VARIABLE SCOPE AND FUNCTION DEFINITION

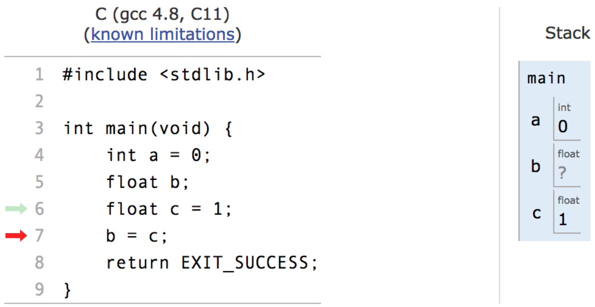
Introduction to C Programming

université de BORDEAUX

VARIABLE SCOPE

BIRTH AND LIFE OF A VARIABLE

- ▷ A block is determined by enclosing curly brackets
- ▷ A variable's lifetime is limited to the block where it was declared.
- ▷ The corresponding block defines the variable's scope



C (gcc 4.8, C11)
(known limitations)

```c
#include <stdlib.h>

int main(void) {
    int a = 0;
    float b;
    float c = 1;
    b = c;
    return EXIT_SUCCESS;
}
```

Stack

main

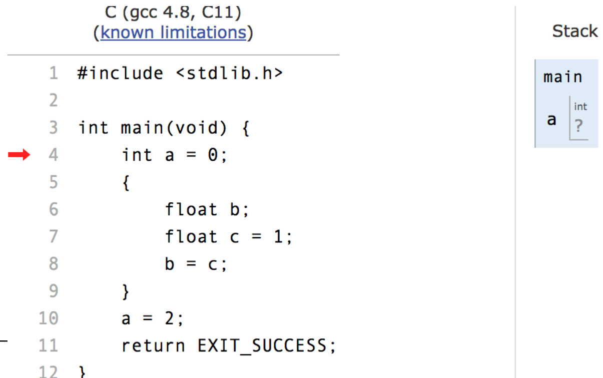
a int 0

b float ?

c float 1

BIRTH AND LIFE OF A VARIABLE

- ▷ A block is determined by enclosing curly brackets
- ▷ A variable's lifetime is limited to the block where it was declared. The corresponding block defines the variable's scope
- ▷ The memory space dedicated to store a variable is allocated when this latter is declared, and released at the end of the block where it was allocated



C (gcc 4.8, C11)
(known limitations)

```c
#include <stdlib.h>

int main(void) {
    int a = 0;
    {
        float b;
        float c = 1;
        b = c;
    }
    a = 2;
    return EXIT_SUCCESS;
}
```

Stack

main

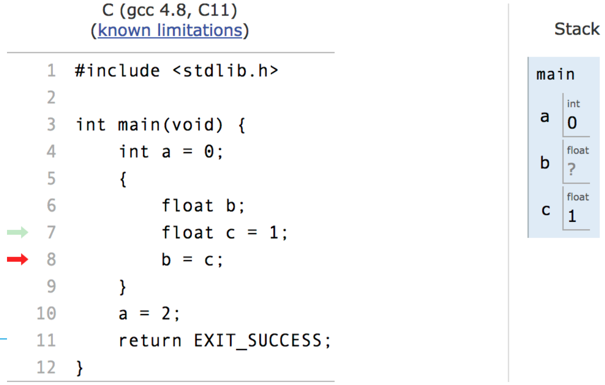
a int ?

