MTH 4300/4299: Programming and Computer Science II

Lecture 04: Functions, References, and Pointers

Jaime Abbariao

Functions

A function is a reusable block of code that performs a specific task.

```
return_type function_name(parameter_type parameter_name, ...) {
    // body
    return value;
}
```

Example:

```
int add(int x, int y) {
    return x + y;
}

int main() {
    int result = add(3, 5);
    std::cout << result << std::endl; // 8
    return 0;
}
```

▒▒ Functions (ii)

A function must be declared before it is called. You can either:

- Define the function above `main`
- Write a forward declaration (prototype) above `main` and define the function below

```
int add(int x, int y); // forward declaration

int main() {
    std::cout << add(3, 5) << std::endl;
    return 0;
}

int add(int x, int y) {
    return x + y;
}
```

Return types

A function's return type tells the compiler what kind of value the function gives back.

```
int square(int x) {
    return x * x;
}

double average(double a, double b) {
    return (a + b) / 2.0;
}

bool is_even(int n) {
    return n % 2 == 0;
}

char grade(int score) {
    if (score >= 90) return 'A';
    if (score >= 80) return 'B';
    if (score >= 70) return 'C';
    if (score >= 60) return 'D';
    return 'F';
}
```

The type of the `return` expression must match (or be convertible to) the declared return type.

▒▒ void

A `void` function performs an action but does not return a value.

```
void print_greeting(std::string name) {
    std::cout << "Hello, " << name << "!" << std::endl;
}

void print_line() {
    std::cout << "----------------" << std::endl;
}
```

▒▒ void (ii)

- You cannot assign the result of a `void` function to a variable.
- You can use `return;` (with no value) to exit a `void` function early.

```
void print_positive(int n) {
    if (n <= 0) {
        std::cout << "Not positive!" << std::endl;
        return; // exit early
    }
    std::cout << n << std::endl;
}
```

▒▒ Do's-and-Dont's with functions

Do:

- Give functions descriptive names: `calculate_area`, `is_prime`, `print_results`
- Keep functions short and focused on a single task
- Use parameters instead of relying on global variables

Don't:

- Don't use global variables to pass data between functions

```
// BAD: uses global variable
int result;

void add(int x, int y) {
    result = x + y;
}
```

```
// GOOD: uses return value
int add(int x, int y) {
    return x + y;
}
```

Don't:

- Don't write functions that do too many unrelated things
- Don't ignore the return value when it indicates success or failure

▒▒▒ Exercise

Write a function `int factorial(int n)` that returns `n!` (n factorial).

Recall: `0! = 1`, and `n! = n * (n-1) * ... * 1`.

Then call it from `main` to print `factorial(5)`.

Expected output:

```
120
```

References

A reference is an alias (another name) for an existing variable.

```
int x = 10;
int& ref = x; // ref is a reference to x

std::cout << x << std::endl;   // 10
std::cout << ref << std::endl; // 10

ref = 20;
std::cout << x << std::endl;   // 20 -- x changed!
```

Key rules:

- A reference must be initialized when it is declared
- A reference cannot be reassigned to refer to a different variable
- Modifying the reference modifies the original variable

```
int a = 5;
int b = 10;
int& ref = a;
ref = b;        // this does NOT make ref refer to b
                // it assigns b's value (10) to a
std::cout << a << std::endl; // 10
```

▒▒ Exercise

What does this program print?

```cpp
#include <iostream>

int main() {
    int a = 3;
    int b = 7;
    int& r = a;

    r = b;
    b = 15;

    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << r << std::endl;

    return 0;
}
```

Passing references into functions

By default, functions receive copies of their arguments (pass by value). With references, the function operates on the original variable (pass by reference).

```
// Pass by value: x is a copy
void increment_copy(int x) {
    x++;
    // the original variable is NOT changed
}

// Pass by reference: x is the original
void increment_ref(int& x) {
    x++;
    // the original variable IS changed
}

int main() {
    int a = 5;
    increment_copy(a);
    std::cout << a << std::endl; // 5 (unchanged)

    increment_ref(a);
    std::cout << a << std::endl; // 6 (changed!)
    return 0;
}
```

▒▒ Why would you want to do this?

