# Chapter 4. Representation of data types in a program

## 4.1 Data types

Actions in a program are performed by data, which in programming are called *operands*. Each operand can be either a variable or a constant value - a constant, as well as an expression or a function.

In programming it is common to say that each operand has its own *data type*. Data of different types are stored differently in computer memory, are processed differently, differ in a different set of operations and functions allowed for each type, and also differ in the set of values allowed for values of each type.

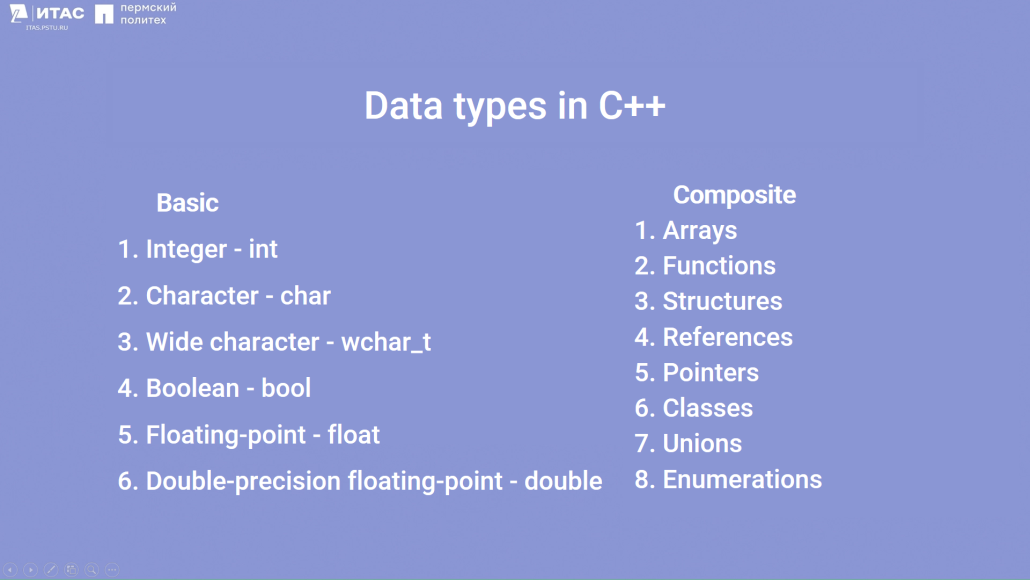
Data types can be categorized into *basic* and *composite* data types (figure 4.1). There are six basic data types from which composite types can be created. 

Figure 4.1 – Data types

Basic data types are the basic types provided by the programming language (figure 4.2). They include the following:

Integer data type: such as int. They allow you to store integers without the decimal part.

Character type: char. It is designed to store individual characters such as letters, numbers or special characters.

Logical type: bool. It can take two values: true or false. Logical data types are used to represent logical states and conditions.

Real data types: for example, float and double. They are used to store floating point numbers that can have a decimal part.

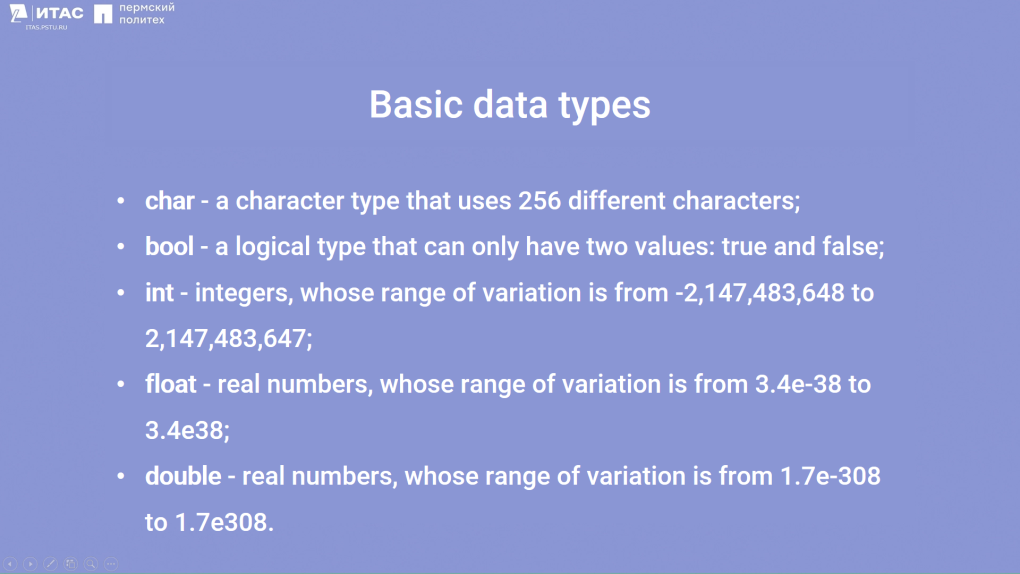


Figure 4.2 – Basic data types

In addition to the basic data types, there are also composite data types, including arrays, functions, structures, references, pointers, classes, joins, and enumerations. They allow you to combine variables and elements into more complex data structures [15].

