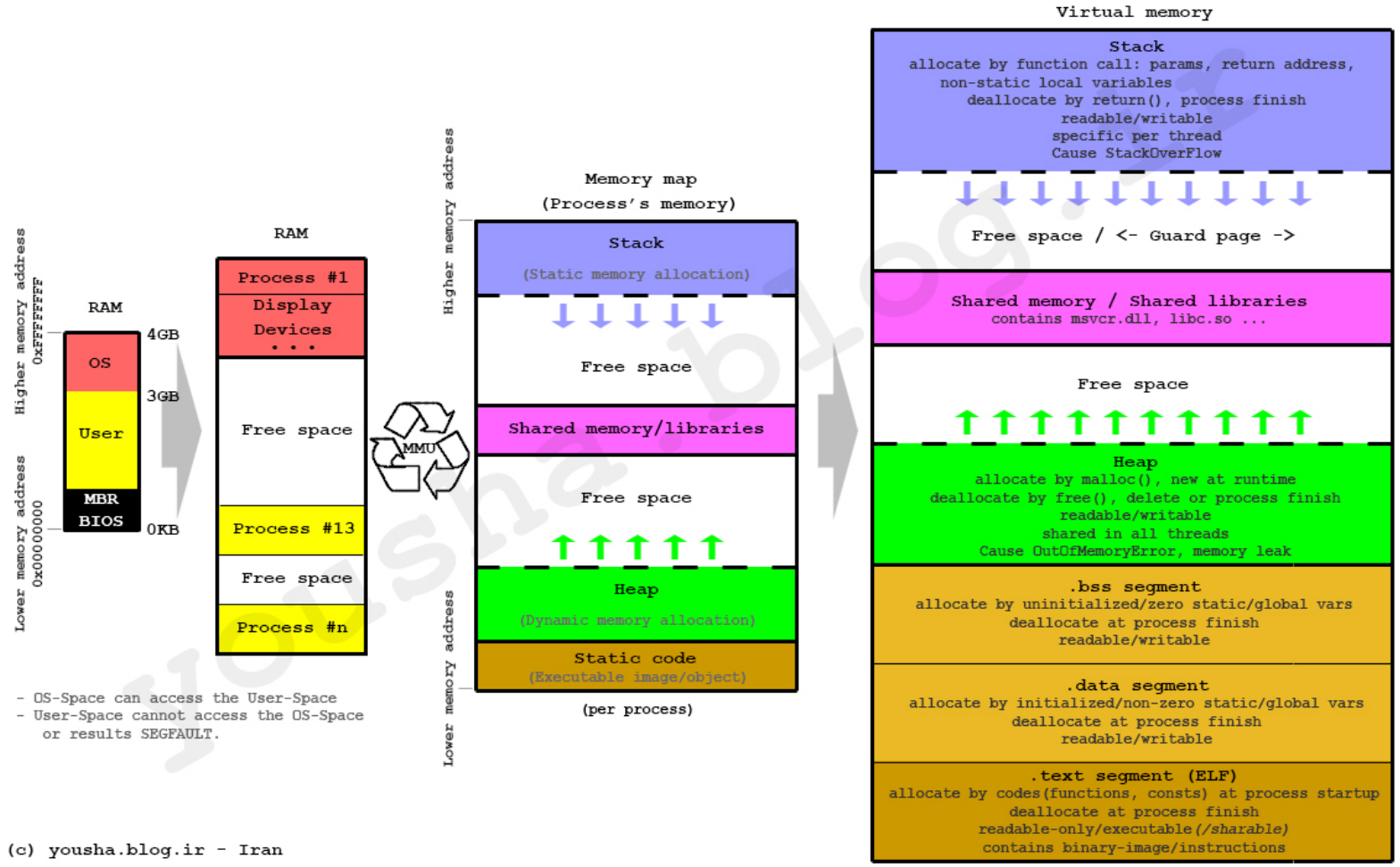
- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple-purpose.

```
union Data {
  int i;
  float f;
  char str[20];
} data;
```

Live coding!

- Examples of structures in C.
- See document on Canvas to replicate it.

