COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Getting started - end

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Recap basic sintax rules and some new ones

- Multiple Conditions
- Logical operators
- Exercise
- Data Types

Multiple Conditions

Write a C program that determines whether a given year is a leap year. The program shall:

Prompt the user to enter a year.
Check if the year is divisible by 4 using the equality operator. If it is, proceed to the next step.
Check if the year is not divisible by 100 using the inequality operator or if it is divisible by 400 using the equality operator. If either condition is true, it's a leap year.
Display a message indicating whether the year is a leap year or not.

Logical operators

In C programming, logical operators are used to perform logical operations on boolean values (true or false) or to make decisions based on multiple conditions. There are three main logical operators in C: **&& (logical AND)**, **|| (logical OR)**, and **! (logical NOT)**.

Logical operators

Logical AND (**&&**):

The logical AND operator (**&&**) combines two conditions and evaluates to true only if both conditions are true.

If both conditions are true, the result is true. If either or both of the conditions are false, the result is false.

```
int x = 5;
int y = 10;
if (x > 0 && y < 20) {
    // This condition is true because both x > 0 and y < 20 are true.
}
```

Logical operators

Logical OR (||):

The logical OR operator (||) combines two conditions and evaluates to true if at least one of the conditions is true.

If either condition is true, the result is true. The result is false only if both conditions are false.

```
int a = 15;
int b = 7;
if (a > 10 || b < 5) {
    // This condition is true because a > 10 is true, even though b < 5 is false.
}
```

Logical operators

Logical NOT (!):

The logical NOT operator (!) is a unary operator that negates the value of a single condition.

If a condition is true, ! makes it false, and if it's false, ! makes it true.

```
int flag = 1;
if (!flag) {
    // Code block executed if flag is false
   // This condition is false because !flag is equivalent to false.
}
bool isAvailable = false;
if (!isAvailable) {
   cout << "Item is not available";
}
```

Logical operators

Order of Evaluation:
C follows a short-circuit evaluation for logical operators. In &&, if the first condition is false, the second condition is not evaluated, because the overall result will be false regardless of the second condition. In ||, if the first condition is true, the second condition is not evaluated, because the overall result will be true regardless of the second condition.

Complex Conditions:
You can use logical operators to create complex conditions by combining multiple subconditions. Parentheses are often used to clarify the order of evaluation.

Logical operators


```
int m = 3;
int n = 7;
if ((m > 0 && n > 5) || (m < 0 && n < 10)) {
   // This complex condition is true if either of the two sets of conditions is
true.
}
```

Practical Use Cases:

Logical operators are used for decision-making in conditional statements (if, else, switch) and loops (while, for).

They are also used for combining conditions in search queries, validation checks, and state machines, among other applications.

Multiple Conditions


```c
#include <stdio.h>

int main() {
    int year;

    printf("Enter a year: ");
    scanf("%d", &year);

    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }

    return 0;
}
```

Exercise

Write a C program that performs currency conversion from US Dollars (USD) to Euros (EUR) and British Pounds (GBP). The program should:

considering the amounts 1,..,100 USD calculate the equivalent amounts in EUR and GBP using the following exchange rates:
1 USD = 0.91 EUR
1 USD = 0.76 GBP
Display a table with the amounts in USD and the converted amounts in EUR and GBP until the amount in GBP exceeds the value 60.
Solve the exercise using the for loop and then one between the while and do-while loop.
Use macro #define to set lower and upper values needed in your program.

Data Types

Data types in C are a fundamental concept that specifies the type of data that a variable can hold. C is a statically typed language, which means that variables must be declared with their data type before they can be used. The choice of data type determines the range of values a variable can store and the operations that can be performed on it.
C provides several built-in data types, categorized into the following groups:

1. **Basic Data Types**
2. **Derived Data Types**
3. **Enumeration Data Types**
4. **User-Defined Data Types**
5. **Void Data Type**

Basic Data Types

- **int**: Represents integer values, both positive and negative. It is used to store whole numbers (positive or negative) without decimal points. The size of an int can vary depending on the compiler and architecture but is typically 4 bytes on most systems. Assuming 32-bit int, the range of values that an int can hold is from -2,147,483,648 to 2,147,483,647.