BIRTH AND LIFE OF A VARIABLE

▷ A block is determined by enclosing curly brackets

▷ A variable's lifetime is limited to the block where it was declared. The corresponding block defines the variable's scope

▷ The memory space dedicated to store a variable is allocated when this latter is declared, and released at the end of the block where it was allocated



C (gcc 4.8, C11)
(known limitations)

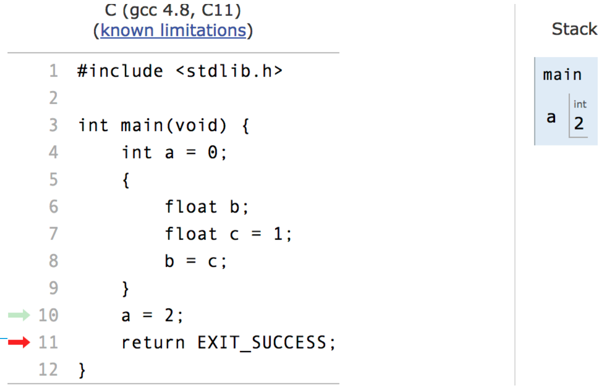
```
#include <stdlib.h>

int main(void) {
    int a = 0;
    {
        float b;
        float c = 1;
        b = c;
    }
    a = 2;
    return EXIT_SUCCESS;
}
```

BIRTH AND LIFE OF A VARIABLE

▷ A block is determined by enclosing curly brackets

▷ A variable's lifetime is limited to the block where it was declared. The corresponding block defines the variable's scope

▷ The memory space dedicated to store a variable is allocated when this latter is declared, and released at the end of the block where it was allocated

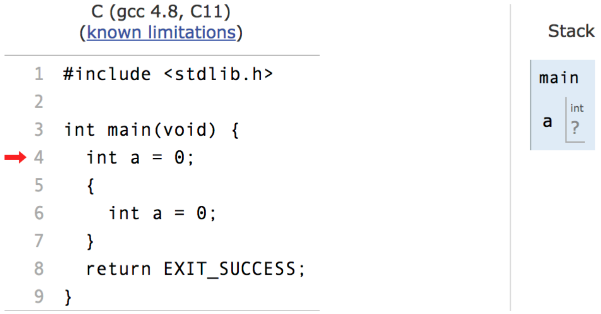


C (gcc 4.8, C11)
(known limitations)

```
#include <stdlib.h>

int main(void) {
    int a = 0;
    {
        float b;
        float c = 1;
        b = c;
    }
    a = 2;
    return EXIT_SUCCESS;
}
```

NAME MASKING - VARIABLE SHADOWING

▷ A local variable masks a variable having the same name that is declared in an outer scope

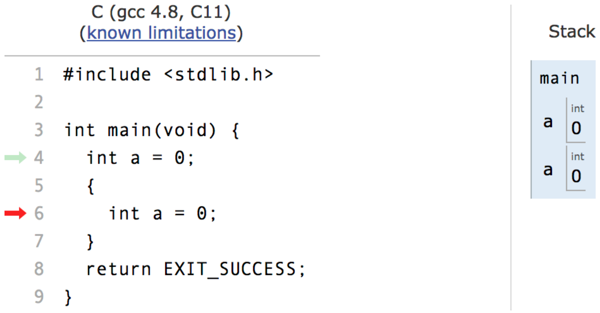


C (gcc 4.8, C11)
(known limitations)

```
#include <stdlib.h>

int main(void) {
  int a = 0;
  {
    int a = 0;
  }
  return EXIT_SUCCESS;
}
```

NAME MASKING - VARIABLE SHADOWING

▷ A local variable masks a variable having the same name that is declared in an outer scope

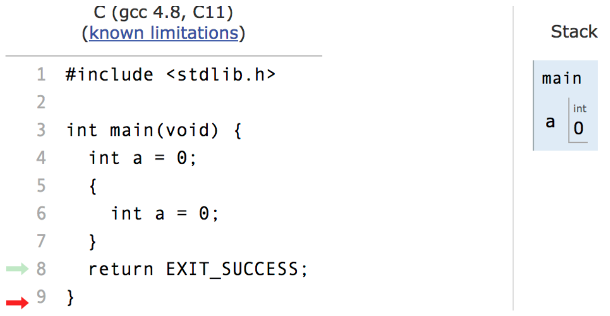


C (gcc 4.8, C11)
(known limitations)

```
#include <stdlib.h>

int main(void) {
  int a = 0;
  {
    int a = 0;
  }
  return EXIT_SUCCESS;
}
```

