ENPM702

Introductory Robot Programming

L5: Functions

v1.0

Lecturer: Z. Kootbally
Semester/Year: Summer/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

≡ Changelog ────────────────────────────────

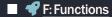- **v1.0**: Original version.

🖋 **Learning Objectives** _____

By the end of this session, you will be able to:

■ Declare, initialize, and manipulate `std :: string` objects using common methods for access, modification, concatenation, and insertion operations.

■ Describe the internal memory model of `std :: string`, including Small String Optimization (SSO), the relationship between size and capacity, and dynamic memory allocation strategies.

■ Manage `std :: string` memory usage efficiently using `reserve()`, `resize()`, and `shrink_to_fit()` methods.

■ Compare and contrast C-style arrays with `std :: array`, understanding compilation requirements and safety considerations.

■ Declare, initialize, and manipulate both single and multidimensional arrays using proper syntax and access patterns.

■ Understand the dynamic nature of `std :: vector`, including its memory management, capacity growth strategies, and performance characteristics.

■ Perform efficient insertion and deletion operations on vectors using `push_back()`, `emplace_back()`, `insert()`, `erase()`, and related methods.

■ Apply best practices for memory management across all STL container types to write efficient and maintainable code.

Introduction to Functions

A function is a named group of statements that can be executed as a unit.

- [ ] 🚀 <u>F: Functions</u>

Code Reusability

Write once, use everywhere! Functions eliminate the need to copy and paste code blocks (DRY = Don't Repeat Yourself). Create a function once and call it multiple times throughout your program.

ⓘ

```cpp
int calculate_area(int length, int width) {
    return length * width;
}
// Use it anywhere:
int room1 = calculate_area(10, 12);
int room2 = calculate_area(8, 15);
```

Modularity & Organization

Break complex programs into smaller, manageable pieces. Functions help organize code logically, making it easier to understand and maintain large projects.

```cpp
// Each function has a specific purpose
void get_user_input();
void process_data();
void display_results();

int main() {
    get_user_input();
    process_data();
    display_results();
}
```

Easier Debugging & Testing

Isolate problems quickly! When code is organized in functions, you can test and debug individual components separately, making bug fixing much more efficient.

```cpp
// Test individual functions
bool is_valid_email(string email) {
    // Validation logic here
    return email.find("@") ≠ string::npos;
}

// Easy to test:
is_valid_email("test@email.com")
```

Abstraction of Complex Operations

Hide complexity behind simple interfaces. Functions allow you to use complex operations without worrying about implementation details.

```cpp
// Complex math hidden behind simple function
double calculate_compound_interest(double principal, double rate,
↳ int years) {
    // Complex formula abstracted away
    return principal * pow(1 + rate, years);
}
```

Function Declarations

A function declaration (or prototype) consists of the return type, the function identifier, and optional parameters (types and names). The body of the function is not part of the header.

```
type identifier(<parameters>); // e.g., int add_numbers(int a, int b);
```

- ☐ `type` – What kind of value the function is expected to return to the calling function (the caller) or main program. When a function does not return anything, its `type` should be `void`.
- ☐ `identifier` – The name given to a function.

> ✏️
> - ☐ We will use the same naming convention as for variables (snake_case).
> - ☐ A function does an action, therefore, the identifier should include a verb.

- ☐ `<parameters>` – If the function takes parameters, the parameters are listed in the parentheses. Each parameter has the form `type identifier`. If the function does not use any parameter, the parentheses are left empty.

🚀 NL.25: Don't use void as an argument type

How are Function Declarations Used?

■ The function declaration provides a promise that the function will be implemented (defined) elsewhere in the code, either later in the same source file or in a different source file.

■ A function declaration allows you to use (call) that function before its actual definition in the code. This is particularly useful in scenarios where multiple functions call each other.

■ During compilation, the linker will look for the actual definition of the function.