- **short**: Represents a smaller integer. It is typically 2 bytes in size. The range of values a short can hold is smaller than that of an int.

- **long**: Represents a larger integer than int. It is typically 4 or 8 bytes in size, depending on the platform (32-bit or 64-bit). The range of values a long can hold is much larger than that of an int.

Basic Data Types

- **float**: Represents single-precision floating-point numbers, which are used to store real numbers with decimal points. It's typically 4 bytes in size. It has limited precision, which means it may not represent some numbers with complete accuracy.

- **double**: Represents double-precision floating-point numbers, providing more significant digits than float. It's typically 8 bytes in size, offering more significant digits and a larger range of representable values than the float data type. double is commonly used when higher accuracy in floating-point calculations is required.

Basic Data Types

Data Type	Size (in bytes)	Range
int	4	-2,147,483,648 to 2,147,483,647
short	2	-32,768 to 32,767
long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Data Type	Size (in bytes)	Precision
float	4	6–7 decimal digits
double	8	15–16 decimal digits
long double	8, 12, 16	Typically more precision than double

Basic Data Types: char

- **char**: Represents a single character and is used to store characters like letters, digits, symbols and control characters. It's typically 1 byte in size, therefore it can represent 256 different values (2^8), which can cover the ASCII character set.
  - Internally, characters are represented as numeric values using character encodings such as ASCII (American Standard Code for Information Interchange) or UTF-8 (Unicode Transformation Format-8). In ASCII, for example, the character 'A' is represented as the numeric value 65, 'B' is 66, and so on.
  - To assign a character to a char variable, you can use single quotes. For example:

```
char myChar = 'A';
```

Basic Data Types: char

- char can also be used to store special characters using escape sequences, which are represented as backslashes followed by a character. For example:

  char newline = '\n'; // Represents a newline character

  char tab = '\t'; // Represents a tab character
- To assign a character to a char variable, you can use single quotes. For example:

  char myChar = 'A';
- You can perform various operations on char variables, such as comparison, addition, subtraction, and more. For example:

Basic Data Types: char


```
char a = 'A';
char b = 'B';
if (a < b) {
    // This condition will be true because 'A' comes before 'B' in ASCII.
}
```

- char is also commonly used to create character arrays (strings) in C. Strings in C are null-terminated arrays of char, where the null character ('\0') marks the end of the string.

The char data type in C is essential for working with characters and strings, and it's used extensively in input/output operations, text processing, and other tasks involving character-based data.

Derived Data Types

- **Arrays**: Collections of elements of the same data type, accessed by an index. For example, int scores[5] creates an array of 5 integers.

- **Pointers**: Variables that store memory addresses, allowing you to manipulate data indirectly. Pointers are used extensively in C for dynamic memory allocation and data structures.

Derived Data Types: array

- **Arrays**: Collections of elements of the same data type, accessed by an index.

  - Arrays in C are designed to store elements of the same data type. All elements within an array must be of the same data type, such as integers, characters, or floating-point numbers.

  - Arrays have a fixed size, which is determined when the array is declared. The size specifies how many elements the array can hold, and it cannot be changed during the program's execution. The size of an array is defined in square brackets [] following the array's name. For example, int numbers[5] declares an array of integers with a size of 5.

Derived Data Types: array

- In C, array elements are accessed using zero-based indexing. This means that the first element of an array is at index 0, the second element is at index 1, and so on. You can access elements using square brackets and the index, like this: *element = array[index];*

- Arrays can be declared and initialized at the same time. For instance:

  int numbers[5] = {1, 2, 3, 4, 5};

  You can also leave out the size when initializing an array, and the compiler will determine the size based on the number of elements provided:

  int numbers[] = {1, 2, 3, 4, 5}; // The size is automatically set to 5

Derived Data Types: array

- To access individual elements, you use the array name followed by the index in square brackets, like *numbers[2]* to access the third element of the array numbers. It's essential to ensure that the index is within the bounds of the array to avoid accessing memory outside the array, which can lead to undefined behavior.

