COMP 322 Lecture 3 - C++ Basics

Junji Duan

2024/1/19

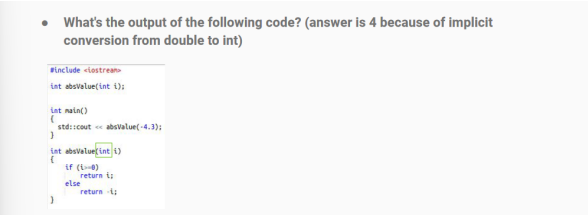
Today's Outline

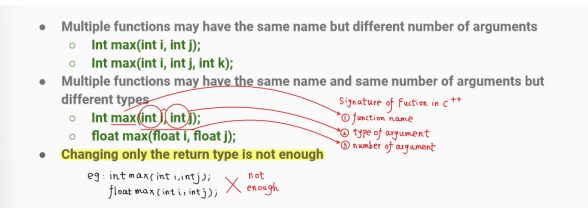
- Functions

Functions

- Same as in C and java
- Should be declared before being used
- Declaration should include the name, return type and arguments type
  - Also called prototype or signature of a function
- If the function doesn't return a value, its return type should be declared void
- Functions can be recursive

Functions Overloading





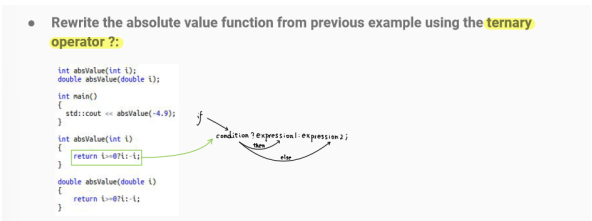
Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameter lists. When you call an overloaded function, the compiler determines which function to call based on the provided parameter types or number.

```cpp
#include <iostream>

// Function overloading example
int add(int a, int b) {
    return a + b;
}

// Overloaded function
double add(double a, double b) {
    return a + b;
}

int main() {
    int sum_int = add(3, 4); // Calls the first add function
    double sum_double = add(3.5, 4.2); // Calls the second add function

    std::cout << "Sum of integers: " << sum_int << std::endl;
    std::cout << "Sum of doubles: " << sum_double << std::endl;

    return 0;
}
```

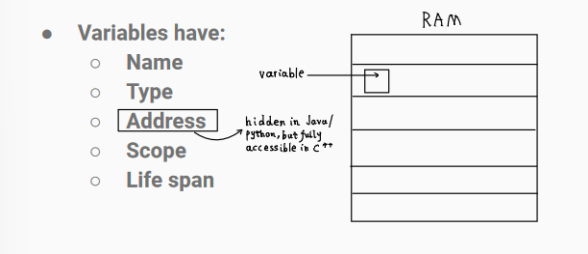
In this example, we define two functions named add, each taking different parameter types (one accepts integers, the other accepts double-precision floating-point numbers). When we call the add function in the main function, the compiler determines which version of the add function to call based on the provided parameter types.

Thus, even though the function names are the same, the compiler can distinguish between them due to different parameter lists and selects the appropriate function to call. This is the essence of function overloading in C++.

Small Quiz

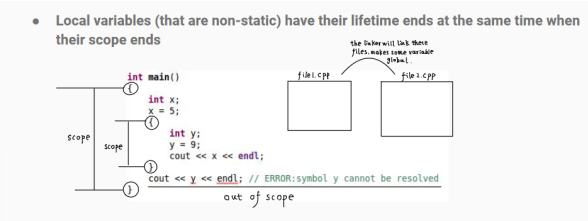


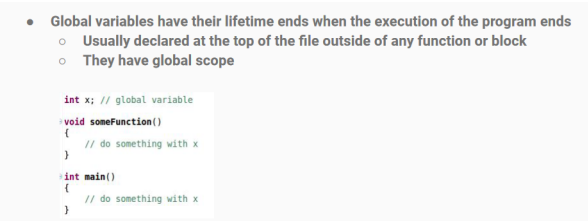
More about variables ...

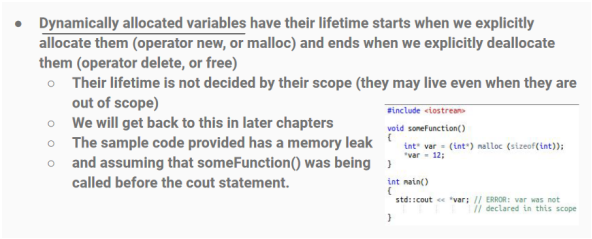


Scope and lifetime of a variable

- When declaring variables we specify the name and type, but we should also keep in mind their scope and lifetime
- Scope of a variable
  - A section of the program where the variable is visible (accessible)
- Lifetime of a variable
  - The time span where the state of a variable is valid (meaning that the variable has a valid memory)
- Local variables (that are non-static) have their lifetime ends at the same time when their scope ends
  - Local variables may also be called automatic variables because they are automatically destroyed at the end of their scope
  - Scope of local variables is comprised from the moment they are declared until the end of the block or function where they reside (in other terms, until the execution hits a closing bracket })







Scope and lifetime of a variable (static)