COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2023-January 2024

UNIVERSITÀ DEGLI STUDI DI PADOVA

Miscellaneous

Università degli Studi di Padova

Miscellaneous

- External variables and scope
- Variables specific declarations
- Preprocessor directives
- Block structure
- Inizialization vs. declaration
- Recursion

Local variables in functions

In C programming, variables can have different scopes depending on where they are declared/defined. The two primary types of variable scopes are global and local. Additionally, the **extern** keyword is used to declare external variables.

- Local variables are declared within a block or a function and are only accessible within that block or function. They are not visible to other functions or outside the block in which they are declared.
  - Each local variable in a function comes into existence only when the function is called and disappears when the function is exited. Such variables are usually known as automatic variables.

Local Variables

Example of a local variable inside a function

```
int main() {
   // Local variable
   int localVariable = 10;
   // ...
   return 0;
}
```

Automatic variables come and go with function invocation, therefore, they do not retain their values from one call to the next and shall be explicitly set upon each entry. If they are not set, they will contain garbage.

Block scope

C supports block scope, where variables declared within a block of code (e.g., within loops or if statements) are only accessible within that specific block.

```
int main() {
    if (condition) {
        // Block scope
        int blockVariable = 20;
        // ...
    }
    // blockVariable is not accessible here
    return 0;
```

Global variables

As an alternative to automatic variables, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function.

- Global variables are declared outside any function, typically at the beginning of a C file or in a header file. They are accessible from any part of the program, including all functions.
- Global variables, being globally accessible, can be used instead of argument lists to communicate data between functions.
- Furthermore, because global variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them have returned.

Global variables

```
// Global variable
int globalVariable;

int main() {
    // Accessing the global variable
    globalVariable = 42;
    return 0;
}
```

A global (external) variable must be defined, **exactly once**, outside of any function; this sets aside storage for it. The variable must also be declared in each function that wants to access it; this states the type of the variable. The declaration may be an explicit **extern** statement or may be implicit from context.

Global variables

```
// Global variable
int globalVariable;

int main() {
    // Accessing the global variable
    globalVariable = 42;
    anyFunction();
  return 0;
}
void anyFunction() {
    extern int globalVariable;
    …
}
```

External variables

In certain circumstances, the extern declaration can be omitted.

- If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an extern declaration in the function.
- In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations.

But, if the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable.

External variables

External variables have external linkage, meaning their declarations are shared across multiple files. The actual variable is defined in one file (**without the extern keyword**), and its declaration (**with extern**) is included in other files.

```
// File: file1.c
int globalVariable;  // Definition

// File: file2.c
extern int globalVariable;  // Declaration

// File: file3.c
extern int globalVariable;  // Declaration
```

External variables

The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header, that is included by **#include** at the front of each source file. The suffix .h is conventional for header names.
External variables can be accessed in any file where their declaration (with extern) is visible. This allows different parts of a program to share and modify the same variable.

```
// File: main.c
extern int globalVariable;  // Declaration
int main() {
  // Accessing the global variable
  globalVariable = 100;
  return 0;
}
```

External variables: considerations

- Initialization:
  - External variables should be initialized in one source file, and their declarations (with extern) should be included in other files where they are used.

- Order of Declarations:
  - Declarations shall appear before the variable is used.

- **Avoid Global Variables When Possible**:
  - While global variables and external variables provide a means for sharing data, excessive use can lead to code that is harder to maintain and reason about. It's often preferable to pass data between functions explicitly.

External variables: considerations

- External variables are always there even when you don't want them. Relying too heavily on external variables leads to programs whose data connections are not all obvious - variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify.

You should have noted that we have used the words definition and declaration when referring to external variables.``**Definition**'' refers to the place where the variable is created or assigned storage. ``**Declaration**'' refers to places where the nature of the variable is stated but no storage is allocated.

External variables: scope rules

The scope of a name is the part of the program within which the name can be used.

- For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared.
- Local variables of the same name in different functions are unrelated.
- The same is true of the parameters of the function, which are in effect local variables.
- The scope of an external variable or a function goes from the point at which it is declared to the end of the file being compiled.
- On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an extern declaration is mandatory.

External variables: scope rules

- It is important to distinguish between the declaration of an external variable and its definition.
- A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside.
- There must be only one definition of an external variable among all the files that make up the source program; other files may contain extern declarations to access it. (There may also be extern declarations in the file containing the definition).
- Array sizes must be specified with the definition but are optional with an extern declaration.
- Initialization of an external variable goes only with the definition.

