

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

- What's the output of the following code? (answer is 4 because of implicit conversion from double to int)

```
#include <iostream>
int absolute(int i);

int main()
{
    std::cout << absolute(4.3);
}

int absolute(int i)
{
    if (i < 0)
        return i;
    else
        return -i;
}
```

- Multiple functions may have the same name but different number of arguments
 - `int max(int i, int j);`
 - `int max(int i, int j, int k);`
- Multiple functions may have the same name and same number of arguments but different types
 - `int max(int i, int j);`
 - `float max(float i, float j);`
- Changing only the return type is not enough
 - eg: `int max(int i, int j);`
 - `float max(int i, int j);` ✗ not enough

Signature of function in C++

- ① function name
- ② type of argument
- ③ number of argument

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.

```
#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

- Rewrite the absolute value function from previous example using the **ternary operator** `?:`.

```
int absolute(int i);
double absolute(double i);

int main()
{
    std::cout << absolute(-4.9);
}

int absolute(int i)
{
    return i < 0 ? i : -i;
}

double absolute(double i)
{
    return i < 0 ? i : -i;
}
```

condition ? expression1 : expression2;

if condition is true, return expression1; else, return expression2;

More about variables ...

- Variables have:
 - Name
 - Type
 - Address
 - Scope
 - Life span

hidden in Java/Python, but fully accessible in C++

RAM

- Dynamically allocated variables have their lifetime starts when we explicitly allocate them (operator new, or malloc) and ends when we explicitly deallocate them (operator delete, or free)
 - Their lifetime is not decided by their scope (they may live even when they are out of scope)
 - We will get back to this in later chapters
 - The sample code provided has a memory leak and assuming that someFunction() was being called before the cout statement.

```
#include <iostream>
void someFunction()
{
    int* var = (int*) malloc (sizeof(int));
    *var = 12;
}

int main()
{
    std::cout << *var; // ERROR: var was not
                      // declared in this scope
}
```

Scope and lifetime of a variable (static)

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 }

- Global static variables have their lifetime ends when the execution of the program ends but their scope is limited to the file in which they are declared (file scope)
 - Scope is affected (reduced) but not the lifetime

however, if the variable is static and variable is variable, can't be used by the other file

only accessible by same file

file1.cpp

file2.cpp

- Local static variables have their lifetime ends when the execution of the program ends but their scope is limited to the function in which they are declared (function scope)
 - Lifetime is affected (extended) but not the scope

created by RAM, by not really created, but normally

if x is not static, it will "die" after the function; it will print 1, 1, 1.

if x is static, it will be remembered inside the RAM, each time calling the function the compiler will read from the RAM, it will print 1, 1, 5.

```
#include <iostream>
int someFunction()
{
    static int x = 0;
    return x;
}

int main()
{
    std::cout << someFunction() << std::endl;
    std::cout << someFunction() << std::endl;
    std::cout << someFunction() << std::endl;
}
```

- Local variables (that are non-static) have their lifetime ends at the same time when their scope ends

scope

scope

scope

out of scope

```
int main()
{
    int x;
    x = 5;
    int y;
    y = 9;
    cout << x << endl;
    cout << y << endl; // ERROR: symbol y cannot be resolved
}
```

the linker will link these files, makes some variable global

file1.cpp

file2.cpp

- Global variables have their lifetime ends when the execution of the program ends
 - Usually declared at the top of the file outside of any function or block
 - They have global scope

```
int x; // global variable

void someFunction()
{
    // do something with x
}

int main()
{
    // do something with x
}
```