NAME MASKING - VARIABLE SHADOWING

▷ A local variable masks a variable having the same name that is declared in an outer scope

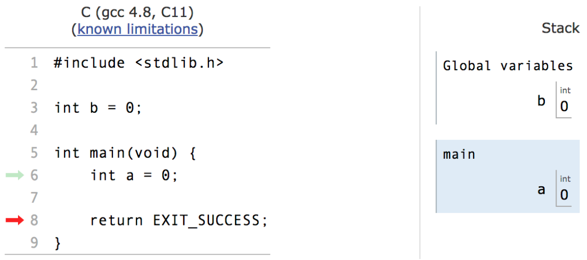


```
#include <stdlib.h>

int main(void) {
  int a = 0;
  {
    int a = 0;
  }
  return EXIT_SUCCESS;
}
```

GLOBAL VARIABLE

▷ A variable declared out of any block is called global and can be used all over the source file (should be avoided as much as possible)

▷ Global variables are permanent (their lifetime is that of the program)

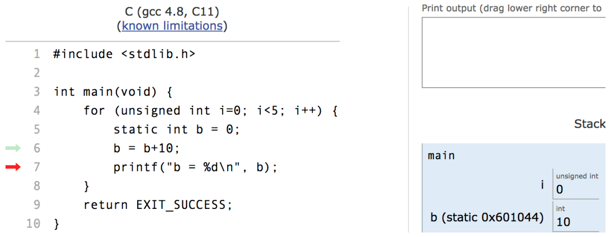


```
#include <stdlib.h>

int b = 0;

int main(void) {
    int a = 0;

    return EXIT_SUCCESS;
}
```

STATIC VARIABLE

▷ A static variable remains in memory while the program is running.

▷ Static variables have a property of preserving their value even after they are out of their scope

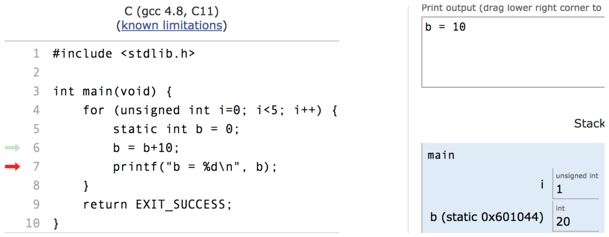


```
#include <stdlib.h>

int main(void) {
    for (unsigned int i=0; i<5; i++) {
        static int b = 0;
        b = b+10;
        printf("b = %d\n", b);
    }
    return EXIT_SUCCESS;
}
```

STATIC VARIABLE

▷ A static variable remains in memory while the program is running.

▷ Static variables have a property of preserving their value even after they are out of their scope



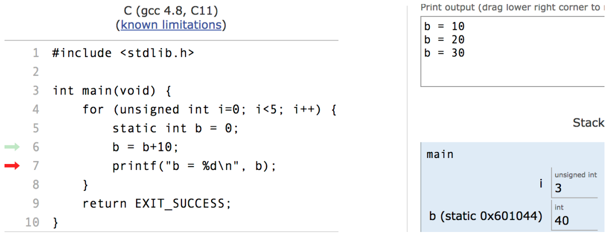
```
#include <stdlib.h>

int main(void) {
    for (unsigned int i=0; i<5; i++) {
        static int b = 0;
        b = b+10;
        printf("b = %d\n", b);
    }
    return EXIT_SUCCESS;
}
```

```
b = 10
```

STATIC VARIABLE

▷ A static variable remains in memory while the program is running.