✏️ **Exercise** #1 ─────────────────────────────

1. <u>Declare</u> a function which sums two integers and returns the result.
2. Call this function in the `main()` function.
3. Generate the 📄 **.o** file: 🖥️ `g++ -std=c++17 -c lecture5.cpp -o lecture5.o`
   - `g++` : The GNU **C⁺** compiler.
   - `-c` : Tells the compiler to generate the object file **without linking**.
   - `lecture5.cpp` : The source file you want to compile.
   - `-o lecture5.o` : Specifies the output object file name. If you omit the `-o lecture5.o`, it will default to `lecture5.o`.
4. Generate the executable: 🖥️ `g++ lecture5.o -o lecture5`
   - This command will link the object file `lecture5.o` and create an executable named `lecture5`. If you do not specify the `-o` option, the output will default to an executable named `a.out`.

Function Definitions

A function definition (often referred to as a function implementation) consists of the
`return_type`, the `identifier`, the `parameters`, and the actual body of the function.

```
type identifier(<parameters>) {
    // body of the function
}
```

```cpp
// function declaration/prototype
int add_numbers(int a, int b);

// function definition/implementation
int add_numbers(int a, int b) {
    return a + b;
}

int main(){
    std::cout << add_numbers(3, 5) << '\n';
}
```

Function Calls

When a function is called, the execution of the program jumps to the function definition, runs the code inside the function, and then returns back to the point from where the function was called, continuing from the next statement.

⚙ **Demonstration** ━━

- Line **15** – `main()` is called automatically.
- Line **16** – Control is passed to `print_hello()`.
- Lines **11**-**12** – The body of `print_hello()` is executed.
- Line **12** – Control is passed to `print_world()`.
- Line **7** – The body of `print_world()` is executed.
- Line **8** – Control is returned to `print_hello()`, which executes the remaining code.
- Line **13** – Control is returned to `main()`, which executes the remaining code.
- Line **18** – The program exits.

```cpp
1   // function declarations
2   void print_hello();
3   void print_world();
4
5   // function definitions
6   void print_world() {
7       std::cout << "world\n";
8   }
9
10  void print_hello() {
11      std::cout << "hello, ";
12      print_world();
13  }
14
15  int main() {
16      print_hello();
17      std::cout << "exit main\n";
18  }
```

✏️ **Exercise** #2 ═══

Inspect the code.

```cpp
void print_hello() {
    std::cout << "hello, ";
    print_world();
}

void print_world() {
    std::cout << "world\n";
}

int main() {
    print_hello();
    std::cout << "exit main\n";
}
```

🖊 **Exercise** #3 ────────────────────────────────

Inspect the code.

```cpp
void prompt_user() {
    std::cout << "Enter a number: ";
    int num{};
    std::cin >> num;
    print_number(num);
}

void print_number(int number) {
    if (number == -1)
        return;

    if (number > 0)
        std::cout << "The number is: " << number << "\n";
    else
        prompt_user();
}

int main(){
  prompt_user();
}
```

👍

Always provide function declarations.

■ Projects can be very large and have many functions. You cannot afford spending time defining functions before other functions (see slide 15).

■ Programs can contain cyclic references and there is no other way to solve them besides declaring functions (see slide 16).

≡ Header Files

Function declarations and implementations are usually split in different files. This separation improves modularity, maintainability, and code organization.

- Function declarations are located in header files (📄 ***.hpp**)
  - Header files are never compiled and always included.
- Function definitions are located in source files (📄 ***.cpp**)
  - Source files are always compiled and never included.

⋮≡ ToDo

- Write function declarations for slide 16 in 📄 **include/lecture5.hpp**
- Include 📄 **lecture5.hpp** in 📄 **lecture5.cpp** with `#include "lecture5.hpp"` to include the declarations.
- Edit 📄 **CMakeLists.txt** to tell CMake where to find 📄 **lecture5.hpp**

```
include_directories(lecture5/include)
```

Documentation

All functions must be documented (with Doxygen in this course). It is generally recommended to document function declarations rather than function definitions for the following reasons:

- ■ The declaration of a function represents its interface (what the function does, its purpose, and how to use it). It is the part of the code that users of the function need to know about.
- ■ Documenting declarations allows developers to understand how to use the function without having to navigate to its implementation (which may be in a separate source file).
- ■ If you document both the declaration and the definition, you risk duplicating comments, leading to potential inconsistency if the comments in one place are updated but not the other.

👍 Document function declarations (located in 📄 *.**hpp** files)

👎 Do not document both function declarations and function definitions.