## 4.2 Variable declaration

To use variables in programming, you must declare them. Variables are declared using a certain syntax (figure 4.3). First, the variable type is specified and then its name. In addition, when declaring a variable, you can immediately initialize it, i.e. assign an initial value to it.

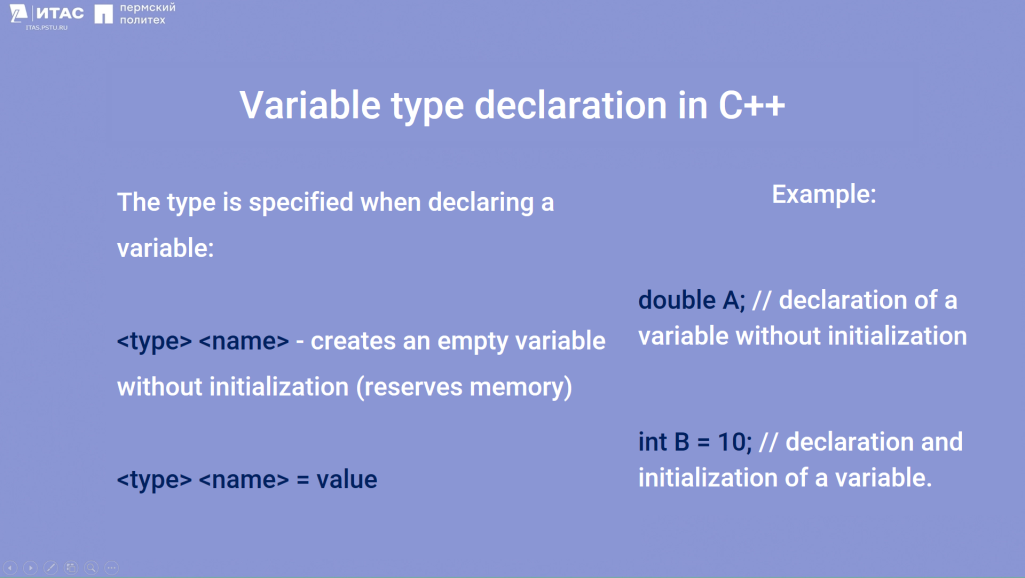


Figure 4.3 – Declaring the type of a variable

When using variables in programming, you must declare them before they are used. This is due to the fact that the language compiler processes the program code from top to bottom.

## 4.3 Type specifiers

The C++ programming language has *type specifiers* that allow you to change the range of values of standard numeric data types.

signed: specifier indicates that a variable can store negative values (with a - sign) and positive values (with a + sign). This specifier is specified by default, i.e. it is not necessary to specify it specially.

unsigned: specifier specifies that the variable can store only positive values or zero, i.e. it has no sign.

short: specifier reduces the range of values by a factor of two compared to a regular type.

long: specifier increases the range of values twice as compared to the usual type due to increased memory reservation.

These specifiers can be combined to obtain the necessary variables (figure 4.4).