Header files

There is one thing to worry about - the declarations shared among files.

As much as possible, it is preferable to centralize this stuff, so that there is only one copy to get and keep updated as the program evolves.

We will place this common material in a header file, which will be included as necessary by using the #include directive.

As far as possible put variables and functions declarations in header files to be included in your program source files.

External variables: summary

External variables are defined outside of any function and are thus available to many functions. **Functions themselves are always external**, because C does not allow functions to be defined inside other functions. By default, external variables and functions have the property that all references to them by the same name are references to the same thing. (The standard calls this property **external linkage**). Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow. If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists.

Static variables

In C programming, there are other types of variables beyond the typical automatic variables (local variables). These include static, register and constants variables.

In C programming, the **static** keyword is used to declare static variables. Static variables have a different behavior compared to regular, automatic variables. Which are the key characteristics and use cases for static variables?

- **Preservation of Value**
- **Lifetime**
- **Internal Linkage**

Static variables key characteristics

Preservation of Value:
Static variables retain their values between function calls. The value is initialized only once, and subsequent calls to the function will use the updated value.

Lifetime:
Static variables have a lifetime equal to the entire duration of the program. They are created when the program starts and exist until the program terminates.

Internal Linkage:
By default, static variables have internal linkage, meaning they are visible only within the same source file. If declared outside any function, they are also restricted to the file they are declared in.

Static variables key characteristics

```
#include <stdio.h>

void exampleFunction() {
   // Static variable with internal linkage
   static int staticVar = 0;

   // Increment and print the static variable
   staticVar++;
   printf("Static Variable: %d\n", staticVar);
}

int main() {
   // Call the function multiple times
   exampleFunction();
   exampleFunction();
   exampleFunction();

   return 0;
}
```

```
Static Variable: 1
Static Variable: 2
Static Variable: 3
```

Static variables use cases

Maintaining State:
Static variables are often used to maintain state across multiple calls to the same function. For example, a counter in a function that needs to remember its value between calls.
Avoiding Name Clashes:
When declared with internal linkage, static variables can be used to avoid naming conflicts between variables in different files. Each file can have its own static variable with the same name.
Initialization Control:
Static variables can be used to control the initialization of a variable. For instance, initializing a variable only once during the first call to a function.

Static variables external linkage

If you want a static variable to have external linkage (i.e., be visible across multiple source files), you can use the extern keyword in the declaration. This will work well if the declaration is in a header file that is included in multiple source files.

```
// File: example.h
extern int globalStaticVar;
```

```
// File: example.c
#include "example.h"
static int globalStaticVar = 0;

void exampleFunction() {
    globalStaticVar++;
    printf("Global Static Variable: %d\n", globalStaticVar);
}
```

In this example, globalStaticVar has external linkage and is shared across multiple source files.

Register variables

UNIVERSITÀ DEGLI STUDI DI PADOVA

In C programming, the **register** keyword is used to declare register variables. Register variables are a way to provide a hint to the compiler that a particular variable should be stored in a register for faster access.

Which are the key characteristics and use cases for register variables?

- **Storage in CPU Register**
- **Compiler Discretion**
- **Limited Use Cases**
- **No Direct Memory Access**

Register variables key characteristics

Storage in CPU Register
The register keyword suggests to the compiler that the variable should be stored in a CPU register rather than in memory. Registers are faster to access than memory locations.

Compiler Discretion
The register keyword is a hint, and the compiler is not obligated to honor it. Modern compilers are often capable of making intelligent decisions about register allocation without explicit hints from the programmer.

Register variables key characteristics

Limited Use Cases
The effectiveness of register variables is often dependent on the architecture and optimization capabilities of the compiler. They are most beneficial for frequently accessed variables in tight loops or critical sections where speed is crucial.

No Direct Memory Access
Register variables cannot be directly accessed using the address-of (&) operator, as they are intended to be stored in registers rather than memory. Attempting to take the address of a register variable may result in a compilation error.

Register variables use cases

Performance Optimization
Register variables are used in performance-critical sections of code where reducing memory access times is crucial for improving execution speed.

Frequent Access
Variables that are accessed frequently in loops or computations may benefit from being declared as register variables.

