

Vectors

Introduction and vectors as function parameters in C++

Professors de Pro1

Department of Computer Science

Contents

- 1 What is a Vector?
- 2 Declaring and Using Vectors
- 3 Example: Birth Analysis
- 4 Strings as Char Vectors
- 5 Vectors as Function Parameters
- 6 Example: Birth Analysis Revisited
- 7 Proposed Exercises

What is a vector?

Definition

A vector is a **container** that stores a **list of values** of the **same type**, in order.

You can add or remove elements and access any element by its **position** (*index*), starting from **0**.

Example: a vector of 6 integers

values →	7	-2	15	0	4	9
indexes →	0	1	2	3	4	5

It is the **first data structure** of the Programming I course.

Vector types

A vector is declared with the following syntax:

```
vector<type> name(size);
```

Vector	C++ type	Every item is. . .
integer vector	vector<int>	an integer
real vector	vector<double>	a real number
boolean vector	vector<bool>	a boolean
char vector	vector<char>	a character
string vector	vector<string>	a string
...	...	...

Strengths and weaknesses

Pros

- **Direct access:** any item can be reached immediately.
- Each item can be handled as if it were a **single variable**.

Cons

- Vectors live in the computer's internal memory, which is **finite**: this imposes a **physical limit** on their size.
- A vector can change its size dynamically, but in this course we will usually provide its **size** when we create it.

Vector construction

Step 1: include the library

At the beginning of the file, with the other inclusion directives:

```
#include <vector>
```

Step 2: declare and construct the vector

To create a vector of size n (a positive int):

```
vector<int>    v(n);    // int vector v of size n  
vector<bool>   w(n);    // bool vector w of size n
```

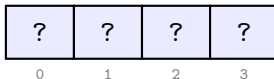
`vector<type> name(size);`

Accessing items: indexes

The line

```
vector<int> v(4);
```

creates a vector `v` that stores **four integers**, referenced by `v[0]`, `v[1]`, `v[2]` and `v[3]`.

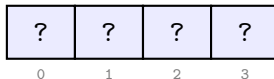


Accessing items: indexes

The line

```
vector<int> v(4);
```

creates a vector `v` that stores **four integers**, referenced by `v[0]`, `v[1]`, `v[2]` and `v[3]`.



To assign value `z` at position (*index*) `i`:

```
v[0] = 8;    // "vector v gets value 8 at index 0"
v[i] = z;    // ERROR if i is not a valid index (here: 0, 1, 2, 3)
```

Important

Indexes start at **0**, not at 1.

Exercise: trace the values of v

```
1  vector<int> v(4);  
2  v[0] = 1;  
3  v[3] = 5*v[0] - 2;  
4  v[2] = 5;  
5  v[1] = (v[2] - 1)/v[0];  
6  v[2] = 0;
```

Exercise: trace the values of v

```
1 vector<int> v(4);  
2 v[0] = 1;  
3 v[3] = 5*v[0] - 2;  
4 v[2] = 5;  
5 v[1] = (v[2] - 1)/v[0];  
6 v[2] = 0;
```

After line	v[0]	v[1]	v[2]	v[3]
2	1	?	?	?
3	1	?	?	3
4	1	?	5	3
5	1	4	5	3
6	1	4	0	3

Exercise: trace the values of v

```
1 vector<int> v(4);  
2 v[0] = 1;  
3 v[3] = 5*v[0] - 2;  
4 v[2] = 5;  
5 v[1] = (v[2] - 1)/v[0];  
6 v[2] = 0;
```

After line	v[0]	v[1]	v[2]	v[3]
2	1	?	?	?
3	1	?	?	3
4	1	?	5	3
5	1	4	5	3
6	1	4	0	3

Final state

1	4	0	3
0	1	2	3

The size of a vector: `size()`

For a vector `u`, the expression `u.size()` gives its size.

Valid indexes

If `n` is the value of `u.size()`, the valid indexes are

$$0, 1, 2, \dots, n - 1$$

In the previous example, `v.size()` is 4 and the valid indexes are 0, 1, 2, 3.

The size of a vector: `size()`

For a vector `u`, the expression `u.size()` gives its size.

Valid indexes

If `n` is the value of `u.size()`, the valid indexes are

$$0, 1, 2, \dots, n - 1$$

In the previous example, `v.size()` is 4 and the valid indexes are 0, 1, 2, 3.

