

# C++

## Functions & Pointers

---

*originally developed by Weiguang Cui,  
refined by Alexander Knebe*



Functions



inline



argc/argv



Arrays



Pointers



Memory



File I/O

# Function — Anatomy



```
type name ( type param1, type param2, ... ) {  
    statements;  
    return value;  
}
```

## type

Return type. Must match the return value. Omitting it is an error in modern C++.

## parameters

Zero or more typed inputs, e.g. `int x`. No need to re-declare them inside the body.

## name

Unique identifier. Choose something meaningful — it's how you call the function.

## statements

The function body in `{}`. Always include `return` unless the type is `void`.

# Function — Where to Put It?

A function can appear anywhere in the file — but it must be declared before it is called.



```
void odd(int x); // prototype: must match actual function

int main() {
    odd(3);      // ready to use function
}

void odd(int x) { // actual definition of function
    // ...
}
```

## Try it!

Can you change the type or param name in the forward declaration?

Can you declare a function inside another function?

If A calls C (which calls B), is declaring only C before A enough? (Inherited declaration)

# Function — inline



```
inline string concatenate(const string& a, const string& b) {  
    return a + b;  
}
```

## Normal Function

Call overhead: save registers, push args to stack, jump to function address, return. Fine for large functions.

## Inline Function

Compiler replaces the call site with the function body directly. Zero call overhead — ideal for tiny functions.

## Restriction

Still a real function — cannot be declared or defined inside another function.

## In Practice

From your code's perspective there is no difference. The distinction is only at compile/run time.

# Calling, Recursion & References



```
y = your_func(x);           // call

// pass by value - x inside func is a copy
int your_func(int x) { x += 10; }

// pass by reference - x IS the original variable
int your_func(int& x) { x += 10; }
```

## Key Questions

Pass an int arg but the param is float — what happens?

Recursion: a function calling itself. Can you make an infinite loop?

Modify x inside your\_func — does x change in main() without &?

Add & to the parameter — try again. What changes?



```
// Recursive example: factorial
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1); // calls itself!
}
```

Every recursive call must have a base case to stop — otherwise it's an infinite loop and will crash with a stack overflow.

# main() — argc & argv



```
int main()                { /* no args */ }  
int main(int argc, char* argv[]) { /* C-style */ }  
int main(int argc, char** argv)  { /* equivalent */ }
```

## argc & argv explained

**argc**

ARGument Count. Number of args including the program name. Always  $\geq 1$ .

**argv**

ARGument Vector. Array of C-strings. `argv[0]` = program name, `argv[1]` = first user arg, ...

**argv[0]**

Always the program name itself, not a user-supplied argument.

## Try it!

Can `main()` return void or other types?

Run with no args — what is `argc`? What is `argv[0]`?

Run: `./myprog hello 42` — what are `argc` and `argv[1]`, `argv[2]`?

Print all `argv` values in a loop.

# Exercise I — Calculator Function

Define ONE function that performs  $+$ ,  $-$ ,  $\times$ , and  $\div$ .

Call it from `main()` to compute  $a+b$ ,  $a-b$ ,  $a\times b$  and  $a\div b$ , then print all four results.

Variables `a` and `b` can be hard-coded or read from the keyboard with `cin`.

## Hints & Notes

- How do you tell the function which operation to perform? Consider adding an extra parameter.
- Think about what the function should return and what its type should be.
- What happens when  $b = 0$  for division? Can you guard against it?

## Exercise II — Recursive Sum

Use the recursion feature of functions to calculate:

$$1 + 2 + 3 + \dots + 1000$$

Write a recursive function `sum(n)` that returns the sum of 1 through `n`.

### Hints & Notes



Base case:  $\text{sum}(1) = 1$ . Recursive case:  $\text{sum}(n) = n + \text{sum}(n-1)$ .



Compare the result to the closed-form formula  $n \times (n+1) / 2$ . Do they match?

## Exercise III — Sum Range via argc/argv

Extend Exercise II to use argc and argv in main().

Read two numbers a and b from the command line, then calculate:

$$a + (a+1) + (a+2) + \dots + b$$

### Hints & Notes

- argv values are strings — use atoi() or stoi() to convert them to integers.
- argv[1] is the first user argument (not the program name).
- What should you print if the user passes too few arguments?