🧪 **Example**

```cpp
/**
 * @brief Print the number @p number to the console
 *
 * @param number The number to be printed
 */
void print_number(int number);
```

Parameters vs. Arguments

The terms Parameters and Arguments are often used interchangeably. However, they do have distinct meanings.

- ◼ Parameters (or formal parameters) are the variables listed in the function's declaration (or definition). They act as placeholders for the values that the function operates on.
- ◼ Arguments (or actual parameters) are the actual values that are passed to a function when it is called. They are substituted for the function's parameters when the function executes.

```cpp
int add_numbers(int x, int y) {  // x and y are parameters
    return x + y;
}

int main() {
    std::cout << add_numbers(3,4) << '\n'; // 3 and 4 are arguments
}
```
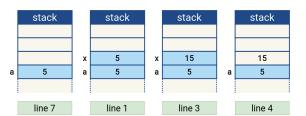
☰ Passing Arguments: Pass-by-Value

When an argument is passed by value, a copy of that argument's value is made, and the function receives that copy. <u>Changes made to the parameter inside the function do not affect the original value outside of the function.</u>

```cpp
void add_ten(int x) {
    // Implicit int x{a};
    x += 10;  // 15
}

int main() {
    int a{5};
    add_ten(a);
    std::cout << a << '\n'; // 5
}
```

	stack		stack		stack		stack
				x	5	x	15		15
a	5	a	5	a	5	a	5

line 7	line 1	line 3	line 4
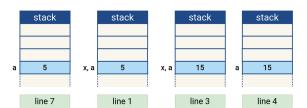
Pass-by-Value

✎
- ■ Memory – Since pass-by-value involves making a copy of the argument, it can be inefficient for large data structures as it consumes additional memory.
- ■ Safety – Functions cannot modify the original data when using pass-by-value, making it a safer choice in scenarios where you want to ensure that the original data remains unchanged.
- ■ Performance – For basic data types (`int`, `char`, etc), the overhead is minimal, and pass-by-value is common. However, for large objects, pass-by-reference or pass-by-const-reference is often preferred to avoid unnecessary copying and improve performance.

☰ Passing Arguments: Pass-by-Reference

Unlike pass-by-value, where a copy of the argument's value is passed to the function, with **pass-by-reference**, the function receives a reference to the original data. This means any modification made to the parameter inside the function will reflect on the original value outside of the function.

```cpp
1  void add_ten(int &x) {
2      // Implicit: int &x{a};
3      x = x + 10;  // 15
4  }
5
6  int main() {
7      int a{5};
8      add_ten(a);
9      std::cout << a << '\n'; //
           ↪  15
10 }
```

✎ **Exercise** #4

Write the function `swap_values` which exchanges the values of two variables.

```cpp
int main(){
    int x{5};
    int y{10};

    std::cout << x << ", " << y << '\n'; // 5, 10
    swap_values(x, y);
    std::cout << x << ", " << y << '\n'; // 10, 5
}
```

📝 For future reference, the standard library provides `std :: swap`

🧪 Example

```cpp
void push_ten(std::vector<int> &v) {  // Passed by reference
    // Implicit: std::vector<int> &v{num_vect};
    v.push_back(6);
}

int main() {
    std::vector<int> num_vect{1, 2, 3, 4, 5};
    push_ten(num_vect);
    for (const auto &item : num_vect) {
        std::cout << item << " ";  // 1 2 3 4 5 6
    }
    std::cout << '\n';
}
```

Pass-by-Reference

■ Memory – Pass-by-reference is memory-efficient, especially for large data structures, as it avoids copying. Only the memory address (reference) is passed.

■ Safety – Since functions can modify the original data when using pass-by-reference, you need to be cautious. This can lead to unintended side effects if not handled properly.

■ Usage with `const` (see next slide) – If you want to pass an argument by reference but don't want to allow the function to modify it, you can use `const` before the reference type. This is called pass-by-const-reference and is common for efficient read-only operations on large data structures.

■ Performance – Passing by reference is often faster for large data structures because it avoids the cost of copying. For basic types, though, the performance difference might be negligible.