- Loops like for and while are commonly used to iterate through arrays to perform operations on each element.

```
for (int i = 0; i < 5; i++) {
    printf("%d\n", numbers[i]);
}
```

Derived Data Types: array

- C allows you to create multidimensional arrays, which are arrays of arrays. For example, a 2D array can be thought of as a table with rows and columns.

- Arrays are often used to store strings in C. Strings in C are character arrays, typically terminated with a null character ('\0'), which marks the end of the string.

- C provides various library functions for manipulating arrays, such as *memcpy()*, *memset()*, and *sort()*, which can help you to perform common array operations efficiently.

Derived Data Types: array

In summary, arrays in C are a versatile and fundamental data type for storing collections of elements. They provide a convenient way to work with groups of related data and are used extensively in C programming for tasks ranging from simple data storage to complex data processing operations. Understanding array indexing and memory management is crucial to use arrays effectively and avoid common programming errors.

Derived Data Types: pointer

Pointers: In the C programming language, a pointer is a powerful and fundamental data type that allows you to store memory addresses and manipulate data indirectly. Pointers are a key feature of C, enabling dynamic memory allocation, efficient array manipulation, and direct access to hardware and other resources.

- Memory Addresses:
  - A pointer is a variable that stores the memory address of another variable or data object. It doesn't contain the actual data but points to the location where the data is stored in memory.

Derived Data Types: pointer

- Declaration:
  - Pointers are declared by specifying the data type they point to, followed by an asterisk (*). For example:

    *int *ptr; // Declares a pointer to an integer*

- Initialization:
  - Pointers can be initialized with the address of a variable using the address-of operator (&). For example:

    int value = 42;

    *int *ptr = &value; // 'ptr' now points to the memory location of 'value'*

Derived Data Types: pointer

- Pointer Arithmetic:
  - Pointers support arithmetic operations such as addition and subtraction. This is particularly useful when working with arrays and dynamic memory allocation. For instance, incrementing a pointer moves it to the next memory location for objects of its type.
- NULL Pointers:
  - Pointers can also be set to a special value called NULL to indicate that they don't currently point to any valid memory location. This is often used as a safety measure to avoid accessing uninitialized pointers.

Derived Data Types: pointer

- Dynamic Memory Allocation:
  - Pointers play a crucial role in dynamic memory allocation using functions like malloc(), calloc(), and realloc(). These functions allocate memory at runtime and return a pointer to the allocated memory.
- Function Pointers:
  - C allows you to define and use function pointers, which are pointers that point to functions instead of data. Function pointers are used for callbacks, dynamic function selection, and creating extensible software.

Derived Data Types: pointer

- Pointer to Structures and Arrays:
  - Pointers can be used to reference complex data structures like arrays and structures. This allows for efficient access to and manipulation of data elements within these structures.
- Pointer Safety and Pitfalls:
  - While pointers provide great flexibility, they can also introduce certain risks, such as null pointer dereferences, memory leaks, and dangling pointers. Careful management and good programming practices are essential to avoid these issues.

Derived Data Types: pointer

In summary, pointers are a fundamental concept in C, providing the ability to access and manipulate memory directly. They are widely used for various tasks, including dynamic memory management, data structures, and interacting with hardware. While they offer great power, they also require responsibility and care to use correctly and avoid common pitfalls.