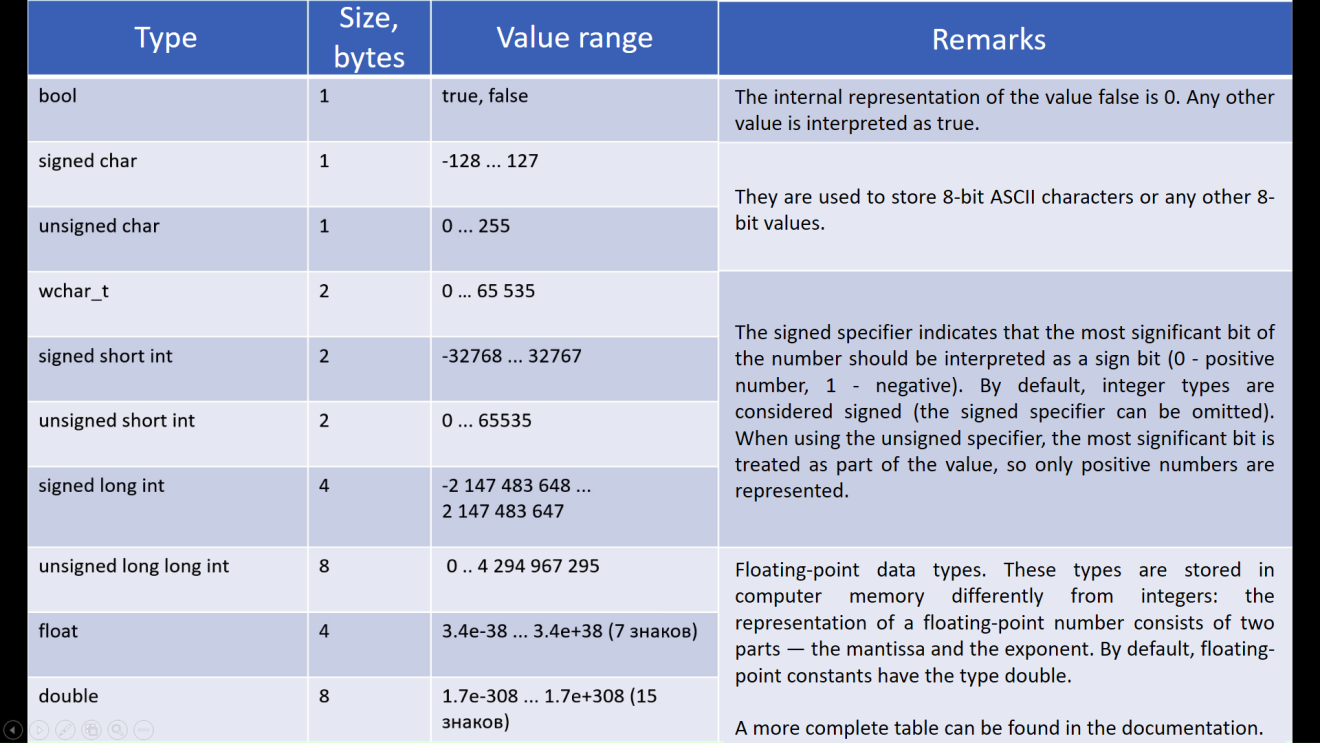


Figure 4.4 – Data type specifiers

## 4.4 Constants

When dealing with variables in programming, in addition to variable values, there are also immutable values called *constants*.

Constants have the following features:

1. They have a fixed value that cannot be changed during program execution. The value is initialized only once.

2. Constants can be used in operations and expressions, but the result of such operations cannot be stored in the memory area where the value of the constant is stored.

3. Constants are often used to specify fixed values such as numeric values, character values, logical values, and other immutable data.

When declaring constants, the const keyword is used along with syntax similar to declaring ordinary variables. An important feature of constants is mandatory initialization when declaring them. An example of a constant declaration is shown in Figure 4.5.