1. Modifying the caller's variables

A function can return only one value. References let you "return" multiple results:

```
void min_max(int a, int b, int& out_min, int& out_max) {
    out_min = (a < b) ? a : b;
    out_max = (a > b) ? a : b;
}

int main() {
    int lo, hi;
    min_max(7, 3, lo, hi);
    std::cout << lo << " " << hi << std::endl; // 3 7
    return 0;
}
```

2. Avoiding expensive copies

Copying large objects (like a `std::string` or a large array) is slow. Passing by reference avoids the copy. Use `const` if you don't need to modify it:

```
void print_string(const std::string& s) {
    std::cout << s << std::endl;
}
```

▒▒ Why should you care about memory?

In Python, you never think about where data lives:

```python
def double_list(lst):
    return [x * 2 for x in lst]

nums = [1, 2, 3, 4, 5]
result = double_list(nums)
```

Python handles everything behind the scenes: allocating memory, copying objects, cleaning up when you're done. This is convenient, but it comes at a cost -- Python is slow.

In C++, you control what happens with memory. This is what makes C++ fast, but it means you need to understand how it works.

▒▒ Python vs C++: What happens when you call a function?

In Python, variables like lists and objects are always passed as references behind the scenes:

```python
def append_item(lst, item):
    lst.append(item)  # modifies the ORIGINAL list

nums = [1, 2, 3]
append_item(nums, 4)
print(nums)  # [1, 2, 3, 4] -- changed!
```

In C++, by default, functions receive a copy of the argument:

```cpp
void try_to_change(int x) {
    x = 999; // this only changes the local copy
}

int main() {
    int n = 5;
    try_to_change(n);
    std::cout << n << std::endl; // still 5!
    return 0;
}
```

C++ gives you the choice: copy or not? Python never gives you that choice.

Why does this matter?

Imagine you have a large dataset -- say, a vector of 1 million integers.

```
// BAD: copies 1 million integers every time you call this
function
double compute_average(std::vector<int> data) {
    double sum = 0;
    for (int i = 0; i < data.size(); i++) {
        sum += data[i];
    }
    return sum / data.size();
}
```

Why does this matter? (ii)

```cpp
// GOOD: passes a reference -- no copy, just as fast as accessing
the original
double compute_average(const std::vector<int>& data) {
    double sum = 0;
    for (int i = 0; i < data.size(); i++) {
        sum += data[i];
    }
    return sum / data.size();
}
```

The only difference is `const std::vector<int>&` instead of `std::vector<int>`. The `&` means "don't copy, use the original." The `const` means "promise not to modify it."

This is the kind of control that makes C++ programs orders of magnitude faster than Python.

▒▒▒ Exercise

Write a function `void swap(int& a, int& b)` that swaps the values of two integers.

Test it from `main`:

```
int x = 10, y = 20;
swap(x, y);
std::cout << x << " " << y << std::endl;
```

Expected output:

```
20 10
```

▒▒ Pointers

A pointer is a variable that stores the memory address of another variable.

```
int x = 42;
int* ptr = &x; // ptr holds the address of x

std::cout << ptr << std::endl;  // address of x (e.g. 0x7ffee...)
std::cout << *ptr << std::endl; // 42 (dereferencing: get the
value at that address)

*ptr = 100;
std::cout << x << std::endl;   // 100
```

- `int*` means "pointer to an int"
- `&x` gets the address of `x`
- `*ptr` dereferences the pointer (accesses the value it points to)

A pointer can be set to `nullptr` to indicate it points to nothing:

```
int* ptr = nullptr;
// dereferencing nullptr is undefined behavior -- never do this!
```

Dynamically allocating data

The new keyword allocates memory on the heap and returns a pointer to it.

```
int* p = new int;     // allocate one int on the heap
*p = 42;
std::cout << *p << std::endl; // 42
```

You can also allocate arrays dynamically:

```
int* arr = new int[5]; // allocate array of 5 ints on the heap

for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << " ";
}
std::cout << std::endl; // 0 10 20 30 40
```

This is useful when the size is not known at compile time:

```
int n;
std::cin >> n;
int* arr = new int[n]; // size determined at runtime
```

▒▒ Memory: The Stack

The stack is managed automatically. Each function call creates a stack frame that holds its local variables. When the function returns, its frame is removed.