Enumeration Data Types

- **enum**: In the C programming language, the enum (short for enumeration) statement allows you to create a user-defined data type consisting of a set of named integer constants. For instance, you can define an enumeration for days of the week. Enums provide a way to create symbolic names for values that are often used together or represent a finite set of related options or states. Enumerations make the code more readable and maintainable by replacing hard-coded integer values with meaningful identifiers. Here's an explanation of how the enum statement works in C:

1. Declaring an Enum: To declare an enum, you use the enum keyword followed by a user-defined type name and a pair of curly braces containing a list of identifiers, known as "enumerators." Each enumerator corresponds to an integer value, which is assigned sequentially starting from zero, unless explicitly specified.

Enumeration Data Types


```
enum Color {
    RED,   // 0
    GREEN, // 1
    BLUE   // 2
};
```

In this example, we've declared an enum called "Color" with three enumerators: RED, GREEN, and BLUE. By default, RED is assigned the value 0, GREEN is assigned 1, and BLUE is assigned 2.

2. Using Enums: Once you've defined an enum, you can use it to declare variables, function parameters, or return types, just like any other data type. Enum variables can only hold one of the enumerator values defined in the enum.

Enumeration Data Types

enum Color favoriteColor = GREEN;

Here, we've declared a variable favoriteColor of type enum Color and assigned it the value GREEN.

3. Custom Enumerator Values: You can explicitly assign values to enumerators if you want to control their integer representation. This can be useful when you need specific values for compatibility or other reasons.

Enumeration Data Types


```
enum Days {
    MONDAY = 1,
    TUESDAY = 2,
    WEDNESDAY = 3,
    THURSDAY = 4,
    FRIDAY = 5,
    SATURDAY = 6,
    SUNDAY = 7
};
```

In this example, we've assigned custom values to the enumerators in the enum Days.

Enumeration Data Types

4. Enum Constants: Enumerators are treated as constants in C, which means they cannot be modified once defined. They are effectively read-only variables.

5. Enum Size: The size of an enum variable in memory is typically the size of an int. However, this can vary depending on the compiler and platform. You can use the sizeof operator to determine the size of an enum.

```
printf("Size of enum Color: %lu bytes\n", sizeof(enum Color));
```

6. Comparing Enums: You can compare enum values using equality operators (==, !=) or use them in switch-case statements for conditional branching based on the enum's value.

Enumeration Data Types

Enums are particularly useful when you want to make your code more self-explanatory by replacing magic numbers with descriptive names, making it easier to understand and maintain. They're commonly used in situations where you have a fixed set of options or states, such as representing colors, days of the week, error codes, and more.

User-Defined Data Types: struct

- **struct**: In C programming, a struct (short for "structure") is a composite data type used to group together variables of different data types under a single name. It allows you to create custom data structures to represent complex, user-defined types. Structs are a fundamental part of C's ability to work with structured data.

1. Structure Declaration:

A structure is defined using the struct keyword, followed by a name that identifies the structure. Inside the curly braces, you specify the member variables (fields) that the structure will contain.

User-Defined Data Types: struct


```
struct Student {
    int studentID;
    char name[50];
    int age;
};
```

2. Creating Structure Variables:

Once a structure is defined, you can create variables of that structure type just like any other data type.

```
struct Student student1;  // Declaring a variable of type Student
```

You can also declare and initialize structure variables together:

```
struct Student student2 = {101, "John Doe", 20};
```

User-Defined Data Types: struct

3. Accessing Structure Members:

To access the members of a structure, you use the dot . operator.

```
student1.studentID = 102;
strcpy(student1.name, "Jane Smith");
student1.age = 21;
```

4. Nested Structures:

You can have structures within structures (nested structures). This allows you to create more complex data structures.

User-Defined Data Types: struct

UNIVERSITÀ DEGLI STUDI DI PADOVA

```
struct Date {
    int day;
    int month;
    int year;
};

struct Employee {
    int empID;
    char empName[50];
    struct Date hireDate;
};
```

User-Defined Data Types: struct

5. Passing Structures to Functions:

You can pass structures to functions by value or by reference (using pointers). Passing by reference is often used when you want to modify the structure within the function.

```
void displayStudent(struct Student s) {
    printf("ID: %d, Name: %s, Age: %d\n", s.studentID, s.name, s.age);
}
```

User-Defined Data Types: struct

6. Size of Structures:

The size of a structure in memory is determined by the sum of the sizes of its members. Padding may be added to align members in memory for efficiency. You can use the sizeof operator to find the size of a structure.