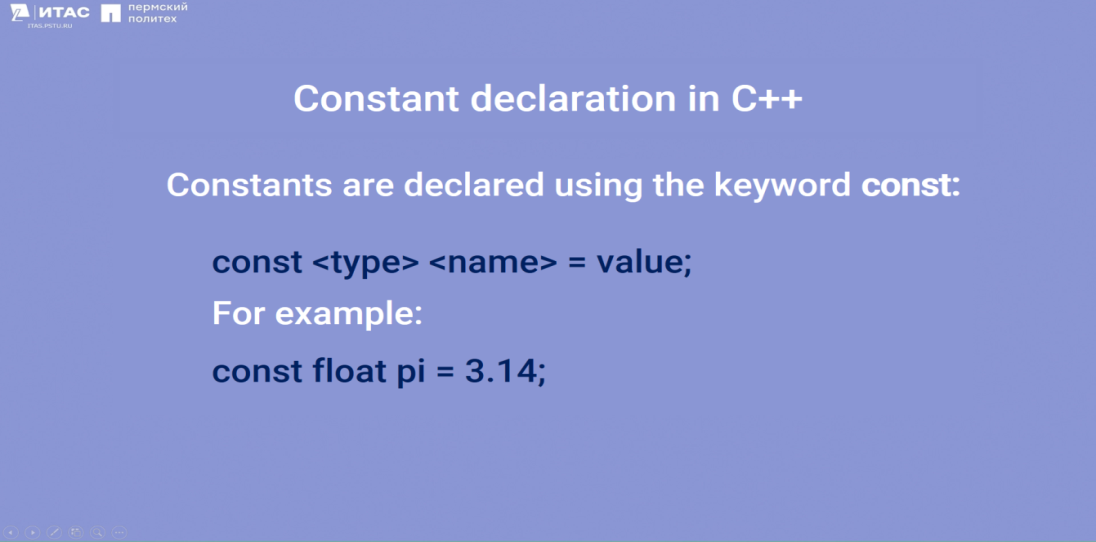


Figure 4.5 - Declaring a constant

## 

## 4.5 Pointers and References

A pointer is a variable that holds the memory address of another variable. It allows you to access the value of that variable through its address.

A reference is an alias for a variable. It provides an alternative name to access the same value. A reference must always be initialized when declared and cannot be reassigned.

To declare a pointer, use "" followed by the variable syntax. To declare a reference, use "&" followed by the variable syntax. These symbols are not only used for declaration but also for direct work with RAM. "&" is needed to get the address of a variable, and "" is used for dereferencing and indirection of pointers (figure 4.6).

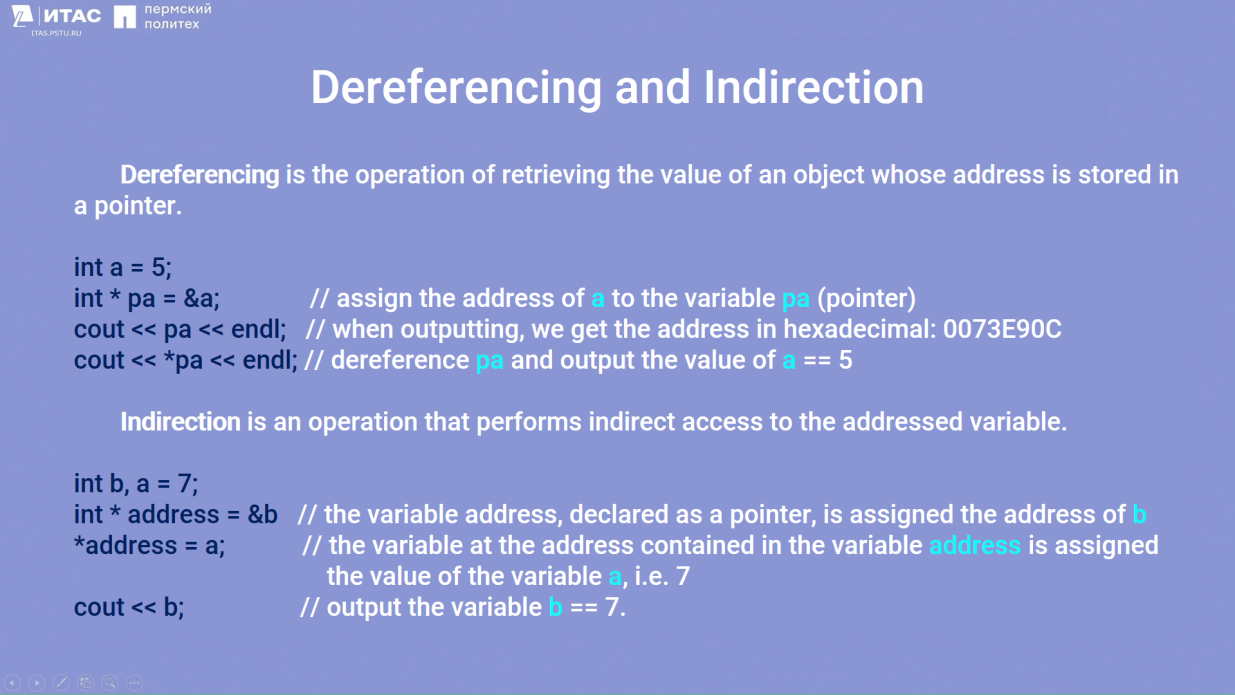


Figure 4.6 – The notions of dereferencing and readdressing

Additional examples demonstrating the principles of pointers and references are presented in figures 4.7, 4.8, and 4.9.