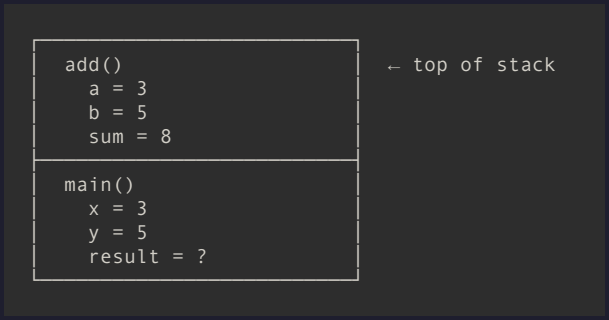
Consider this code:

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int main() {
    int x = 3;
    int y = 5;
    int result = add(x, y);
    return 0;
}
```

▒▒ Memory: The Stack (ii)

When `add` is called, the stack looks like:

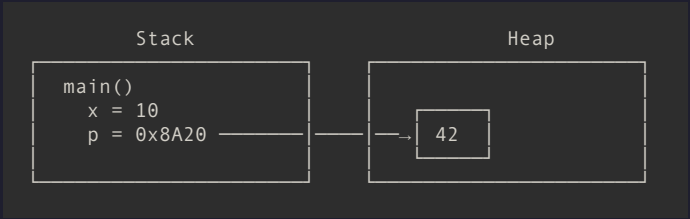


After `add` returns, its frame is gone:

▒▒ Memory: The Heap

The heap is for memory you allocate manually with `new`. It persists until you explicitly `delete` it.

```
int main() {
    int x = 10;           // stack
    int* p = new int(42); // p is on the stack, but it points to
the heap
    delete p;
    return 0;
}
```



Every variable lives at a specific memory address. You can see the address using the `&` operator:

```
int x = 42;
std::cout << &x << std::endl; // prints something like
0x7ffee3b4a8bc
```

Cleaning up dynamically allocated data

Memory allocated with `new` is not automatically freed. You must use `delete` to free it, otherwise you have a memory leak.

```
int* p = new int;
*p = 42;
// ... use p ...
delete p; // free the memory

int* arr = new int[5];
// ... use arr ...
delete[] arr; // use delete[] for arrays
```

Rules:

- Every `new` must have a matching `delete`
- Every `new[]` must have a matching `delete[]`
- After deleting, don't use the pointer again (set it to `nullptr`)
- Never `delete` something that wasn't allocated with `new`

```
int* p = new int(10);
delete p;
p = nullptr; // good practice
```

▒▒ Exercise

Write a program that:

1. Asks the user for a number `n`
2. Dynamically allocates an array of `n` integers
3. Fills the array with the values `1, 2, ..., n`
4. Prints the array
5. Frees the memory

Expected output:

```
Enter n: 5
1 2 3 4 5
```

Pointer arithmetic

When you add an integer to a pointer, it moves by that many elements (not bytes).

```
int* arr = new int[5];
for (int i = 0; i < 5; i++) {
    arr[i] = (i + 1) * 10;
}

int* p = arr;          // points to arr[0]
std::cout << *p << std::endl;       // 10
std::cout << *(p + 1) << std::endl; // 20
std::cout << *(p + 3) << std::endl; // 40
```

In fact, `arr[i]` is equivalent to `*(arr + i)`.

You can also use pointer arithmetic to iterate:

```
int* end = arr + 5;
for (int* p = arr; p != end; p++) {
    std::cout << *p << " ";
}
std::cout << std::endl; // 10 20 30 40 50

delete[] arr;
```

▒▒ Exercise

Given the following code, what does the program print?

```cpp
#include <iostream>

int main() {
    int arr[] = {5, 10, 15, 20, 25};
    int* p = arr + 1;

    std::cout << *p << std::endl;
    std::cout << *(p + 2) << std::endl;
    std::cout << p[-1] << std::endl;
    std::cout << p[3] << std::endl;

    return 0;
}
```

▒▒ Lab

Write a program that manages a dynamically allocated array of student grades:

1. Ask the user for the number of students `n`
2. Dynamically allocate an array of `n` doubles
3. Read `n` grades from the user
4. Write a function `double average(const double* grades, int n)` that computes the average using pointer arithmetic (no `[]` operator)
5. Write a function `void curve(double* grades, int n, double amount)` that adds `amount` to every grade using pointer arithmetic
6. Print the average before and after curving by 5 points
7. Free the memory