Register variables example

```
#include <stdio.h>

void computeSum() {
   // Register variables for faster access
   register int i, sum = 0;

   // Loop to compute the sum
   for (i = 1; i <= 100; ++i) {
      sum += i;
   }

   printf("Sum: %d\n", sum);
}

int main() {
   // Call the function
   computeSum();

   return 0;
}
```

In this example, the i and sum variables are declared as register variables within the computeSum function. The compiler may choose to store these variables in registers for faster access during the loop.

Register variables usage considerations

Compiler Optimization

Modern compilers often perform sophisticated optimizations, and the use of the register keyword may not significantly impact performance. It's advisable to rely on compiler optimizations and profile your code to identify actual bottlenecks.

Compiler Warnings

Some compilers may issue a warning if they choose to ignore the register keyword. Always pay attention to compiler warnings and consider alternatives if necessary.

Register variables usage considerations

Use with Caution
Overusing register variables or using them inappropriately may hinder the compiler's ability to optimize code. It's important to profile and test your code to ensure that the use of register variables provides a measurable performance improvement.

In summary, register variables in C are a way to suggest to the compiler that certain variables should be stored in registers for faster access. However, their impact on performance can vary, and reliance on compiler optimizations is often sufficient for achieving good performance in modern C programs.

Constant variables

In C programming, constant variables are declared using the **const** keyword. Constant variables represent values that should not be modified during the program's execution.

Which are the key characteristics and use cases for constant variables?

- **Immutability**
- **Initialization at Declaration**

Constant variables key characteristics

Immutability:
Constant variables are read-only, meaning their values cannot be modified after initialization. Any attempt to modify a constant variable will result in a compilation error.

Initialization at Declaration:
Constant variables must be initialized at the time of declaration. Once initialized, their values cannot be changed.

Constant variables use cases

Symbolic Constants:
Constants are often used to define symbolic names for values that remain constant throughout the program. This improves code readability and makes it easier to maintain.

Preventing Unintended Modifications:
Constant variables help prevent unintentional modifications of values. By marking a variable as constant, you signal your intent that the value should not be changed.

Constant variables use cases

Improved Code Understanding:
Constants with meaningful names enhance code understanding. For example, using *const int DAYS_IN_WEEK = 7;* makes the code more readable than using the literal value 7 throughout the program.

Global Constant Variables:
Constants can be declared globally, making them accessible across multiple functions or translation units.

Constant variables use cases

```
// Global constant variable
const double PI = 3.14159;

void printCircleArea(double radius) {
    // Using the global constant variable
    double area = PI * radius * radius;
    printf("Circle Area: %f\n", area);
}

int main() {
    // Calling a function that uses the global constant
    printCircleArea(5.0);

    return 0;
}
```

Constant variables in enumeration

Enumerations provide a way to create a set of named integer constants. Enumerated constants are also known as enumerators. In the following example, SUNDAY, MONDAY, etc., are constants within the enumeration Days.

```
#include <stdio.h>

// Enumeration with named constants
enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };

int main() {
   // Using enumerated constants
   enum Days today = TUESDAY;
   printf("Today is %d\n", today);

   return 0;
}
```

Constant variables considerations

Initialization Requirement:
Constant variables must be initialized at the time of declaration. Unlike regular variables, you cannot assign a value to a constant variable later in the program.

Compile-Time Constants:
Constant variables are evaluated at compile time. This means that the compiler knows their values during the compilation process.

Use of #define:
In addition to const, constants can also be defined using preprocessor directives, such as #define. However, using const is generally preferred as it provides type checking.

Difference between #define and const in C?

#define is a preprocessor directive. Data defined by the #define macro definition are preprocessed, so that your entire code can use it. This can free up space and increase compilation times.
const variables are considered variables, and not macro definitions.
In short, **CONSTs are handled by the compiler**, **#DEFINEs are handled by the pre-processor**.

The big advantage of const over #define is type checking. #defines can’t be type checked, so this can cause problems when trying to determine the data type. If the variable is, instead, a constant then we can grab the type of the data that is stored in that constant variable.

Difference between #define and const in C?

#define	const
#define is a preprocessor directive.	Constants are used to make variables constant such that never change during execution once defined.
Is used to define a substitution.	The constant variable value is used in the program.
Its syntax is -: **#define token value**	Its syntax is -: **const type constant_name;**
It shall not be terminated with a (;) semicolon	It shall be terminated with a (;) semicolon

Difference between #define and const in C?