☰ Passing Arguments: Pass-by-const-Reference

Pass-by-const-reference combines two concepts: passing by reference and the use of the `const` keyword. When you pass an argument by `const` reference, you pass a reference (i.e., the memory address) of the argument to the function, but the function is prohibited from modifying the data being referred to.
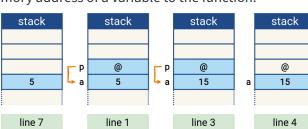
```cpp
void print_vector(const std::vector<int> &v) {  // Passed by const reference
    // Implicit: const std::vector<int> &v{num_vect};
    for (const int &item : v) {
        std::cout << item << " ";  // 1 2 3 4 5
    }
    // v.push_back(100);  // This would be an error since v is const
}

int main() {
    std::vector<int> num_vect{1, 2, 3, 4, 5};
    print_vector(num_vect);
}
```

☰ Passing Arguments: Pass-by-Pointer

Pass by pointer involves passing the memory address of a variable to the function.

```cpp
void add_ten(int* p) {
    // Implicit: int* p{&a};
    *p += 10;
}

int main() {
    int a{5};
    add_ten(&a);
    std::cout << a << '\n';  //
    ↪   15
}
```

Pass-by-Pointer

■ Null Pointers – One notable difference between references and pointers is that pointers can be null (pointing to no valid memory location). Before dereferencing a pointer inside a function, you should ensure it's not null.

■ Safety – Because pointers can be null or can be inadvertently modified to point to invalid memory locations, there's an inherent risk when working with them.

■ Usage – Pass-by-pointer is especially common in scenarios involving dynamic memory allocation with **new** (though modern C++ encourages the use of smart pointers).

≡ Returning Values ─────────────────

Functions in C++ must handle return values according to their declared return type:

- **void functions** do not return any value. The function ends when it reaches a `return;` statement or the closing brace.

```cpp
void print_message() {
    std::cout << "Hello World\n";
    // No return statement needed
}
```

- **Non-void functions** must explicitly return a value of the specified type. Failing to return a value results in **UB**.
    - Incorrect - causes **UB**:

    ```cpp
    int calculate_sum() {
        auto result{5 + 3};
        // UB - missing return statement
    }
    ```

    - Correct - explicit return:

    ```cpp
    int calculate_sum() {
        return 5 + 3;
    }
    ```

> **Key Rule**
>
> ⊙ Every execution path in a non-**void** function must reach a **return** statement.

📕 Return-by-Value

Return-by-value means the function returns a copy of the value, not a reference or pointer to the original data. This is the default and most common way to return values from functions, particularly for fundamental data types.

```cpp
int add(int a, int b) {
    int sum{a + b};
    return sum;  // Returns copy of sum
}

int main() {
    int result{add(1, 2)};
    std::cout << result << '\n'; // 3
}
```

- The function `add` creates a copy of `sum`'s value and returns it to the caller.
- The local variable `sum` is destroyed when the function ends, but the returned copy persists.
- This returned value is used to initialize `result` in `main()`.
- **Key point:** Changes to `result` don't affect the original `sum` variable (which no longer exists anyway).

📜 **Copy Elision** ————————————————————————————————

Copy elision is a compiler optimization that eliminates unnecessary copying or moving of objects during function returns, parameter passing, and object construction.

≡ Compiler Strategy for Return Values ──────────────

1. **Copy Elision (RVO/NRVO):**
   - ■ Constructs the object directly at the destination
   - ■ **Return Value Optimization (RVO):** for temporary objects.
   - ■ **Named Return Value Optimization (NRVO):** for named local objects.
   - ■ Most efficient – no copying or moving required.
2. **Implicit Move:**
   - ■ Moves the object instead of copying (from C++11).
   - ■ Applies to local objects and rvalue expressions.
   - ■ Much more efficient than copying for expensive objects.
3. **Copy Construction:**
   - ■ Creates a copy of the object.
   - ■ Fallback when elision and move are not possible.
   - ■ Most expensive option for complex objects.
4. **Compilation Error:** If the object is neither copyable nor movable.

📕 Return Value Optimization (RVO)

Return Value Optimization (RVO) is a compiler optimization that eliminates temporary objects created during function returns by constructing the return value directly at the caller's destination.