# Arrays

# & Pointers

Memory layout, address-of, dereferencing, dynamic allocation

# Arrays

An array is a series of elements of the same type in contiguous memory, addressable by index.

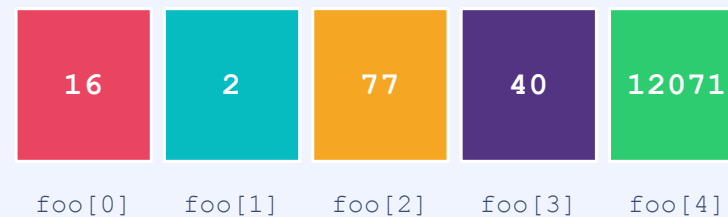


```
type name[elements];

int foo[5] = { 16, 2, 77, 40, 12071 }; // explicit init
int baz[5] = { };                      // zero-init
int bar[]  = { 16, 2, 77, 40, 12071 }; // size inferred

// Get the number of elements:
int len = sizeof(foo) / sizeof(foo[0]);
```

## Memory layout:



Access: foo[0]...foo[4] | ⚠️ foo[5] is undefined behaviour!

- sizeof(foo) gives bytes, not elements — divide by sizeof(foo[0]).
- Accessing beyond the array bounds is undefined behaviour — no crash guarantee.

# Multi-Dimensional Arrays

```
int dummy[3][5]; // 3 rows, 5 columns

// Initialise:
int grid[2][3] = { {1,2,3}, {4,5,6} };

// Access:
grid[1][2] → 6 (row 1, col 2)

// Size:
int rows = sizeof(grid) / sizeof(grid[0]);
int cols = sizeof(grid[0]) / sizeof(grid[0][0]);
```

## 2D layout (row-major):

|       |       |       |       |
|-------|-------|-------|-------|
| 1     | 2     | 3     | row 0 |
| 4     | 5     | 6     | row 1 |
| col 0 | col 1 | col 2 |       |

## Questions

Which dimension is rows and which is columns?

What does `cout << dummy` print? (It's not the values!)

How do you get rows and columns separately with `sizeof`?

How do you iterate over all elements with nested for loops?

# Array as a Function Parameter



```
#include <iostream>
using namespace std;

void printarray(int arg[], int length) {
    for (int n = 0; n < length; ++n)
        cout << arg[n] << ' ';
    cout << '\n';
}

int main() {
    int first[] = {5, 10, 15};
    int second[] = {2, 4, 6, 8, 10};
    printarray(first, 3);
    printarray(second, 5);
}
```

## Key Points

### **int arg[]**

Arrays decay to a pointer — no size info. Always pass length separately.

### **Modifying**

Changes to `arg[]` inside the function affect the original array in `main()`. Why?

### **2D array**

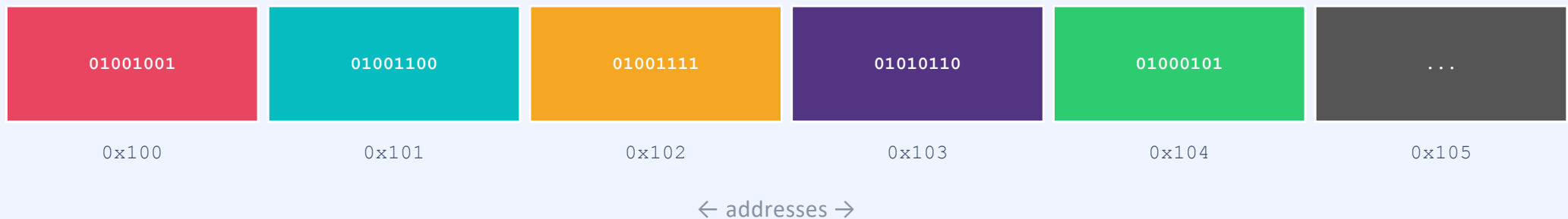
`void f(int a[][5], int rows)` — the column size must be specified.

### **Try it**

Modify values inside `printarray` and check if they changed in `main()`.