Warning! `u.size()` is **not** an `int`

It is an **unsigned int** (it cannot be negative), so it may produce unexpected results. Try `cout << u.size() - w.size();` when `w` is bigger!

We strongly suggest **turning the size into an integer** first:

```
int n = u.size();
```

Birth analysis: a single year

Problem

The input channel contains the birth years of a population, all of them in the range **1991–2020**. How many people were born in year **2000**?

Example

Input: 1991 2000 1999 2001 2000 2000 **Output:** 3

Birth analysis: a single year

Problem

The input channel contains the birth years of a population, all of them in the range **1991–2020**. How many people were born in year **2000**?

Example

Input: 1991 2000 1999 2001 2000 2000 **Output:** 3

This one is easy, **no vector needed**:

```
1 //pre:  birth years in the range [1991, 2020] on the input channel
2 //post: counter2000 is the frequency of year 2000 in the input
3 const int YEAR = 2000;    // constant declaration
4 int counter2000 = 0;
5 int year;
6 while (cin >> year)
7     if (year == YEAR) ++counter2000;
```

Birth analysis: every year

New problem

What if we want the birth frequency of **every** year from 1991 to 2020?

Naive approach

One variable per year: `counter1991`, `counter1992`, ..., `counter2020`. That is 30 variables and 30 cases (`if`) in the code! ×

Birth analysis: every year

New problem

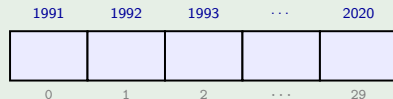
What if we want the birth frequency of **every** year from 1991 to 2020?

Naive approach

One variable per year: counter1991, counter1992, ..., counter2020. That is 30 variables and 30 cases (if) in the code! ×

Solution: a vector counters

Each **index** represents a **year**:



Size: $2020 - 1991 + 1 = 30$. Year $y \longrightarrow \text{index } y - 1991$.

Birth analysis: counting with a vector

```
1 //pre:  birth years in the range [1991, 2020] on the input channel
2 //post: counters stores the birth frequency of each year
3 const int MIN = 1991;
4 const int MAX = 2020;
5 vector<int> counters(MAX - MIN + 1);
6 for (int i = 0; i < MAX - MIN + 1; ++i) counters[i] = 0;
7 int year;
8 while (cin >> year) ++counters[year - MIN];
```

Birth analysis: counting with a vector

```
1 //pre:  birth years in the range [1991, 2020] on the input channel
2 //post: counters stores the birth frequency of each year
3 const int MIN = 1991;
4 const int MAX = 2020;
5 vector<int> counters(MAX - MIN + 1);
6 for (int i = 0; i < MAX - MIN + 1; ++i) counters[i] = 0;
7 int year;
8 while (cin >> year) ++counters[year - MIN];
```

Writing the results to the output channel:

```
1 for (int i = 0; i < MAX - MIN + 1; ++i)
2     cout << "Number of births in year " << MIN + i
3         << " is " << counters[i] << endl;
```

Birth analysis: full program

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 //in:  birth years of a population in [1991, 2020] on the input channel
6 //out: writes the birth frequency of each year
7 int main() {
8     const int MIN = 1991;
9     const int MAX = 2020;
10    vector<int> counters(MAX - MIN + 1);
11    for (int i = 0; i < MAX - MIN + 1; ++i) counters[i] = 0;
12    int year;
13    while (cin >> year) ++counters[year - MIN];
14    for (int i = 0; i < MAX - MIN + 1; ++i)
15        cout << "Number of births in year " << MIN + i
16             << " is " << counters[i] << endl;
17 }
```

Birth analysis: a simpler version

```
1 int main() {  
2     const int MIN = 1991;  
3     const int MAX = 2020;  
4     vector<int> counters(MAX - MIN + 1, 0);    // all items start at 0  
5     int year;  
6     while (cin >> year) ++counters[year - MIN];  
7     int n = counters.size();  
8     for (int i = 0; i < n; ++i)  
9         cout << "Number of births in year " << MIN + i  
10            << " is " << counters[i] << endl;  
11 }
```

Two simplifications

- 1 vector<int> counters(MAX - MIN + 1, 0); gives the initial value 0 to **all** items: no loop needed.
- 2 The second loop uses the **size of the vector** (counters.size()) instead of computing MAX - MIN + 1 again.

Using strings as char vectors

If we include the **string** library, we can use a `string` as a vector of characters (`vector<char>`):