7. Practical Use Cases:

Structs are widely used for organizing and manipulating data in C. They are essential for tasks like representing complex records (e.g., students, employees, products), creating linked lists, and handling data serialization/deserialization.

User-Defined Data Types: struct

8. Typedef for Structures:

You can use typedef to create aliases for structure types, making the code more readable and concise.

```
typedef struct {
    int x;
    int y;
} Point;
```

Structures in C are crucial for representing more complex data structures and modeling real-world entities in your programs.

User-Defined Data Types: union

- **union**: In C programming, a union is a composite data type that is similar to a struct in that it allows you to group together variables of different data types under a single name. However, unlike a struct, a union can hold only one of its members at a time. Indeed, all members of a union share the same memory. Structures and unions are therefore very similar to each other, but the unions have the purpose of avoiding any memory waste caused by unused members.

1. Union Declaration:

A union is defined using the union keyword, followed by a name that identifies the union. Inside the curly braces, you specify the member variables (fields) that the union will contain.

User-Defined Data Types: union


```
union CarInfo {
    int carID;
    float price;
    char owner[50];
};
```

2. Creating Union Variables:

Once a union is defined, you can create variables of that union type, similar to any other data type.

```
union CarInfo car1;  // Declaring a variable of type CarInfo
```

You can also declare and initialize union variables together:

```
union CarInfo car2 = {101};  // Initializing with the carID
```

User-Defined Data Types: union

3. Accessing Union Members:

To access the members of a union, you use the dot . operator just like with structures. However, unlike structures, only one member can be active at a given time in a union.

```
car1.carID = 102; // carID is active
printf("Car ID: %d\n", car1.carID);

car1.price = 25000.50; // Now, price is active
printf("Car Price: %.2f\n", car1.price);
```

User-Defined Data Types: union

4. Practical Use Cases:
Unions are used when you want to efficiently use memory for a variable that can have one of several data types at different times. For example, in a database, you might want to store data for a single field (e.g., a car's ID, price, or owner), but you only need to access one field at a time.

5. Size of Unions:

The size of a union is determined by the size of its largest member. The union allocates enough memory to store the largest member. It does not store data for all members simultaneously.

User-Defined Data Types: union

6. Typedef for Unions:
As with structures, you can use typedef to create aliases for union types, making the code more readable and concise.