> 📝 Since **C**[+ 17], RVO is guaranteed for prvalue expressions (pure rvalues) - temporary objects created in **return** statements.

```cpp
std::vector<int> create_vector() {
    return std::vector<int>{1, 2, 3};
    //        ^^^^^^^^^^^^^^^^^^^^^^^^
    //        Prvalue - guaranteed RVO
}

int main() {
    std::vector<int> vect{create_vector()};
    // Construction directly in vect's location
}
```

- ■ **Without RVO:** Function would create temporary vector, then copy/move it to **vect**
- ■ **With RVO:** The vector {**1, 2, 3**} is constructed directly in **vect**'s memory location
- ■ **Result:** No temporary object creation, no copy/move operations
- ■ **Performance:** Eliminates potentially expensive copy/move operations for large objects

📜 Named Return Value Optimization (NRVO)

Named RVO (NRVO) is a compiler optimization that eliminates copying/moving when returning a named local object by constructing it directly at the caller's destination.

> 📝 Unlike RVO, NRVO is not guaranteed by the C++ standard - it's an optional optimization that compilers may or may not perform.

```cpp
std::vector<int> create_vector() {
    std::vector<int> local_vec{1, 2, 3};
    // ... potential operations on local_vec ...
    return local_vec;  // Named object return
}

int main() {
    std::vector<int> result{create_vector()};
}
```

- **With NRVO:** `local_vec` is constructed directly in `result`'s memory location.
- **Without NRVO:** `local_vec` is constructed locally, then moved/copied to `result`.
- **Detection:** Print the memory address for `local_vec` and `result`.
- **Limitations:** NRVO may fail in some situations.

🧪 **Example: When Copy Elision Fails – Control Flow** ───────────

```cpp
std::vector<int> conditional_return(bool flag) {
    std::vector<int> vec1{1, 2, 3};
    std::vector<int> vec2{4, 5, 6};
    return flag ? vec1 : vec2;  // NRVO fails!
}

std::vector<int> multiple_paths(int choice) {
    std::vector<int> result{};
    if (choice == 1) {
        result = {1, 2, 3};
        return result;          // NRVO fails!
    } else if (choice == 2) {
        result = {4, 5, 6};
        return result;          // NRVO fails!
    }
    return result;              // NRVO fails!
}
```

Why NRVO fails

❓
- ■ Compiler cannot determine at compile-time which object will be returned.
- ■ Multiple named objects are potential return candidates.
- ■ NRVO requires a single, unambiguous return object.

🧪 **Example: When Copy Elision Fails – Assignment** ────────────

```cpp
std::vector<int> create_vector() {
  std::vector<int> local_vect{1, 2, 3};
  std::cout << "&local_vect: " << &local_vect << '\n';  //@1
  return local_vect;
}

int main() {
  std::vector<int> v1{};
  v1 = create_vector(); // No copy elision
  std::cout << "&v1: " << &v1 << '\n';  //@2
}
```

Why NRVO fails

❓
- ◼ Copy elision only works during object initialization, not assignment.
- ◼ Assignment sequence:
  1. Objects `v1` already exists (default constructed).
  2. Function returns temporary object.
  3. Assignment operator copies/moves temporary to `v1`.
  4. Temporary is destroyed.

Copy Elision Summary

- ☐ RVO: Eliminates temporaries - guaranteed C[+ 17]
- ☐ NRVO: Eliminates named object copies - optional
- ☐ Fails when: Assignment (not initialization), multiple return paths.
- ☐ Fallback: Move → Copy → Compile error.

📜 Return-by-Reference

Return-by-reference means the function returns a reference (alias) to an existing object rather than a copy of the object.

- Returning by reference can be useful when you want to:
    - Allow the caller to modify the returned object directly.
    - Avoid the overhead of copying large objects.

🧪 Example

```cpp
// Function to return a reference to the element at a given index
int& get_element(std::vector<int>& vec, int index) {
  return vec[index];  // Returning a reference to the element
}

int main() {
  std::vector<int> my_vector{1, 2, 3, 4, 5};

  // Get a reference to the element at index 2
  int& ref = get_element(my_vector, 2);

  // Modify the element via the reference
  ref = 10;

  // Check the new value
  std::cout << my_vector[2] << '\n';
}
```

🧪 Example

```cpp
// Function to return a const reference to the element at a given index
const int& get_element(std::vector<int>& vec, int index) {
  return vec[index];  // Returning a reference to the element
}

int main() {
  std::vector<int> my_vector = {1, 2, 3, 4, 5};

  // Get a reference to the element at index 2
  const int& ref = get_element(my_vector, 2);

  // No intention of modifying the returned reference
  std::cout << ref << '\n';
}
```