▷ Static variables have a property of preserving their value even after they are out of their scope



C (gcc 4.8, C11)
(known limitations)

```c
#include <stdlib.h>

int main(void) {
    for (unsigned int i=0; i<5; i++) {
        static int b = 0;
        b = b+10;
        printf("b = %d\n", b);
    }
    return EXIT_SUCCESS;
}
```

Print output (drag lower right corner to

```
b = 10
b = 20
b = 30
```

NAME MASKING - VARIABLE SHADOWING

What is the output of this program ?

```c
#include <stdio.h>
#include <stdlib.h>
int a=5, b=12;
int main(void)
{
    int a=3, i=0;
    printf("%d",a);
    for(i=0; i<10; i=i+1){
        int a=4;
        printf("%d",a);
    }
    printf("%d",b);
    return EXIT_SUCCESS;
}
```

FUNCTIONS

DEFINITION

▷ The function is the fundamental programming unit of the C language

▷ A function is often defined to perform a single task, and its name should reflect it

▷ A function allows one to factorize code

▷ A function contains declarations and instructions

DECLARING A FUNCTION

▷ A function is characterized by its prototype (or signature):

▷ A name

▷ A (possibly empty) list of parameters (corresponding to local variables within the function)

▷ A return data type

```
return_type name (parameters);
float max (float a, float b);
```

VOID

▷ void is a special data type

▷ As a return data type, it indicates that there is no returned value

```
void print_sum (int a, int b) {
    ...
}
```

▷ As a parameter, it indicates that there is no parameter

```
int get_day (void) {
...
}
```

DEFINITION VS DECLARATION

▷ Definition = code of the function

▷ Declaration = only its prototype, followed by ';' (required before any use)

▷ A definition implies a declaration

```
void foo (void);

void bar (void) {
    ...
    foo ();
    ...
}

void foo (void) {
    ...
    bar ();
    ...
}
```

RETURNING A VALUE

▷ Instruction **return** allows one to quit the function immediately, irrespective of the position of the **return** within the function

▷ Returns the value yielded by an expression if the return data type is not void

▷ Mandatory if the return data type is not void

```
void print_sum (float a, float b) {
    printf ("%f + %f = %f\n", a, b, a + b);
}
...
float print_and_return_sum (float a, float b) {
    printf ("%f + %f = %f\n", a, b, a + b);
    return a + b;
}
```

RETURNED VALUE

▷ One may ignore the value returned by a function

▷ One cannot use in an expression a function that does not return a result

```
float print_and_return_sum (float a, float b) {
    printf ("%f + %f = %f\n", a, b, a + b);
    return a + b;
}
int main (void) {
    float sum = 0.0f;
    print_and_return_sum (3.5f, 1.1f);
    sum = print_and_return_sum (3.5f, 1.1f);
    printf ("%f", sum);
    return EXIT_SUCCESS;
}
```

FUNCTION CALL

▷ A function call is an expression which transfers control to the first instruction of the function definition, and passes arguments to it (if any). It is represented as

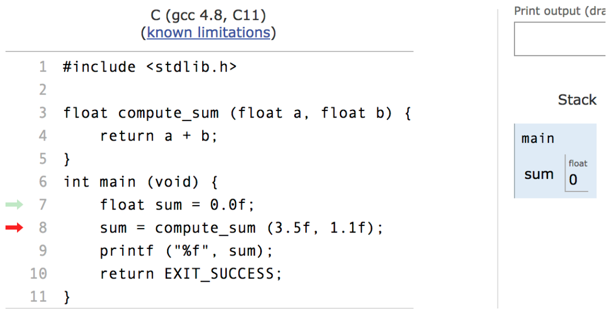
`function_name (expressions_list)`

where `function_name` is a declared function name and `expressions_list` is a list of expressions (separated by commas)

