

C++ for Absolute Beginners

Solutions to All 20 Exercises — Organised by Topic

Each solution includes a complete working program and an explanation of the key concepts. Try to solve each exercise yourself first!

Table of Contents

Section 1 — Very Basics (solutions 1–9)

- 01. Simple Function (Functions) ★★★
- 02. Temperature Converter (Functions) ★★★
- 03. Power Function (Functions) ★★★
- 04. Grade Classifier (Functions) ★★★
- 05. Pointer to a Variable (Pointers) ★★★
- 06. Pointer to an Array (Pointers) ★★★
- 07. Pointer to a Structure (Pointers) ★★★
- 08. Matrix Construction and Arithmetic (Matrices) ★★
- 09. Transpose and Symmetry Check (Matrices) ★★

Section 2 — Operators & Syntax (solutions 10–30)

- 10. Determinant, Inverse and Linear System (Matrices) ★★★
- 11. Single Value on the Heap (Dynamic Memory) ★
- 12. Dynamic 1D Array (Dynamic Memory) ★★
- 13. Dynamic 2D Array (Dynamic Memory) ★★
- 14. Dynamic 3D Array (Dynamic Memory) ★★★
- 15. Dynamic Array of Objects (Dynamic Memory) ★★★
- 16. Swap via Pointers (Pointers) ★★
- 17. Dynamic String Buffer (Dynamic Memory) ★★★
- 18. Pointer Arithmetic on a Heap Array (Dynamic Memory) ★★★
- 19. Resizable Array (Dynamic Memory) ★★★

Section 1 — Very Basics

Solution 1 of 20

Simple Function

★★★

Topic: Functions

Solution Code

```
#include <iostream>

int square(int n) {
    return n * n;
}

int main() {
    for (int i = 1; i <= 5; i++) {
        std::cout << square(i) << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Explanation

Define the function before main so the compiler knows about it. The function takes one int parameter and returns an int.

Temperature Converter



Topic: Functions / Math

Solution Code

```
#include <iostream>

double celsiusToFahrenheit(double c) {
    return c * 9.0 / 5.0 + 32.0;
}

int main() {
    double celsius;
    std::cout << "Celsius: ";
    std::cin >> celsius;
    std::cout << "Fahrenheit: "
    << celsiusToFahrenheit(celsius)
    << std::endl;
    return 0;
}
```

Explanation

Using 9.0 and 5.0 (not 9 and 5) ensures floating-point division. The function returns a double for decimal precision.

Power Function



Topic: Functions / Nested Calls

Solution Code

```
#include <iostream>

int multiply(int a, int b) {
    return a * b;
}

int power(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result = multiply(result, base);
    }
    return result;
}

int main() {
    std::cout << "2^8 = " << power(2, 8) << std::endl;
    std::cout << "3^4 = " << power(3, 4) << std::endl;
    return 0;
}
```

Explanation

*multiply is a helper that power calls repeatedly — this is a nested function call. Defining functions in dependency order (helper first, caller second) means no forward declarations are needed. power delegates the actual multiplication to multiply rather than using * directly, keeping each function focused on one job.*

Grade Classifier



Topic: Functions / Nested Calls

Solution Code

```
#include <iostream>

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

bool isPassing(int score) {
    return letterGrade(score) != 'F';
}

int main() {
    int score;
    std::cout << "Score: ";
    std::cin >> score;
    std::cout << "Grade: " << letterGrade(score) << std::endl;
    std::cout << "Passed: " << (isPassing(score) ? "Yes" : "No") << std::endl;
    return 0;
}
```

Explanation

isPassing calls *letterGrade* instead of duplicating its threshold logic — this is the key benefit of nested/composed function calls. If the grading boundaries ever change, you only update *letterGrade*. The ternary operator (*isPassing(...)* ? "Yes" : "No") converts the *bool* return value into a readable string.

Pointer to a Variable



Topic: Pointers / Single Variable

Solution Code

```
#include <iostream>

int main() {
    int x = 10;
    int* p = &x; // p holds the address of x

    std::cout << "Value via pointer: " << *p << std::endl;

    *p = 99; // change x through the pointer

    std::cout << "After change -- x: " << x
    << " *p: " << *p << std::endl;
    return 0;
}
```

Explanation

*& (address-of) gives the memory address of a variable. * (dereference) follows the pointer to the value it points at. Because `p` points to `x`, writing `*p = 99` changes `x` directly — they refer to the same memory location. This is why both print 99 after the assignment.*

Pointer to an Array



Topic: Pointers / Array

Solution Code