# Pointers — Memory & Addresses

Every variable lives at a memory address. A pointer is a variable that stores an address.



## & — Address-of

```
foo = &myvar;
```

Gets the memory address of myvar and stores it in the pointer foo.

## \* — Dereference

```
baz = *foo;
```

Follows the pointer — reads the value stored at the address foo holds.

# Pointer — Declaration & Why Type Matters



```
type* name;           // pointer declaration

int    myvar = 42;
int*   foo = &myvar;  // foo holds the address of myvar
int    baz = *foo;    // baz = 42 (value at address)

// Type matters for pointer arithmetic:
float mtv = 1.3f;
int*   poi = &mtv;    // ERROR: type mismatch!
float* pf = &mtv;     // OK
```

## Why does the type matter?

The type tells the compiler how many bytes to read when dereferencing. `int*` reads 4 bytes; `char*` reads 1.

Pointer arithmetic uses the type size: `(ptr + 1)` advances by `sizeof(*ptr)` bytes — not by 1 byte.

Assigning `&floatVar` to an `int*` is a type error — the byte patterns differ.



```
int  a, b;
cout << &a << " " << &b;  // how close are the addresses?

// Accessing b through a's pointer (dangerous but instructive):
int* pa = &a;
cout << *(pa + 1);         // might be b — try it!
```

# Pointer & Array — Two Notations



```
int  myarray[20];
int* mypointer = myarray;    // pointer to first element

// These four are all equivalent:
myarray[5]    = 0;
*(myarray+5)  = 0;
mypointer[5]  = 0;
*(mypointer+5) = 0;

// Pointer arithmetic steps by sizeof(element) bytes:
long long int big[20];
*(big + 5) = 0;                // steps by 8 bytes each time
```

## Try it!

Print `mypointer[1]` — does it give the same value as `myarray[1]`?

What is the value (address) of `myarray` itself — not `myarray[0]`?

Redefine as `long long int myarray[20]`. Does `*(myarray+5)` still give the 6th element?

The step size of pointer arithmetic equals `sizeof(element)`, not 1 byte.

An array name is a constant pointer to its first element — you cannot reassign it (`myarray = somePtr` is an error), but a separate pointer variable can be moved freely.

# Pointer & Dynamic Memory



```
int* foo;

// Allocate an array of 5 ints on the heap
foo = new int[5];

// nothrow variant: returns nullptr on failure instead of
throwing
foo = new (nothrow) int[5];

// Re-allocate (address will likely differ!)
foo = new int[8];

delete    foo;    // frees one element  (use for non-array
new)
delete[] foo;    // frees entire array (use this for
new[])
```

## delete vs delete[]

### delete

For single objects: `int* p = new int; delete p;`

### delete[]

For arrays: `int* p = new int[5]; delete[] p;` Using `delete` (not `delete[]`) on an array is undefined behaviour.

### After delete

The pointer still holds the old address — access is undefined. Set to `nullptr` after deleting.

Also available from `<cstdlib>`: `malloc` / `calloc` / `realloc` / `free` — C-style allocation. In C++, prefer `new/delete` or smart pointers.

# delete vs delete[] — Live Demo



```
#include <iostream>
int main() {
    int* p = new int[5];
    for (int i = 0; i < 5; i++) p[i] = 1 + i;

    // Print: addr *p p[1] *(p+2) p[3] p[4]
    std::cout << p << " " << *p << " " << p[1] << " " << *(p+2)
               << " " << p[3] << " " << p[4] << "\n";

    delete p;                // ⚠️ wrong for array — UB!
    p = new int[6];
    std::cout << p << " " << *p << " " << p[1] << "... \n";
    // addresses may differ — values unreliable!

    delete[] p;              // correct
    p = new int[7];
    std::cout << p << " " << *p << " " << p[1] << "... \n";
    // notice how the address changes each time
    return 0;
}
```

## What to observe

Does the address change after each delete + new? Why?

After delete (not delete[]), can you still read p[1]? Is the value trustworthy?

After delete[], same question. The memory is released — never trust those values.

Pay attention to whether the first element address is the same each time you allocate.