🧪 **Example**

```cpp
// Function to return a reference to a static variable
int& f() {
  static int var{1};
  std::cout << var << '\n';
  return var;
}

// usage
int main() {
  f() = 5;          // 1
  auto& ref = f();  // 5
  ref = 10;
  f();  // 10
}
```

Potential Pitfall

Returning a reference to a local variable is **UB**

```cpp
int& get_value() {
    int local_value = 10;
    return local_value;  // Dangerous
} // local_value goes out of scope

int main() {
    int& ref = get_value();
    std::cout << ref << '\n';  // UB
}
```

≡ Return Type Deduction

In C++14 and later, you can simply declare the return type of a function as `auto`, and the compiler will deduce the return type based on the return statement within the function.

```cpp
// Declaration
auto add(int a, int b);

// Definition
auto add(int a, int b) { return a + b; }

int main() {
    std::cout << add(2, 3) << '\n';
}
```

> ✍ It is recommended to clearly write the return type instead of using return type deduction.

Return Type Conversion

When a function's return expression type differs from the declared return type, C⁺ performs type conversion to match the function signature.

```cpp
int truncate_double() {
    double value = 10.75;
    return value;          // 10.75 → 10 (truncated)
}

bool to_bool() {
    int value = 0;
    return value;          // 0 → false, non-zero → true
}

char to_char() {
    int ascii = 65;
    return ascii;          // 65 → 'A'
}
```

👍
- ■ Use explicit casts for clarity.
- ■ Avoid narrowing conversions.
- ■ Consider compiler warnings.
- ■ Use appropriate return types.

Function Overloading

Function overloading is the ability to define multiple functions with the same name in the same scope, but with different parameters.

- The function called is determined at compile-time based on the number and types of arguments passed to the function.
- This allows functions to be tailored for different data types or different numbers of parameters while using a consistent name.

☰ Uniqueness of Functions

The uniqueness of a function is characterized by the list of formal parameters.

```cpp
int add(int a, int b) { return a + b; }
double add(double a, double b) { return a + b; }
double add(int a, double b) { return a + b; }
double add(double a, int b) { return a + b; }
double add(double a, double b, double c) { return a + b + c; }

int main() {
  std::cout << add(2, 3) << '\n';            // add(int, int)
  std::cout << add(2, 3.5) << '\n';          // add(int, double)
  std::cout << add(2.5, 3) << '\n';          // add(double, int)
  std::cout << add(2.5, 3.2) << '\n';        // add(double, double)
  std::cout << add(2.5, 3.2, 4.75) << '\n';  // add(double, double, double)
}
```

The return type of the function is **NOT** considered for uniqueness in function over-
loading.

```cpp
double return_value(double a) {
  return a;
}

int return_value(double a) { //Error: redefinition of 'return_value'
  return a;
}
```

☰ Overload Resolution Process ───────────────────────

1. **Exact Match** – The compiler first looks for an overload with parameters that match the types of the arguments in the function call exactly. If found, this overload is chosen.
2. **Promotion** – If an exact match isn't found, the compiler looks for a function where the argument can be promoted to match the parameter type.
3. **Standard Conversion** – If no promotions are suitable, the compiler looks for a function where the argument can undergo a standard type conversion to match the parameter type.
4. If none of the above can be applied, a compiler error is raised.

🧪 **Example** ──

```cpp
int add(int a, int b) { return a + b; }
int add(int a, float b) { return a + b; }

int main() {
  std::cout << add(2, 3) << '\n';     // add(int, int) -- exact match
  std::cout << add(2, 3.5f) << '\n';  // add(int, float) -- exact match
}
```

🧪 **Example** ──

```cpp
1  int add(int a, int b) { return a + b; }
2  int add(int a, float b) { return a + b; }
3  int add(int a, double b) { return a + b; }
4
5  int main() {
6    std::cout << add(2.5, 3) << '\n';   // add(double, int) -- ???
7    std::cout << add('h', false) << '\n';  // add(char, bool) -- ???
8  }
```

❓ **Which overloaded functions will lines 6 and 7 call?**

Default Parameters

Default parameters allow a function to be called without providing **one or more trailing parameters**