```
#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* p = arr; // array name decays to pointer to first element

    std::cout << "Elements: ";
    for (int i = 0; i < 5; i++)
        std::cout << *(p + i) << " ";
    std::cout << std::endl;

    *(p + 2) *= 2; // double the third element

    std::cout << "After change: ";
    for (int i = 0; i < 5; i++)
        std::cout << *(p + i) << " ";
    std::cout << std::endl;

    return 0;
}
```

Explanation

When you write `int* p = arr`, the array name decays to a pointer to its first element — no `&` needed. `*(p + i)` is exactly equivalent to `arr[i]`; the compiler generates identical code. Writing `*(p + 2) *= 2` modifies `arr[2]` directly because `p` and `arr` point to the same memory. This makes the relationship between arrays and pointers explicit: in C++, array indexing is pointer arithmetic in disguise.

Pointer to a Structure



Topic: Pointers / Struct

Solution Code

```
#include <iostream>
#include <cmath>

struct Point {
    double x;
    double y;
};

int main() {
    Point pt;
    Point* p = &pt;

    p->x = 3.0; // arrow operator: same as (*p).x = 3.0
    p->y = 4.0;

    double dist = std::sqrt(p->x * p->x + p->y * p->y);

    std::cout << "Point: (" << p->x << ", " << p->y << ")" << std::endl;
    std::cout << "Distance from origin: " << dist << std::endl;
    return 0;
}
```

Explanation

The arrow operator `->` is the standard way to access struct (or class) members through a pointer. It combines dereferencing and member access into one step: `p->x` means `(*p).x`. `std::sqrt` from computes the square root. For a 3-4-5 right triangle the hypotenuse is exactly 5.

Matrix Construction and Arithmetic



Topic: Matrices / Construction / + * scalar

Solution Code

```
#include <iostream>
#include "matrix.h"

int main() {
    math::matrix<double> A = {{1,2},{3,4}};
    math::matrix<double> B = {{5,6},{7,8}};

    std::cout << "A:
" << A;
    std::cout << "A+B:
" << A + B;
    std::cout << "A*B:
" << A * B;
    std::cout << "A*2:
" << A * 2.0;

    return 0;
}
```

Explanation

*The initializer-list constructor lets you write matrices exactly as you would on paper. All four arithmetic operators (+, -, *, and scalar *) are overloaded, so expressions look natural. The << operator prints each row tab-separated with a newline at the end — no manual loop needed. Note the math:: namespace prefix required by the header.*

Transpose and Symmetry Check



Topic: Matrices / Transpose / IsSymmetric

Solution Code

```
#include <iostream>
#include "matrix.h"

int main() {
    math::matrix<double> M = {{1,2,3},{4,5,6},{7,8,9}};
    math::matrix<double> S = {{1,2,3},{2,5,6},{3,6,9}};

    std::cout << "M:
" << M;
    std::cout << "Transpose of M:
" << ~M;
    std::cout << "M symmetric: "
<< (M.IsSymmetric() ? "Yes" : "No") << "
";

    std::cout << "S:
" << S;
    std::cout << "Transpose of S:
" << ~S;
    std::cout << "S symmetric: "
<< (S.IsSymmetric() ? "Yes" : "No") << "
";

    std::cout << "Transpose of S equals S: "
<< (~S == S ? "Yes" : "No") << "
";

    return 0;
}
```

Explanation

The `~` (tilde) operator is overloaded to return the transpose — an unconventional but compact choice. `IsSymmetric()` compares every $m(i,j)$ against $m(j,i)$ internally. The `==` operator is also overloaded for element-wise comparison, so `~S == S` is a clean way to verify symmetry programmatically. `M` is not symmetric because its off-diagonal elements are not mirrored.

Section 2 — Operators & Syntax

Solution 10 of 20

Determinant, Inverse and Linear System

★★★

Topic: Matrices / Det / Inverse / Solve

Solution Code

```
#include <iostream>
#include "matrix.h"

int main() {
    math::matrix<double> A = {{ 2, 1, -1},
    {-3, -1, 2},
    {-2, 1, 2}};

    math::matrix<double> b = {{8}, {-11}, {-3}};

    std::cout << "Det(A): " << A.Det() << "