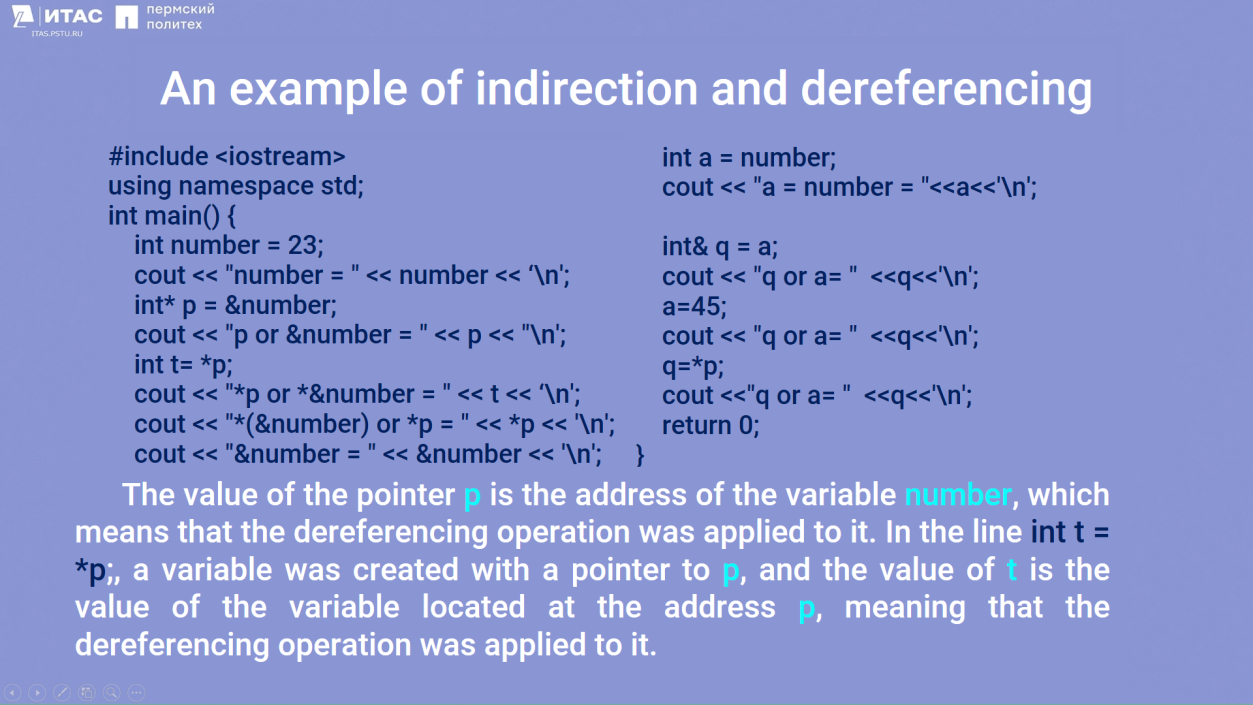


Figure 4.7 – Example of using dereferencing and dereferencing

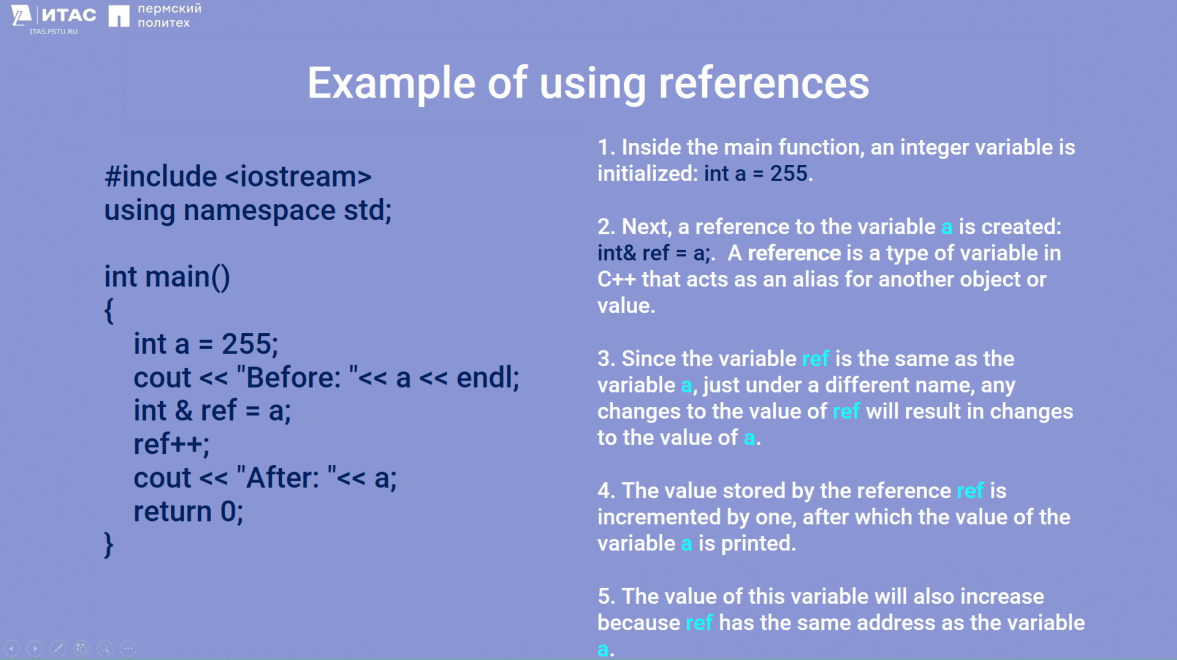


Figure 4.8 – Example of using dereferencing and dereferencing

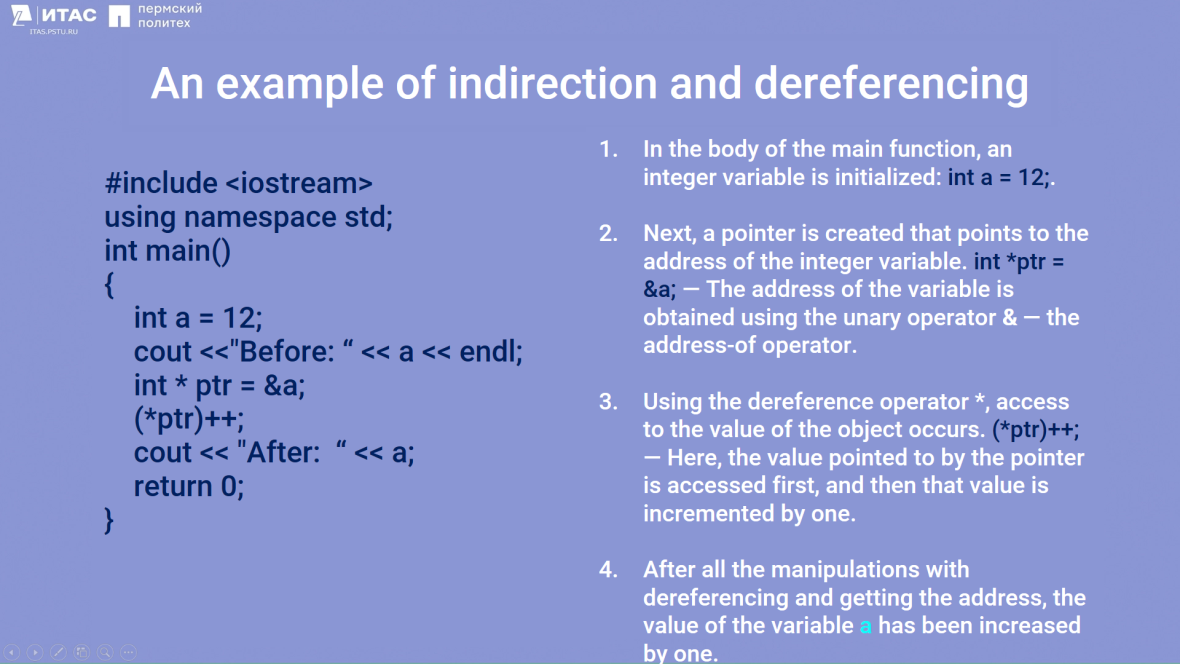


Figure 4.9 – Example of using references

## 4.6 Memory Classes

In C++, there exists the concept of memory class. To understand this information, it's necessary to introduce certain concepts:

*Lifetime* – the period during program execution in which a variable or function exists. Lifetime can be either *permanent* (throughout the entire program) or *temporary* (during the execution of a block enclosed in curly braces).

*Scope* – the part of the program text where the memory associated with a given identifier is accessible. Scope is divided into local and global scopes.

*Local Scope* – enclosed within curly braces. A variable functions from its declaration point until the closing of the curly braces.

*Global Scope* – outside any curly braces. Defined outside the program block. A variable functions from its declaration point until the end of the program.

An example demonstrating the principles of variable lifetime and its scope is presented in figure 4.10.