`max (5, 6)`

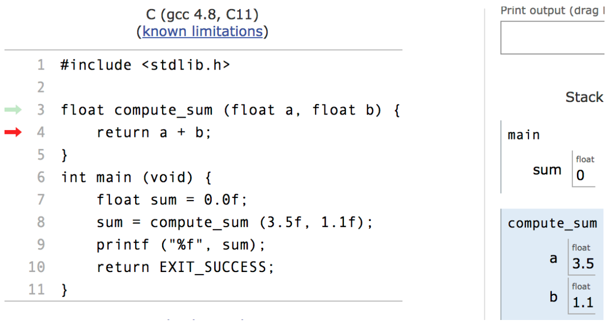
▷ The values of expressions belonging to `expressions_list` are the arguments passed to the function

FUNCTION CALL



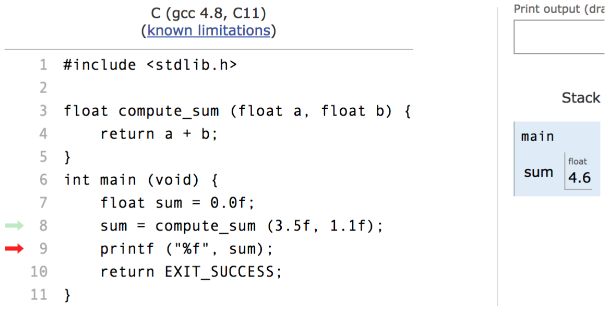
```
#include <stdlib.h>

float compute_sum (float a, float b) {
    return a + b;
}
int main (void) {
    float sum = 0.0f;
    sum = compute_sum (3.5f, 1.1f);
    printf ("%f", sum);
    return EXIT_SUCCESS;
}
```

FUNCTION CALL



```
#include <stdlib.h>

float compute_sum (float a, float b) {
    return a + b;
}
int main (void) {
    float sum = 0.0f;
    sum = compute_sum (3.5f, 1.1f);
    printf ("%f", sum);
    return EXIT_SUCCESS;
}
```

FUNCTION CALL



```c
#include <stdlib.h>

float compute_sum (float a, float b) {
    return a + b;
}
int main (void) {
    float sum = 0.0f;
    sum = compute_sum (3.5f, 1.1f);
    printf ("%f", sum);
    return EXIT_SUCCESS;
}
```

PARAMETER PASSING: ONLY BY VALUE

- ▷ All arguments are passed as values
- ▷ It means that only the values of the expressions are provided to the function
- ▷ The argument values may be converted to match parameter data types
- ▷ The function does not know the origin (i.e., memory location) of the value provided as parameter
- ▷ The function uses the values without any possible direct side effect on the expression at the origin of the value

FUNCTION CALL

```c
int max (int a, int b) {
//1st call: a = 3 and b = 4
//2nd call: a = 1 and b = 4
    if (a < b) {
        return b;
    }
    return a;
}
int main (void) {
    int a = 1, x = 3, y = 4, m = 0;
    m = max (x, y); // equivalent to m = max (3, 4)
    printf (”max(%d,%d)=%d”, x, y, m);
    printf (”max(%d,%d)=%d”, a, y, max (a, y));
}
```

DEFINING A FUNCTION

WRITING A FUNCTION

- ▷ One has to determine the use of the function
- ▷ A function should correspond to a single task
  - ▷ E.g., one should not mix computation and display

```
int minimum (int a, int b) {
    int min = b;
    if (a < b) {
        min = a;
    }
    printf ("minimum = %d\n", min); //TO AVOID !
    return min;
}
```

  - ▷ Why?

