MTH 4300/4299: Programming and Computer Science II

Lecture 05: Pointers, arrays, and vectors

Jaime Abbariao

Arrays

An array is a fixed-size collection of elements of the same type, stored in contiguous memory.

```
int grades[5]; // declares an array of 5 ints (uninitialized)

int scores[4] = {90, 85, 77, 92}; // declares and initializes

int nums[] = {1, 2, 3}; // size is inferred as 3
```

Access elements using an index starting at 0:

```
int scores[4] = {90, 85, 77, 92};

std::cout << scores[0] << std::endl; // 90 (first element)
std::cout << scores[3] << std::endl; // 92 (last element)

scores[1] = 100; // modify the second element
```

▒▒ Arrays (ii)

You can loop through an array using its size:

```cpp
int arr[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << " ";
}
std::cout << std::endl; // 10 20 30 40 50
```

▒▒ Arrays (iii)

Important things to know about arrays:

- The size must be known at compile time (a constant, not a variable)
- There is no bounds checking — accessing `arr[5]` on a size-5 array is undefined behavior
- Arrays do not know their own size — you must track it yourself
- When passed to a function, arrays decay to pointers (they lose their size information)

```cpp
// This function has no way to know how big arr is
void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

These limitations are why we prefer `std::vector` in most situations.

std::vector

`std::vector` is a dynamic array from the standard library. It manages its own memory and knows its size.

```
#include <vector>

std::vector<int> nums;              // empty vector
std::vector<int> scores = {90, 85, 77}; // initialized with values
std::vector<double> vals(5, 0.0);   // 5 elements, all 0.0
```

Add and access elements:

```
std::vector<int> v = {10, 20, 30};

v.push_back(40);           // adds 40 to the end
std::cout << v[0] << std::endl;    // 10
std::cout << v.size() << std::endl; // 4
std::cout << v.back() << std::endl; // 40
```

Loop through a vector:

```cpp
std::vector<int> v = {10, 20, 30, 40, 50};

// index-based loop
for (int i = 0; i < v.size(); i++) {
    std::cout << v[i] << " ";
}
std::cout << std::endl;

// range-based for loop (preferred)
for (int x : v) {
    std::cout << x << " ";
}
std::cout << std::endl;
```

▒▒ std::vector (iii)

Other useful operations:

```
std::vector<int> v = {5, 10, 15, 20};

v.pop_back();               // removes last element → {5, 10, 15}
v.insert(v.begin() + 1, 99); // insert 99 at index 1 → {5, 99, 10,
15}
v.erase(v.begin());         // remove first element → {99, 10, 15}
v.clear();                  // remove all elements
std::cout << v.empty() << std::endl; // 1 (true)
```

▒▒ std::vector (iv)

Why use `std::vector` over arrays?

- Dynamic size — grows and shrinks as needed
- Knows its size — `v.size()` always works
- Automatic memory management — no `new`/`delete` needed
- Safe access — `v.at(i)` throws an exception if `i` is out of bounds
- Works with functions — pass by reference without losing size info

```
void print_vector(const std::vector<int>& v) {
    for (int x : v) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
}
```

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.

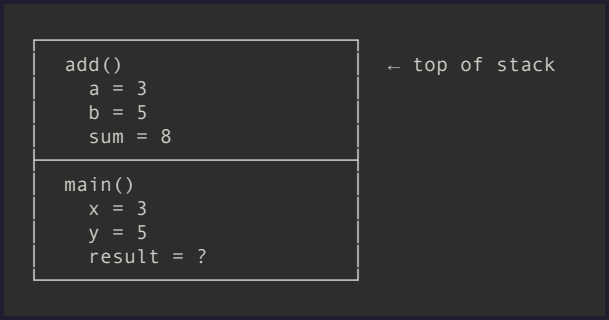
Consider this code:

```
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:

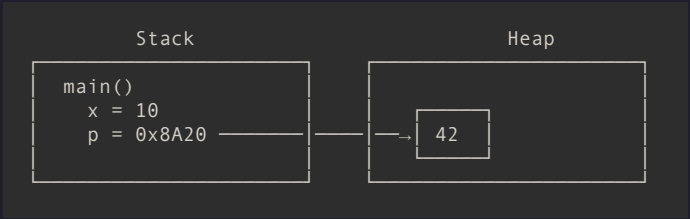


After `add` returns, its frame is gone:

▒▒ 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 tracks daily temperatures using a `std::vector` and reports statistics:

1. Ask the user to enter temperatures one at a time, entering `-999` to stop
2. Store the temperatures in a `std::vector<double>`
3. Write a function `double compute_average(const std::vector<double>& temps)` that returns the average temperature
4. Write a function `double find_max(const std::vector<double>& temps)` that returns the highest temperature
5. Write a function `double find_min(const std::vector<double>& temps)` that returns the lowest temperature
6. Write a function `int count_above(const std::vector<double>& temps, double threshold)` that returns how many temperatures are above the threshold
7. Print the number of days, average, max, min, and how many days were above the average

Expected output:

```
Enter temperatures (-999 to stop):
72.5
85.0
68.3
91.2
77.8
-999

Days recorded: 5
Average: 78.96
High: 91.2
Low: 68.3
Days above average: 2
```