Since const are considered variables, we can use pointers on them. This means we can typecast, move addresses, and everything else you'd be able to do with a regular variable besides change the data itself, since the data assigned to that variable is constant.

In general, const is a better option if we have a choice and it can successfully apply to the code. There are situations when #define cannot be replaced by const. For example, #define can take parameters (the parameters are not checked for data type). #define can also be used to replace some text in a program with another text.

#define parameters usage

For example, the following macro INCREMENT(x) can be used for x of any data type.

```
#include <stdio.h>
#define INCREMENT(x) ++x
int main()
{
    char* ptr = “Example";
    int x = 10;
    printf("%s ", INCREMENT(ptr));
    printf("%d", INCREMENT(x));
    return 0;
}
```

Output:

```
xample 11
```

#define parameters usage

The macro arguments are not evaluated before macro expansion.

```
#include <stdio.h>
#define MULTIPLY(a, b) a* b
int main()
{
    // The macro is expanded as 2 + 3 * 3 + 5, not as 5*8
    printf("%d", MULTIPLY(2 + 3, 3 + 5));
    return 0;
}
```

Output:

```
16
```

#define parameters usage

The previous problem can be solved using parenthesis.

```
#include <stdio.h>
// here, instead of writing a*a we write (a)*(b)
#define MULTIPLY(a, b) (a) * (b)
int main()
{
    // The macro is expanded as (2 + 3) * (3 + 5), as 5*8
    printf("%d", MULTIPLY(2 + 3, 3 + 5));
    return 0;
}
```

Output:

```
40
```

#define parameters usage

```
#include <stdio.h>

#define square(x) x* x
int main()
{
        // Expanded as 36/6*6
        int x = 36 / square(6);
        printf("%d", x);
        return 0;
}
```

Output:

```
36
```

```
#include <stdio.h>

#define square(x) (x * x)
int main()
{
        // Expanded as 36/(6*6)
        int x = 36 / square(6);
        printf("%d", x);
        return 0;
}
```

Output:

```
1
```

#define parameters usage

The parameters passed to macros can be concatenated using operator **##** called Token-Pasting operator.

```
    #include <stdio.h>
    #define merge(a, b) a##b
    int main()
    {
        printf("%d ", merge(12, 34));
    }
```

Output:

```
1234
```

A token passed to macro can be converted to a string literal by using **#** before it.

```
#include <stdio.h>
#define get(a) #a
int main()
{
        // Example is changed to “Example"
        printf("%s", get(Example));
}
```

Output:

```
Example
```

#define usage

The macros can be written in multiple lines using '\'. The last line doesn't have '\'.

```
#include <stdio.h>
#define PRINT(i, limit)                                      \
    while (i < limit) {                                      \
            printf("Example ");                              \
            i++;                                             \
    }
int main()
{
    int i = 0;
    PRINT(i, 3);
    return 0;
}
```

Output:

```
Example Example Example
```

#define usage

Preprocessors also support **if-else directives** which are typically used for conditional compilation.

```
int main()
{
#if VERBOSE >= 2
    printf("Trace Message");
#endif
}
```

A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like **#define**, **#ifdef** and **#ifndef** are used.

#define usage

```
#include <stdio.h>

// Uncomment one of the following lines to enable or disable DEBUG mode
#define DEBUG

#ifdef DEBUG
   #define DEBUG_PRINT(msg) printf("Debug: %s\n", msg)
#else
   #define DEBUG_PRINT(msg)
#endif

#ifndef RELEASE
   #define RELEASE
#endif

int main() {
   DEBUG_PRINT("This is a debug message.");

#ifdef RELEASE
   printf("Release mode: This code is included.\n");
#else
   printf("Release mode: This code is excluded.\n");
#endif

   return 0;
}
```

Output:

```
Debug: This is a debug message.
Release mode: This code is included.
```

In this example:

DEBUG can be defined to enable debug mode (printing debug messages).
RELEASE is conditionally defined using #ifndef, making it defined if not already defined.
You can experiment with commenting or uncommenting the #define DEBUG line to observe the changes in the output.

These directives are commonly used to create flexible and maintainable code by allowing conditional inclusion or exclusion of specific code sections based on compile-time configurations.