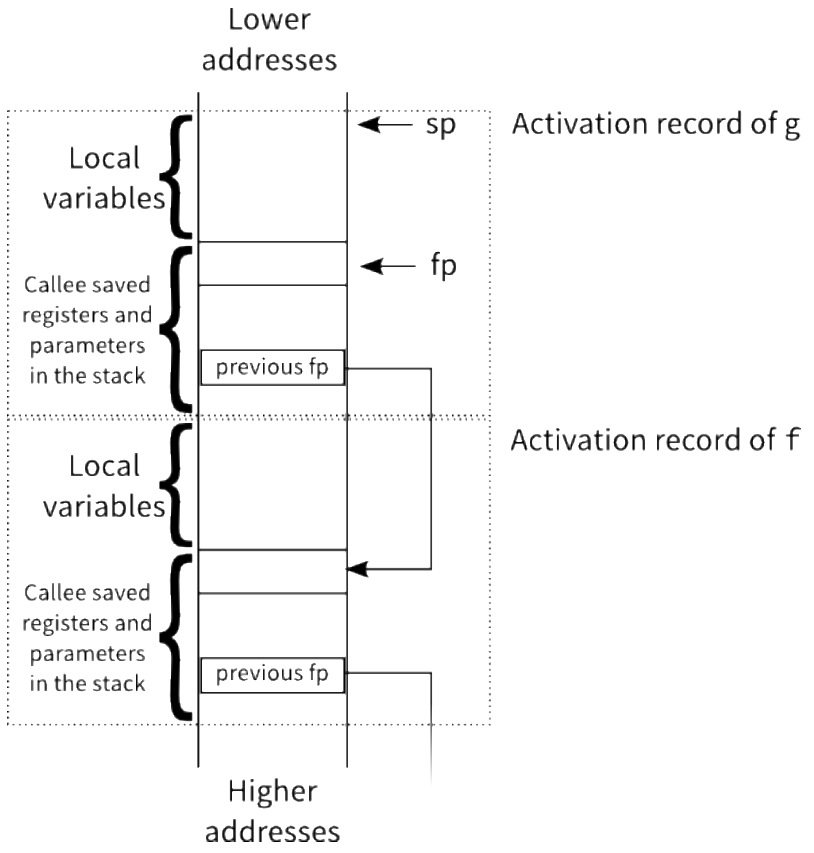
Memory allocation



https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

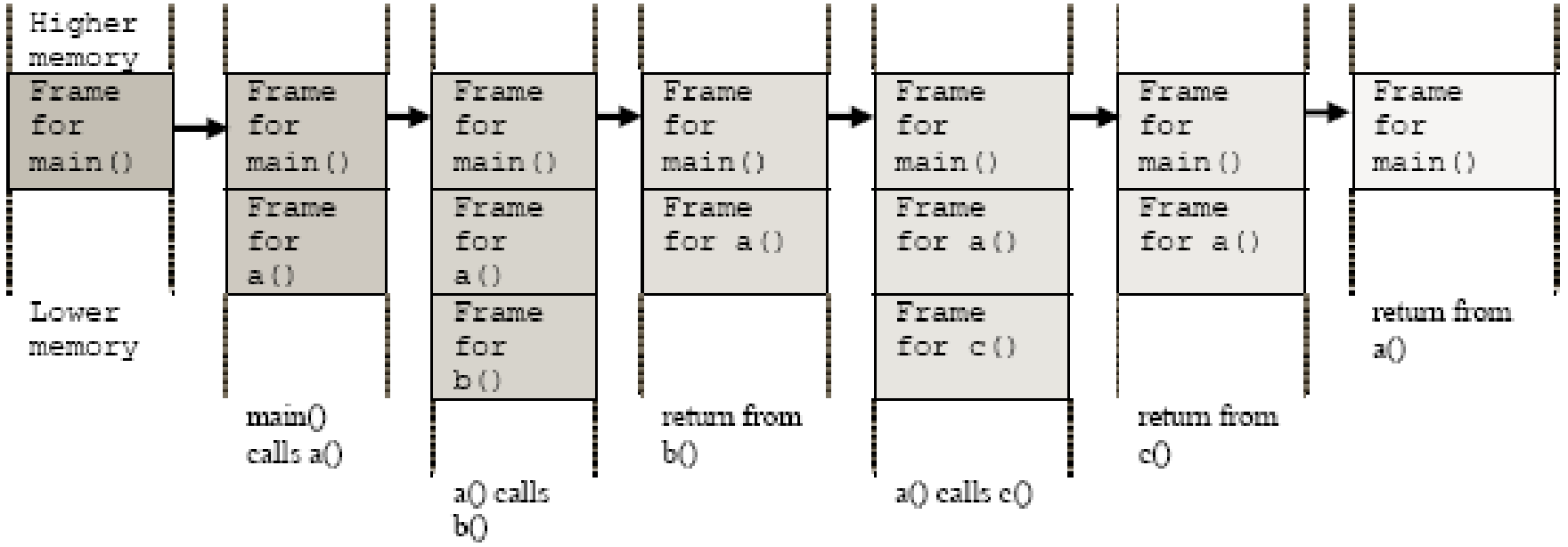
The Stack

- The stack size is allocated in RAM at runtime.
- The stack holds:
  - The return address of functions.
  - Local variables, including local arrays, structures, unions, and in C++, classes.
  - Registers that must be preserved when register contents are saved on entry into subroutines.
- It is possible to overflow !