DEFINING THE PROTOTYPE

- ▷ What is needed by the function?
  - ▷ Parameters
- ▷ Does it return something?
- ▷ Are there cases of error?
- ▷ If so, there are 3 solutions:
  - ▷ Set a comment stating the allowed cases
  - ▷ Return an error code
  - ▷ Display a message and quit the program

COMMENT

- ▷ A comment is useful to state the forbidden cases

```
/**
* Copies the array 'src' into the 'dst' one.
* 'dst' is supposed to be large enough.
*/
void copy (int src[], int dst[]);
```

- ▷ But it does not prevent the user from disregarding the advice

ERROR CODE

- ▷ It is possible to return an error code if the function was not supposed to return any result
- ▷ ⚠Otherwise, one must always ensure that the error code can be distinguished from a normal result (be careful zbout the chosen value)

ERROR CODE

```
#define ERROR_CODE -1
int minimum(int t[], int size){
    if(size<=0){
        return ERROR_CODE;
    }
    int min=t[0];
    int i;
    for(i=1; i<size; i=i+1){
        if(min<t[i]){
            min=t[i];
        }
    }
    return min;
}
```

▷ We cannot know if there is an error or if the minimum is -1

ERROR CODE

```
 #define ERROR_CODE -1
/**
* Returns the length of the given
* string or ERROR_CODE if NULL.
*/
int length(char* s){
    if(s==NULL){
        return ERROR_CODE;
    }
    int i;
    for(i=0; s[i]!='\0'; i=i+1);
    return i;
}
```

▷ the length of a string cannot be negative

ERROR CODE

▷ if all values are taken, use a pointer for the result of the error code

```
#define ERROR_CODE 0
#define SUCCESS_CODE 1
int quotient(int a, int b, int * res){
    if(b==0){
        return ERROR_CODE;
    }
    *res=a/b;
    return SUCCESS_CODE;
}
```

SERIOUS ERRORS

▷ In case of a serious error, one can interrupt the program

▷ One should always quit a function as soon as possible by treating the error cases in the first place

▷ the `assert` function declared in `assert.h` which will stop the program if the condition is not valid. It can be disabled while compiling the code.

```
#include <assert.h>
void foo(char *ptr, int min, int max) {
    assert(ptr); // the pointeur must not be NULL
    assert(min <= max); // min must not be greater than max
    // ...
}
```

SERIOUS ERRORS

▷ You can also define your own ”assert” function which cannot be disabled while compiling. You shall use the exit function declared in stdlib.h and which takes as parameters EXIT_FAILURE or EXIT_SUCCESS.

```
void exit_if(int condition, const char *comment) {
    if (condition) {
        fprintf (stderr, comment);
        exit(EXIT_FAILURE);
    }
}

int foo(char *ptr){
    //...
    exit_if(ptr == NULL, ”A fatal error occurred”);
    //...
}
```

COMMAND LINE PROCESSING

COMMAND LINE PROCESSING

▷ Command line processing : \$ command arg_1 arg_2...

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

▷ argc : array size (number of arguments of the program)

▷ argv: array of arguments (char* means string of chars)

▷ argv[0] is always the program name (so argc ≥ 1)

COMMAND LINE PROCESSING

▷ Command line processing : \$ command arg_1 arg_2...

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

▷ Arguments are passed as strings

▷ In order to use one of the main parameters as a numeric value, one should first convert it

▷ The standard library provides some useful conversion functions (declared in stdlib.h)

```
 double atof (const char * s);
int atoi (const char * s);
long atol (const char * s);

long strtol (const char *str, char **endptr, int base);
unsigned long strtoul (const char *str, char **endptr, int base);
double strtod (const char *str, char **endptr);
```

COMMAND LINE PROCESSING

Example :

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

int main(int argc, char *argv[])
{
    if (argc != 2)
        return EXIT_FAILURE;

    int v = atoi(argv[1]);
    if (v % 2 == 0)
        return EXIT_SUCCESS;
    return EXIT_FAILURE;
}
```

→ More details in future classes

DOGGY BAG

TO TAKE AWAY ...

- ▷ Any variable has a lifetime bounded to the block where it is defined
- ▷ One should avoid global variables
- ▷ A function must be declared or defined before its first use
- ▷ All the arguments and the returned value of a function are results of expression evaluation
- ▷ One should always be able to distinguish a normal result from an error case

QUESTIONS?