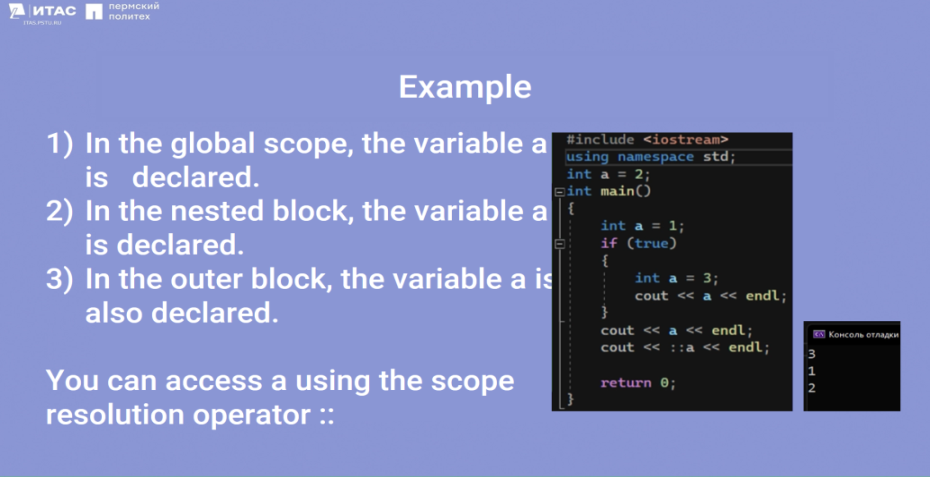


Figure 4.10 – Variable Lifetime and Scope

The example shows how scope works. In the first case, 3 is printed because this number is within the scope. Next, 1 is printed because 3 has already gone out of scope, and its lifetime has ended. Using double colons (::) refers to the global scope, so 2 is printed to the console.

Memory Class – defines the lifetime, scope, and location in RAM. If the memory class is not explicitly specified, the compiler will assign it automatically. There are four types of memory classes: auto, extern, static, and register.

*Auto:*

1. The lifetime of this variable is from its description to the end of the block.
2. Not used for global variables.
3. When exiting the block, the memory where the variable's description is located is freed.
4. For local variables, auto is applied by default, so there is no need to explicitly assign it.

*Register:*

This specifier replicates auto in everything, but variables with this memory class are placed directly into the processor registers. This can significantly speed up program execution if the variable is used frequently in calculations.

*Static* (data segment):

1. Initialization – once when the statement containing the definition of these variables is executed for the first time.

2. They can be global - visible in all program files and local - visible only in the file' in which they are described. If you set the variable static:

1. Outside the block: it will be global (lifetime – permanent (the entire program)).

2. Inside the block: it will be local (lifetime – until the end of the program).

Using the static modifier, you can describe static fields and methods of a class. They can be considered global variables or functions, accessible only within the scope of the class. An example demonstrating the difference between memory classes like auto and static is presented in figure 4.11.

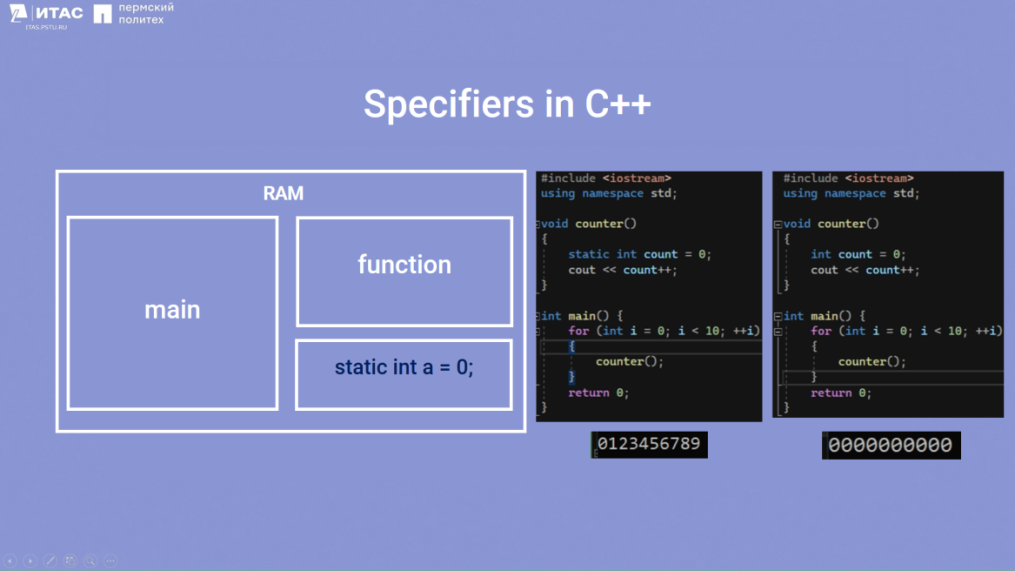


Figure 4.11 - Difference between memory classes auto and static

*Extern:* declares the most global variable (associates a variable with a specific memory area and describes the variable's properties).