- ■ If the user does not supply an explicit argument, the default value will be used.
- ■ If the user does supply an argument, the user-supplied argument is used.

```cpp
void add(int a, int b=0, int c=0){
    std::cout << a + b + c << '\n';
}

int main(){
    add(1);        // a=1, b=0, c=0
    add(1, 2);     // a=1, b=2, c=0
    add(1, 2, 3);  // a=1, b=2, c=3
    add();         // Error: no matching function for call to 'add()'
}
```

> ### Where to Declare Default Parameters?
>
> Default parameters can be specified either in the declaration or in the definition,
> but not in both.
>
> ```cpp
> void add(int a, int b=0, int c=0); // Declaration with defaults
> void add(int a, int b=0, int c=0){ // ERROR: Defaults repeated!
>     std::cout << a + b + c << '\n';
> }
> ```

☰ Single Source of Truth Principle

Having defaults in multiple places could lead to inconsistencies. The compiler needs one
authoritative source for default values.

👍
1. Default parameters are specified ONLY in the declaration, not in the definition.
2. The definition should not repeat the default values.
3. If no separate declaration exists, defaults can be specified in the definition.

☰ Default Parameters in Overloading

Default parameters can be used in overloading but they are not considered for uniqueness.

```
// COMPILATION ERROR: These are considered identical signatures!
void func(int a, int b);        // Signature: func(int, int)
void func(int a, int b = 5);    // Signature: func(int, int) – defaults ignored!
```

≡ Default Parameters vs. Overloading

🚀 **F.51: Where there is a choice, prefer default arguments over overloading**

🧪 Example

```cpp
int return_int(int a){
  return a;
}

int return_int(int a, int b){
  return a * b;
}
```

🧪 Example

```cpp
int return_int(int a, int b=1){
  return a*b;
}
```

Stack Frame

A stack frame, often just called a **frame**, is a section of the stack segment that contains information about a single function call. When a function is called, its information is **pushed** onto the stack in a stack frame, and when the function returns, its stack frame is **popped off** the stack.

✎ **Exercise** #5

1. Analyze the stack frames for the following program (manual).
2. Analyze the stack frames for the following program (debugger).

```cpp
void f(int& x, int y, int z) {
  x += y + z;
}

int g(int a, int b) {
  int result{};
  result = a + b;
  f(result, a, b);
  return result;
}

int main() {
  int x{10};
  int y{20};
  int z{};
  z = g(x, y);
  std::cout << z << '\n';
}
```

Recursive Functions

A recursive function is a function that calls itself, either directly or indirectly, in order to achieve a task. Recursive solutions are often used for problems that can be naturally divided into subproblems of the same kind.

When designing recursive functions, there are two main components to consider:

- ☐ Base Case(s) – These are conditions under which the function will not call itself, preventing infinite recursion. The base case offers a definitive answer without further recursion.
- ☐ Recursive Case(s) – This is where the function calls itself, usually with a modified version of the original input.

🧪 Example

```cpp
long long factorial(int n) {
    if (n ≤ 1) return 1;       // Base case: 0! or 1! is 1
    return n * factorial(n-1); // Recursive call
}

int main() {
    int n{};
    std::cout << "Enter a number: ";
    std::cin >> n;
    std::cout << "Factorial of " << n << " is " << factorial(n) << '\n';
}
```

The main Function

The `main` function is the entry point of the program.

```
int main();
or
int main(int argc, char *argv[]);
```

- `argc` stands for argument count. It represents the number of command-line arguments passed to the program, including the program's name itself.
- `char *argv[]` stands for argument vector. This is an array of character pointers that represents the actual command-line arguments passed to the program.
- The `main()` function returns an `int`. By convention, `return 0;` indicates successful execution and `return 1;` (or other non-zero values) indicates an error.
  - If you don't provide a return statement, the compiler will implicitly insert `return 0;` at the end of the function.

Example

```cpp
int main(int argc, char *argv[]) {
    std::cout << "Number of arguments: " << argc << '\n';
    for (int i{0}; i < argc; i++) {
        std::cout << "Argument " << i << ": " << argv[i] << '\n';
    }
}
```

- ▣ ./lecture5 hello 1 2 world 3

Next Class

- Lecture6: Smart Pointers.