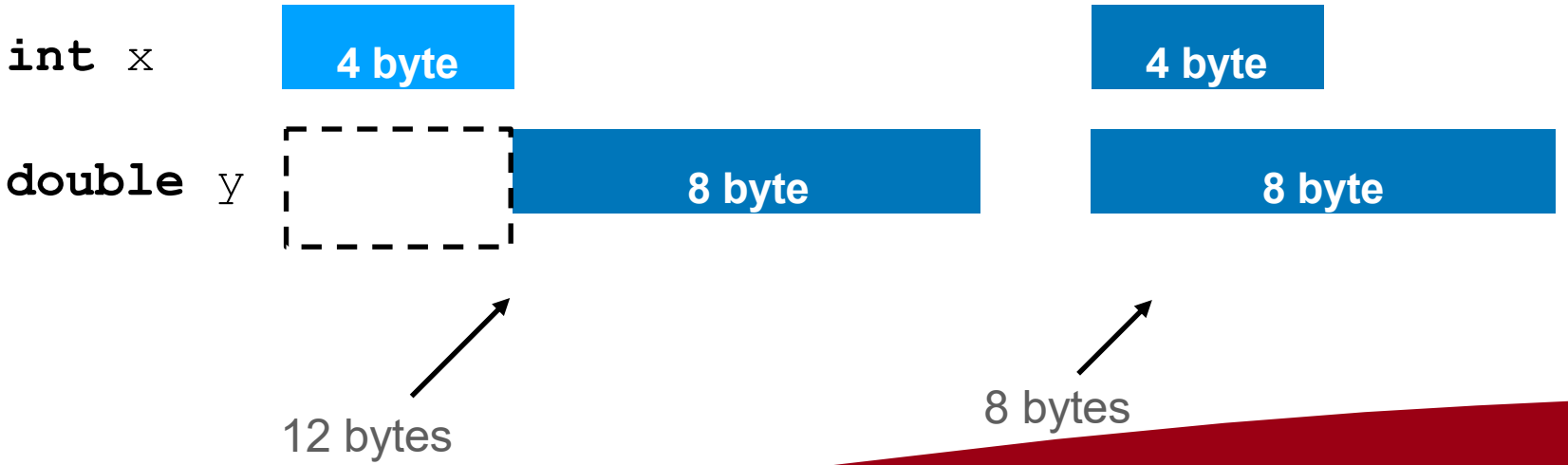
7. Limitations of Unions:
Due to their design, unions are not suitable for scenarios where you need to store multiple pieces of data simultaneously. If you need to access multiple members simultaneously, you should use a struct.

Use union data type when you need to use only one of several data types at a given time. They are particularly useful in cases where you want to save memory and are sure that only one member of the union will be active at any given time.

union vs struct


```
struct number {
   int x;
   double y;
};
```

```
union number {
   int x;
   double y;
};
```



Void Data Type

- void: Represents the absence of a data type. It is commonly used for functions that do not return a value (void functions) and for pointers to unspecified data types.

```
void functionName() {
    // No return value
}
```

bool Data Type

- The **bool** data type, introduced in C99 with the <stdbool.h> header, provides a convenient way to represent boolean values, which can be either true or false.
- To use the bool data type, include the <stdbool.h> header at the beginning of your C program.
  - #include <stdbool.h>
  - With the <stdbool.h> header included, you can declare boolean variables using the bool keyword:
    - bool isTrue = true;
    - bool isFalse = false;
  - In this example, isTrue is assigned the value true (1), and isFalse is assigned the value false (0).

bool Data Type

- In C, true is defined as 1, and false is defined as 0. These are the only valid values for boolean variables.
- You can use boolean operators such as && (logical AND), || (logical OR), and ! (logical NOT) with boolean variables to perform logical operations.
  - bool a = true;
  - bool b = false;
  - bool result = a && b; // result is false (0)

- The bool data type is commonly used for representing conditions and flags in C programs. It helps improve code readability and maintainability by making the intent of the variables more explicit.

bool Data Type

- Here's a practical example of using bool to check whether a number is even or odd:

```
#include <stdio.h>
#include <stdbool.h>
bool isEven(int number) {
    return(number % 2 == 0);
}
int main() {
    int num = 7;
    bool even = isEven(num);
    if (even) {
        printf("%d is even.\n", num);
    } else {
        printf("%d is odd.\n", num);
    }
    return 0;
}
```

bool Data Type

- While the bool data type is part of the C99 standard, some older C compilers and environments may not fully support it. It's essential to verify compiler compatibility and include the <stdbool.h> header when working with boolean variables.

In summary, the bool data type in C, introduced with the <stdbool.h> header, provides a convenient and standardized way to work with boolean values. It improves code readability and allows you to represent conditions and flags more explicitly in C programs.

Data Types

Choosing the appropriate data type is essential for efficient memory usage and program correctness. Using the wrong data type can lead to unexpected behavior and errors in your code. Understanding C's data types and their characteristics is crucial for writing robust and efficient programs.

In C, data types are a fundamental building block for defining variables, functions, and data structures, and they play a crucial role in how you manipulate and store data in your programs.