    ";

    std::cout << "Inverse of A:
    " << !A;

    math::matrix<double> x = A.Solve(b);
    std::cout << "Solution x:
    " << x;

    return 0;
}
```

Explanation

Det() uses Gaussian elimination with partial pivoting — a determinant of -1 means the matrix is well-conditioned and has a unique solution. The *!* operator calls *Inv()* which also uses pivoted row reduction. *Solve()* is more numerically stable than computing *!A * b* directly because it avoids the full inversion step. The system $2x+y-z=8$, $-3x-y+2z=-11$, $-2x+y+2z=-3$ has the exact solution $x=2$, $y=3$, $z=-1$.

Single Value on the Heap



Topic: Dynamic Memory / new / delete / Single Variable

Solution Code

```
#include <iostream>

int main() {
    double* p = new double;
    *p = 3.14159;

    std::cout << "Value: " << *p << std::endl;
    std::cout << "Doubled: " << (*p * 2) << std::endl;

    delete p;
    p = nullptr;

    return 0;
}
```

Explanation

new double allocates `sizeof(double)` bytes on the heap and returns a `double*`. The dereference operator `*` is used both to write (`*p = 3.14159`) and read (`*p`) the value. Setting `p = nullptr` after `delete` is good practice — any accidental later use of `p` will cause a clean crash rather than silent corruption of freed memory.

Dynamic 1D Array



Topic: Dynamic Memory / new[] / delete[] / 1D Array

Solution Code

```
#include <iostream>

int main() {
    int n;
    std::cout << "Size: ";
    std::cin >> n;

    int* arr = new int[n];

    for (int i = 0; i < n; i++)
        arr[i] = (i + 1) * (i + 1);

    std::cout << "Array: ";
    int sum = 0;
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
        sum += arr[i];
    }
    std::cout << std::endl;
    std::cout << "Sum: " << sum << std::endl;

    delete[] arr;
    arr = nullptr;

    return 0;
}
```

Explanation

new int[n] allocates a contiguous block of n integers whose size is decided at runtime — impossible with a stack array. delete[] (with brackets) is essential here: it tells the runtime to free the entire block. Plain delete would only free the first element and leave the rest leaked.

Dynamic 2D Array



Topic: Dynamic Memory / new[] / delete[] / 2D Array

Solution Code

```
#include <iostream>
#include <iomanip>

int main() {
    const int rows = 3;
    const int cols = 4;

    int** m = new int*[rows];
    for (int r = 0; r < rows; r++)
        m[r] = new int[cols];

    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++)
            m[r][c] = r * 10 + c;

    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++)
            std::cout << std::setw(2) << std::setfill('0')
            << m[r][c] << " ";
        std::cout << std::endl;
    }

    for (int r = 0; r < rows; r++)
        delete[] m[r];
    delete[] m;

    return 0;
}
```

Explanation

A 2D dynamic array is a pointer to an array of pointers (`int**`). Each row is an independently allocated 1D array. Freeing must happen in reverse order: each row first, then the outer pointer array. Doing it the wrong way around loses the row addresses before you can free them, causing a memory leak.

Dynamic 3D Array



Topic: Dynamic Memory / new[] / delete[] / 3D Array

Solution Code

```
#include <iostream>
#include <iomanip>

int main() {
    const int layers = 2;
    const int rows = 3;
    const int cols = 4;

    int*** arr = new int**[layers];
    for (int l = 0; l < layers; l++) {
        arr[l] = new int*[rows];
        for (int r = 0; r < rows; r++)
            arr[l][r] = new int[cols];
    }

    for (int l = 0; l < layers; l++)
        for (int r = 0; r < rows; r++)
            for (int c = 0; c < cols; c++)
                arr[l][r][c] = l * 100 + r * 10 + c;

    for (int l = 0; l < layers; l++) {
        std::cout << "Layer " << l << ":" << std::endl;
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++)
                std::cout << std::setw(3) << std::setfill('0')
                << arr[l][r][c] << " ";
            std::cout << std::endl;
        }
        if (l < layers - 1) std::cout << std::endl;
    }

    for (int l = 0; l < layers; l++) {
        for (int r = 0; r < rows; r++)
            delete[] arr[l][r];
        delete[] arr[l];
    }
    delete[] arr;

    return 0;
}
```

Explanation

*int*** is a pointer to a pointer to a pointer — three levels of indirection matching three dimensions. The allocation mirrors the declaration: layers first, then rows per layer, then columns per row. Freeing reverses this exactly: columns (innermost) first, then rows, then layers.*

Dynamic Array of Objects



Topic: Dynamic Memory / new[] / delete[] / Objects / Structs

Solution Code

```
#include <iostream>

struct Particle {
    double x;
    double y;
    double mass;
};

int main() {
    int n;
    std::cout << "n = ";
    std::cin >> n;