#define usage

There are some standard macros which can be used to print program file (__FILE__), Date of compilation (__DATE__), Time of compilation (__TIME__) and Line Number in C code (__LINE__)

```
#include <stdio.h>

int main()
{
        printf("Current File :%s\n", __FILE__);
        printf("Current Date :%s\n", __DATE__);
        printf("Current Time :%s\n", __TIME__);
        printf("Line Number :%d\n", __LINE__);
        return 0;
}
```

Output:

```
Current File :C:\myProgramFolder\myProgram.cpp
Current Date :Nov 12 2023
Current Time :19:11:41
Line Number :8
```

#define usage

We can remove already defined macros using : **#undef MACRO_NAME**

```
#include <stdio.h>
// div function prototype
float div(float, float);
#define div(x, y) x / y

int main()
{
		// use of macro div
		// Note: %0.2f for taking two decimal value after point
		printf("%0.2f", div(10.0, 5.0));
// removing defined macro div
#undef div
		// function div is called as macro definition is removed
		printf("\n%0.2f", div(10.0, 5.0));
		return 0;
}

// div function definition
float div(float x, float y) { return y / x; }
```

Block structure

C is not a block-structured language, because functions may not be defined within other functions. On the other hand, variables can be defined in a block-structured fashion within a function.

Declarations of variables (including initializations) may follow the left brace that introduces any compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks and remain in existence until the matching right brace.

An automatic variable declared and initialized in a block is initialized each time the block is entered

Block structure

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declarations

```
int x;
int y;
f(double x)
{
    double y;
}
```

then within the function f, occurrences of x refer to the parameter, which is a double; outside f, they refer to the external int. The same is true of the variable y. As a matter of style, it's best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great.

Inizialization

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, before the program begins execution.

Inizialization

For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

instead of

```
int low, high, mid;
low = 0;
high = n - 1;
```

Inizialization

Initialization of automatic variables are just shorthand for assignment statements.
Explicit assignments are more easily understood because initializers in declarations are harder to see.
An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

For example, to initialize an array days with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this example.

Inizialization

If there are fewer initializers for an array than the specified size, the others will be zero for external, static and automatic variables.
It is an error to have too many initializers.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

char pattern = “hello";

is a shorthand for the longer but equivalent

char pattern[] = { ‘h', ‘e', 'l', ‘l’, ‘o’ , '\0' };

In this case, the array size is six (five characters plus the terminating '\0').

Recursion

Università degli Studi di Padova

C functions may be used recursively; that is, a function may call itself either directly or indirectly.

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster.

But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent.

Recursion is especially convenient for recursively defined data structures like trees, that we will see later in the course.

Let’s see some typical examples of recursive functions in C.

Factorial Calculation

```
#include <stdio.h>

// Recursive factorial function
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0 || n == 1) {
        return 1;
    } else {
        // Recursive case: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}

int main() {
    int number = 5;
    printf("Factorial of %d is %d\n", number, factorial(number));
    return 0;
}
```

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n.

Fibonacci Sequence

```
#include <stdio.h>

// Recursive Fibonacci function
int fibonacci(int n) {
   // Base case: Fibonacci of 0 is 0, and Fibonacci of 1 is 1
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      // Recursive case: F(n) = F(n-1) + F(n-2)
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
}

int main() {
   int term = 6;
   printf("Fibonacci term at position %d is %d\n", term, fibonacci(term));
   return 0;
}
```

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones.

Binary Search

Binary search is an algorithm that finds the position of a target value within a sorted array.

```
#include <stdio.h>

// Recursive binary search function
int binarySearch(int arr[], int low, int high, int target) {
    if (low <= high) {
        int mid = low + (high - low) / 2;

        // Base case: target found at mid
        if (arr[mid] == target) {
            return mid;
        }

        // Recursive case: search in the left or right half
        if (arr[mid] > target) {
            return binarySearch(arr, low, mid - 1, target);
        } else {
            return binarySearch(arr, mid + 1, high, target);
        }
    }

    // Base case: target not found
    return -1;
}
```

```
int main() {
    int sortedArray[] = {2, 4, 6, 8, 10, 12, 14, 16};
    int target = 10;
    int size = sizeof(sortedArray) / sizeof(sortedArray[0]);

    int result = binarySearch(sortedArray, 0, size - 1, target);

    if (result != -1) {
        printf("Target %d found at position %d\n", target, result);
    } else {
        printf("Target %d not found in the array\n", target);
    }

    return 0;
}
```