1. The variable is defined anywhere in the program, including in another file.

2. Used to create variables accessible across all files of the program.

3. If the variable is initialized (given an initial value) during declaration, extern is ignored.

An example using extern is presented in figure 4.12.

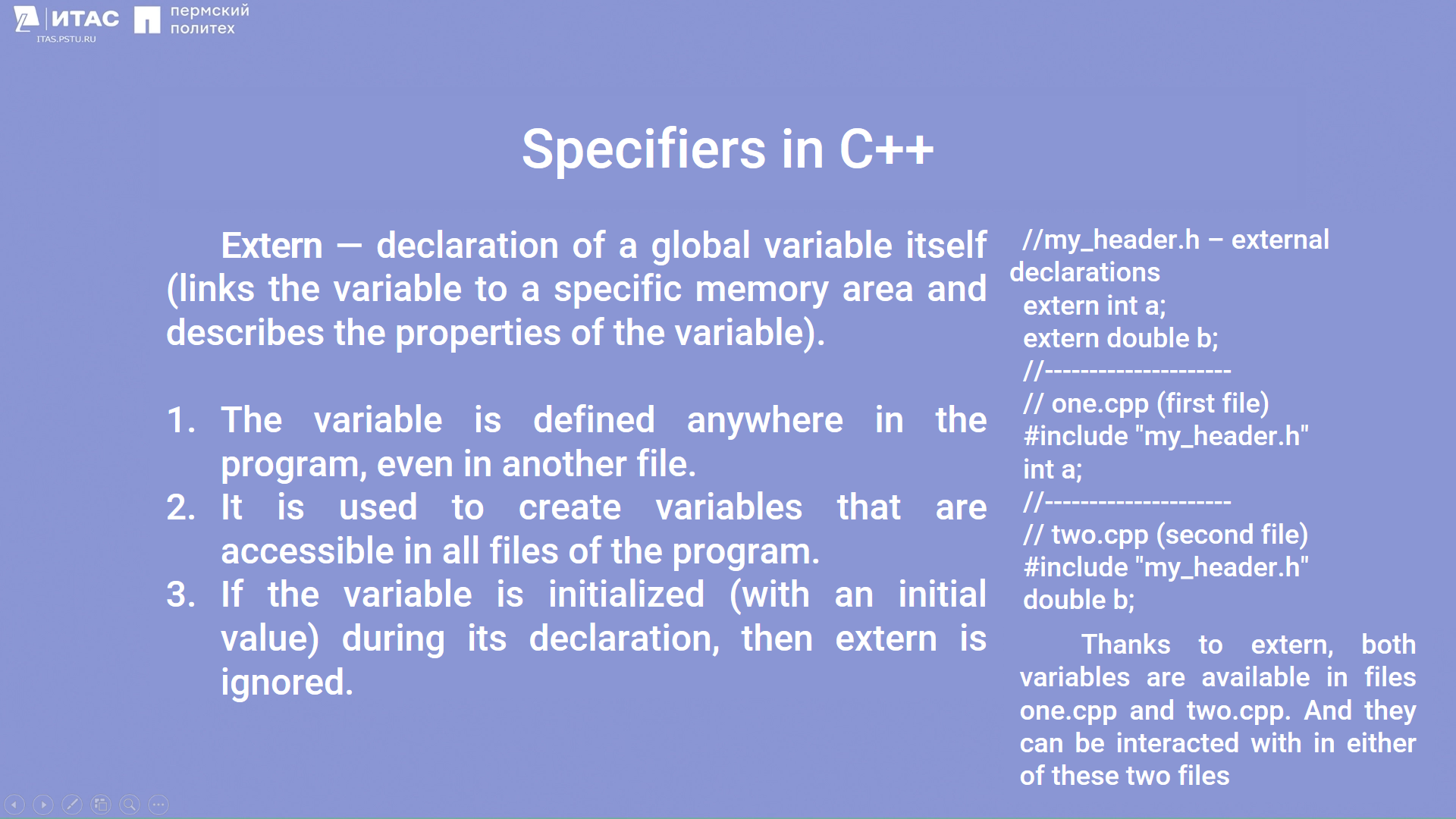


Figure 4.12 – Example of working with extern memory class

Thanks to extern, both variables are accessible in files one.cpp and two.cpp, allowing interaction with them in either of these two files.