    Particle* p = new Particle[n];

    for (int i = 0; i < n; i++) {
        p[i].x = i * 1.0;
        p[i].y = i * 2.0;
        p[i].mass = (i + 1) * 0.5;
    }

    double total_mass = 0.0, sum_xm = 0.0, sum_ym = 0.0;
    for (int i = 0; i < n; i++) {
        total_mass += p[i].mass;
        sum_xm += p[i].x * p[i].mass;
        sum_ym += p[i].y * p[i].mass;
    }

    std::cout << "Total mass: " << total_mass << std::endl;
    std::cout << "Centre of mass: x=" << sum_xm / total_mass
    << " y=" << sum_ym / total_mass
    << std::endl;

    delete[] p;
    p = nullptr;

    return 0;
}
```

Explanation

new Particle[n] allocates n Particle objects contiguously and calls the default constructor on each. Members are accessed with dot notation $p[i].x$ just like a normal array of structs. `delete[]` calls the destructor on every element before freeing the block. The centre of mass formula $\text{sum}(x*m)/\text{sum}(m)$ is a weighted average — each position contributes proportionally to its mass.

Swap via Pointers



Topic: Pointers / Functions / Dynamic Memory

Solution Code

```
#include <iostream>

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int x = 5, y = 9;
    std::cout << "Before: x=" << x << " y=" << y << std::endl;
    swap(&x, &y);
    std::cout << "After: x=" << x << " y=" << y << std::endl;
    return 0;
}
```

Explanation

Passing pointers to a function lets it modify the caller's variables directly. The temporary variable is essential — without it, assigning `*a = *b` first would overwrite the original value of `*a` before it could be saved. This is the classic swap pattern and the foundation of many sorting algorithms.

Dynamic String Buffer



Topic: Dynamic Memory / new[] / delete[] / char

Solution Code

```
#include <iostream>

int main() {
    int n;
    std::cout << "n = ";
    std::cin >> n;

    char* buf = new char[n + 1];

    for (int i = 0; i < n; i++)
        buf[i] = 'A' + (i % 26);
    buf[n] = '\0';

    std::cout << buf << std::endl;

    delete[] buf;
    buf = nullptr;
    return 0;
}
```

Explanation

A C-style string is a char array terminated by '\0'. The +1 in new char[n+1] reserves space for that terminator — forgetting it causes undefined behaviour when anything reads the string. The modulo trick (i % 26) cycles through the 26 letters endlessly. std::cout knows to stop printing when it hits the null terminator.

Pointer Arithmetic on a Heap Array



Topic: Dynamic Memory / Pointers / Arithmetic

Solution Code

```
#include <iostream>
#include <cmath>

int main() {
    int n = 8;
    double* arr = new double[n];

    for (int i = 0; i < n; i++)
        *(arr + i) = std::pow(2.0, i);

    std::cout << "Forward: ";
    for (int i = 0; i < n; i++)
        std::cout << *(arr + i) << " ";
    std::cout << std::endl;

    std::cout << "Reverse: ";
    double* end = arr + n - 1;
    for (int i = 0; i < n; i++, end--)
        std::cout << *end << " ";
    std::cout << std::endl;

    delete[] arr;
    return 0;
}
```

Explanation

**(arr + i) is exactly equivalent to arr[i] — the compiler generates the same code. Using pointer arithmetic explicitly makes the underlying memory model visible. The reverse traversal starts a pointer at the last element (arr + n - 1) and decrements it each iteration. Both the index and the pointer are updated in the same for-loop update clause, which is valid C++.*

Resizable Array



Topic: Dynamic Memory / new[] / delete[] / Growth

Solution Code

```
#include <iostream>

int main() {
    int capacity = 4;
    int* arr = new int[capacity];
    for (int i = 0; i < capacity; i++)
        arr[i] = i + 1;

    // Grow to capacity 8
    int newCap = 8;
    int* newArr = new int[newCap];
    for (int i = 0; i < capacity; i++)
        newArr[i] = arr[i];
    delete[] arr;
    arr = newArr;

    for (int i = capacity; i < newCap; i++)
        arr[i] = i + 1;

    for (int i = 0; i < newCap; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;

    delete[] arr;
    return 0;
}
```

Explanation

This is the manual equivalent of what `std::vector` does automatically when you `push_back` beyond its capacity. The key steps are: allocate a larger block, copy old data, delete the old block, reassign the pointer. After `delete[] arr`, the old pointer is invalid — we immediately replace it with `newArr` so we never accidentally use the freed memory. This is exactly why `std::vector` is preferred: it handles all of this safely and automatically.