Example of stack allocation

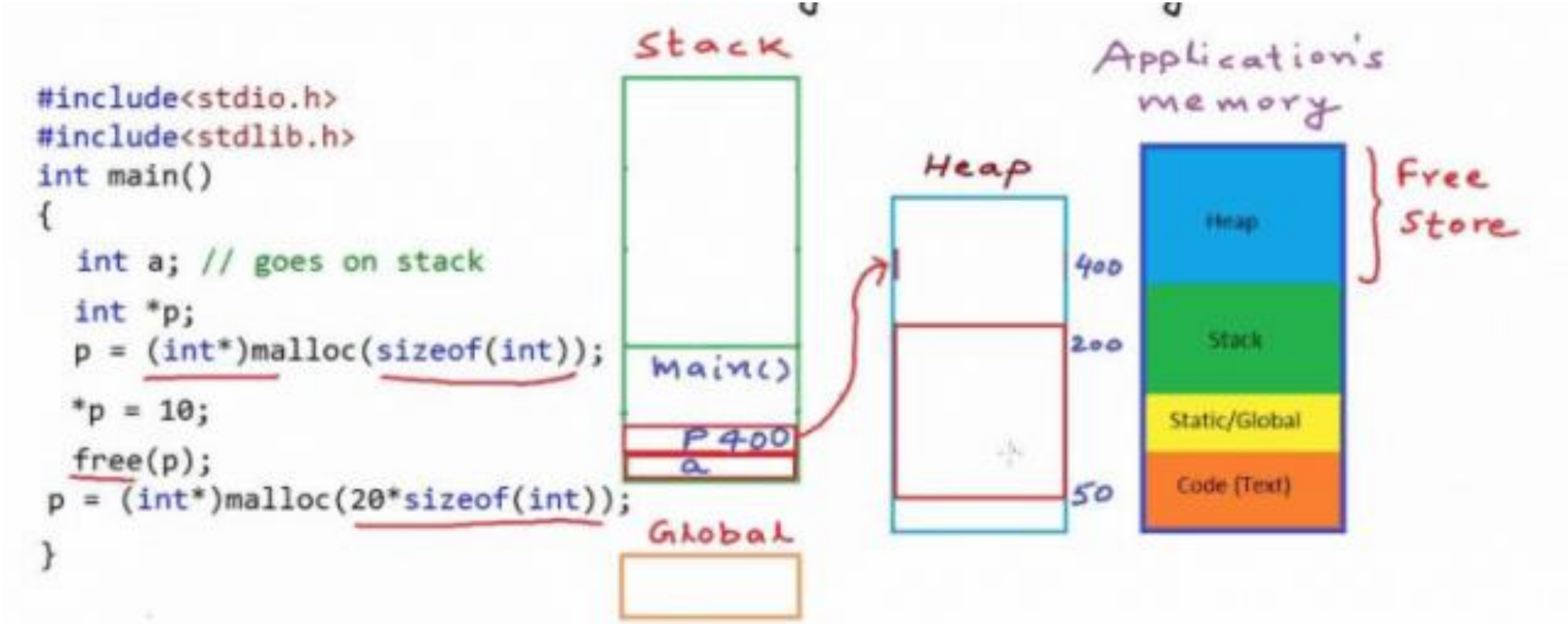
```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```



Allocating variables

- Variables defined outside of functions are allocated in the static code area.
- Variables defined inside functions are allocated in the stack.
- Memory can be obtained from the heap using the function:
  void *malloc(int num_bytes);
  this returns the starting address in the heap with num_bytes guaranteed to be accessible.
- Once memory is not needed any longer, it can be returned to the heap with: **void free(void *address);**

The Heap and the Stack



https://www.youtube.com/watch?v=_8-ht2AKyH4

Live coding!

- Example of memory allocation in C.
- See document on Canvas to replicate it.

Recursion

- In C, 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();
}
```

- If the recursive function does not have an exit condition you cause a stack overflow.

Live coding!

- Example of recursion in C.
- See document on Canvas to replicate it.

Example exam questions

- Name 5 types used in C.
- What is the size of an integer in C?
- How is a for loop structured? What is it used for?
- Are arguments passed by value or reference in C?
- What is the difference between a *struct* and a *union* in C?
- Describe the use of the *stack* and the *heap* memory in C.
- Give an example of how the stack can overflow.
- Give an example of a recursive function. Can you also provide its code?