```
#include <string>
```

Size of a string `s`

`s.length()` (but, as with any vector, `s.size()` also works)

Example: `string s = "hello";`

h	e	l	l	o
0	1	2	3	4

`s[0]` is 'h', `s[4]` is 'o', `s.length()` is 5

Example: number of different letters in a word

Problem

Given a word on the input channel, compute how many **different letters** it has.

Examples

banana	→	3	(b, a, n)
programming	→	8	

Example: number of different letters in a word

Problem

Given a word on the input channel, compute how many **different letters** it has.

Examples

banana	→	3	(b, a, n)
programming	→	8	

Idea: the **new letter test**

Compare each character $s[i]$ with the **previous ones** $s[0], \dots, s[i-1]$. If it does not appear before, it is a new letter.

This requires using the string as a **vector of char**.

Different letters: code

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 //in:  a word on the input channel
6 //out: the number of different letters in the input word
7 int main() {
8     string s;
9     cin >> s;
10    int n = s.length();    // int n = s.size() also works
11    int count = 0;
12    for (int i = 0; i < n; ++i) {
13        // **new letter test** for the char at position i
14        int j = 0;
15        while (s[j] != s[i]) ++j;
16        // if s[i] was not found at any previous position, it is new
17        if (j == i) ++count;
18    }
19    cout << count << endl;
20 }
```

Different letters: trace for "banana"

b	a	n	a	n	a
0	1	2	3	4	5

i	s[i]	j at the end	New letter?
0	b	0	yes ($j = i$)
1	a	1	yes ($j = i$)
2	n	2	yes ($j = i$)
3	a	1	no
4	n	2	no
5	a	1	no

Why does the while loop always stop?

In the worst case it stops at $j == i$, because $s[i] == s[i]$.

Reminder: value vs. reference parameters

Pass by value

A **copy** of the argument is made at the call point. The function works on the copy.

Pass by reference

No copy is made. The argument acts as an **alias** of the variable in the call, so the function works directly on the original data.

Why does it matter for vectors?

Vectors can represent **very large** data sets. Passing a vector by value means a considerable **memory expense** to store the copy.

This is only acceptable when the expense is **unavoidable**, e.g., when we want to **keep the original** vector and create another one as a function of the original.

Three ways to pass a vector

Role of the vector	Passed by	Syntax	Frequency
Input	constant reference	<code>const vector<T>& v</code>	very frequent
Output / Input-Output	reference	<code>vector<T>& v</code>	frequent
Input, copy needed	value	<code>vector<T> v</code>	infrequent

Rule of thumb

By default, use `const vector<T>&` for input vectors. Use a value parameter only if you really need a copy.

Case 1: input vector $\rightarrow$ constant reference

Constant reference

A usual reference parameter (no copy) with a **notice** that the function is **not going to change** the vector.

It is a **very frequent** case. Examples:

```
1 //pre:  ---
2 //post: returns the sum of the values of v
3 int sum(const vector<int>& v)
4
5 //pre:  u and v have the same size
6 //post: returns the scalar product of u and v
7 double scalar_product(const vector<double>& u, const vector<double>& v)
```

Case 1: implementing sum

```
1 //pre:  ---
2 //post: returns the sum of the values of v
3 int sum(const vector<int>& v) {
4     int s = 0;
5     int n = v.size();
6     for (int i = 0; i < n; ++i) s += v[i];
7     return s;
8 }
```

Case 1: implementing sum

```
1 //pre:  ---
2 //post: returns the sum of the values of v
3 int sum(const vector<int>& v) {
4     int s = 0;
5     int n = v.size();
6     for (int i = 0; i < n; ++i) s += v[i];
7     return s;
8 }
```

The compiler protects us

Any attempt to modify `v` inside `sum`, such as `v[0] = 0;`, produces a **compilation error**.

Case 2: output or input/output vector → reference

The function has to **modify** the vector, so we pass it by **reference**. It is a **frequent** case.

```
1 //pre:  there are at least v.size() words on the input channel
2 //post: vector v stores the first v.size() words from cin
3 void read_vector(vector<string>& v) {
4     int n = v.size();
5     for (int i = 0; i < n; ++i) cin >> v[i];
6 }
7
8 //pre:  ---
9 //post: each value of v has been replaced by its square
10 void square(vector<int>& v) {
11     int n = v.size();
12     for (int i = 0; i < n; ++i) v[i] = v[i]*v[i];
13 }
```

Case 3: input vector, copy unavoidable $\rightarrow$ value

Problem

A list of positive integers is stored in a vector v . We want to **keep** v and compute a new vector w such that $w[i]$ is the **first prime number** $\geq v[i]$.