# char Array vs string — Pointer Behaviour

```


#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

int main() {
    char myarray[4] = {'a','b','c','d'};
    char* p = myarray;
    cout << typeid(myarray).name() << "\n"; // type name
    cout << myarray << "\t" << (long long)p << "\t" << *(p+2) <<
"\n";
    cout << (long long)myarray << "\t" << p << "\t" << *(p+3) <<
"\n";

    string mystr = "abcd";
    string* p2 = &mystr;
    cout << typeid(mystr).name() << "\n";
    cout << &mystr << "\t" << (long long)p2 << "\n";
}

```

## char vs string

### char[]

A raw array. cout prints chars until \0. You can pointer-arithmetic over individual bytes.

### string

A class. &mystr gives the object address. Use mystr.c\_str() to get a char\* to its contents.

### Re-init

char myarray[4] = {'d','c','b','a'}; after the first init is NOT allowed.

### Index

myarray[3] works for both; string also has .at(i), .substr(), .size() etc.

Observe how cout treats char\* (prints string content) differently from long long (prints numeric address). The cast (long long) forces numeric output.

# Returning Multiple Values via Pointers & References



```
void my_func1(int* a) { *a += *a; } // pointer param
void my_func1(int& a) { a += a; } // reference param
void my_func2(int a) { a += a; } // value param
(copy)

int main() {
    int x = 5;
    my_func1(&x); // or my_func1(x) for ref version
    // x is now 10 - changed in-place

    int y = 5;
    my_func2(y);
    // y is still 5 - only the copy changed
}
```

## Returning an array

### Via pointer

Declare array as static inside the function, return a pointer. (Static persists after the function returns.)

### Via struct

Wrap multiple return values in a struct and return it by value — clean and safe.

### Global vars

Technically possible but avoid — creates hidden coupling between functions.

### Try it

int\* vs int& param: what is the difference in the call site syntax?

To return an array: (1) use a static local array and return a pointer, or (2) use a struct. Never return a pointer to a local (non-static) variable — it's destroyed when the function returns!

# Exercise IV — Return a Matrix from a Function

Try using both approaches to return a matrix variable (using the Eigen or matrix.h library) from a function:

1. Return a pointer to a static local variable.
2. Wrap the matrix in a struct and return the struct.

## Hints & Notes



Which approach is safer? Which is more readable?



What happens if you try to return a pointer to a non-static local matrix?

# Exercise I — Pointer & String

Given the string "A string."

Line 1: print the letter at index 0, the pointer address, and the letter 't'.

Then advance the pointer by +2.

Line 2: print the new pointer address, and the letters 'r' and 'g' via the pointer.



```
string s = "A string.";
char* p = &s[0];      // or: const char* p = s.c_str();

// Line 1: index 0, address, letter 't'
cout << p[0] << " " << (long long)p << " " << p[2] << "\n";

p += 2;               // advance pointer by 2

// Line 2: address, 'r', 'g'
cout << (long long)p << " " << p[3] << " " << p[5] << "\n";
```

Remember: advancing a `char*` by 1 steps by 1 byte (1 char). `p[n]` is shorthand for `*(p+n)`.

## Exercise II — Return Multiple Values from a Function

Revisit Exercise I from the Functions section: a function that computes  $a+b$ ,  $a-b$ ,  $a \times b$  and  $a \div b$ .

How can you return all four results at once?

Clue: return a pointer from the function. What must the return type of the function be?

### Hints & Notes



Option 1: return a pointer to a static int array of 4 elements.



Option 2: define a struct `{ int add, sub, mul; double div; }` and return it.



Which option requires less setup code? Which is safer for long-term use?

# Exercise III — Raw Pointer Exploration

Move your pointer to a random address in memory and try to print the value.

Does the value make sense? Try different types — does the interpretation change?

Bonus: can you print the value at the very first or last byte of your process's memory space? (Can you even get that address?)

## Hints & Notes

- Use (long long) to print addresses as numbers.
- Accessing arbitrary memory addresses causes undefined behaviour — expect crashes or garbage values.
- This exercise shows why unchecked pointer use is dangerous. Modern C++ prefers smart pointers.