Example

v:	0	1	2	3	4	5
	1	5	4	9	15	10
w:	0	1	2	3	4	5
	2	5	5	11	17	11

It is an **infrequent** case: here the copy is exactly what we need.

Case 3: first_primes

```
1 //pre:  all values of v are positive
2 //post: returns a vector whose value at each position i is the first
3 //      prime number greater than or equal to the original v[i]
4 vector<int> first_primes(vector<int> v) {
5     int n = v.size();
6     for (int i = 0; i < n; ++i)
7         while (not is_prime(v[i])) ++v[i];
8     return v;
9 }
```

Case 3: first_primes

```
1 //pre:  all values of v are positive
2 //post: returns a vector whose value at each position i is the first
3 //      prime number greater than or equal to the original v[i]
4 vector<int> first_primes(vector<int> v) {
5     int n = v.size();
6     for (int i = 0; i < n; ++i)
7         while (not is_prime(v[i])) ++v[i];
8     return v;
9 }
```

After the call `vector<int> w = first_primes(v);` we get the vector of primes `w`, and `v` **has not changed**: the function worked on a copy.

Note: formally we return a **single** object, but since it is a vector, functions can now **return several values** (all of the same type).

Warning!

This very same code would **not compile** if `v` were passed by constant reference (`++v[i]` modifies it).

Birth analysis revisited: main program

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 // ... MIN, MAX, counter_vector_gen and write_vector (next slide) ...
6
7 //in:  birth years of a population in [1991, 2020] on the input channel
8 //out: writes the birth frequency of each year
9 int main() {
10     vector<int> counters(MAX - MIN + 1);
11     counter_vector_gen(counters);
12     write_vector(counters);
13 }
```

Notice

- counter_vector_gen **fills** the vector → reference.
- write_vector only **reads** it → constant reference.
- The main function is now short and easy to read.
- **Top-down design**: first main, then the functions it uses.

Birth analysis revisited: functions

```
1  const int MIN = 1991;
2  const int MAX = 2020;
3
4  //pre:  counters has size MAX - MIN + 1. All birth years in the input
5  //      sequence are in the range [MIN, MAX]
6  //post: counters[i] is the frequency of births in year MIN + i
7  void counter_vector_gen(vector<int>& counters) {           // OUTPUT
8      int n = counters.size();
9      for (int i = 0; i < n; ++i) counters[i] = 0;
10     int year;
11     while (cin >> year) ++counters[year - MIN];
12 }
13
14 //pre:  ---
15 //post: writes the frequency vector counters on the output channel
16 void write_vector(const vector<int>& counters) {           // INPUT
17     int n = counters.size();
18     for (int i = 0; i < n; ++i)
19         cout << "Number of births in year " << MIN + i
20              << " is " << counters[i] << endl;
21 }
```

Proposed exercises

Write the following functions, choosing carefully how to pass each vector:

- ❶ `scalar_product(u, v)`: returns $\sum_i u_i \cdot v_i$.
`u = [1, 2, 3], v = [4, 5, 6] -> 32`
- ❷ `maximum(v)`: returns the largest value of a non-empty vector.
`[3, 9, 2, 7] -> 9`
- ❸ `reverse(v)`: reverses the order of the values of `v`.
`[1, 2, 3, 4] -> [4, 3, 2, 1]`
- ❹ `is_palindrome(s)`: checks whether a string reads the same backwards.
`"radar" -> true`
- ❺ `letter_frequency(s)`: returns a vector of size 26 with the number of occurrences of each lowercase letter in `s`.
- ❻ Solve the **different letters** problem again using a vector of 26 counters. Map each letter 'a'..'z' to an index 0..25 with `c - 'a'`.

Vectors on one slide

- A vector stores a **homogeneous** data set with **direct access**: `vector<int> v(n);`
- Valid indexes: `0, ..., n - 1`. Use `int n = v.size();`
- A string can be used as a vector of `char`.
- Parameter passing:

Input	<code>const vector<T>&</code>	(very frequent)
Output / In-Out	<code>vector<T>&</code>	(frequent)
Input + copy	<code>vector<T></code>	(infrequent)

Never pass a vector by value unless you really need a copy!

Questions?

Vectors: Introduction and vectors as function parameters in C++