Data Structures and Abstract Data Type (ADT) Review

<gerdela@scss.tcd.ie>

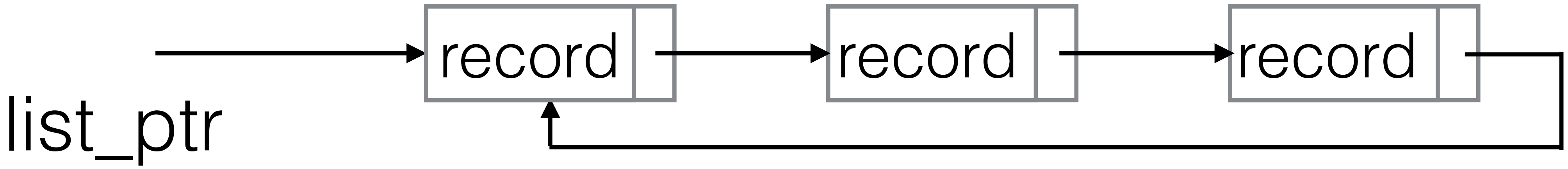
So far we know

- Linked Lists
  - diagrams
  - add to front/end, insert before/after, delete front/end/current, search, reverse, concatenate, split.
- ~ Trees and ~~Graphs (more in next topics)
- Arrays and 2d arrays (aka *matrix*)

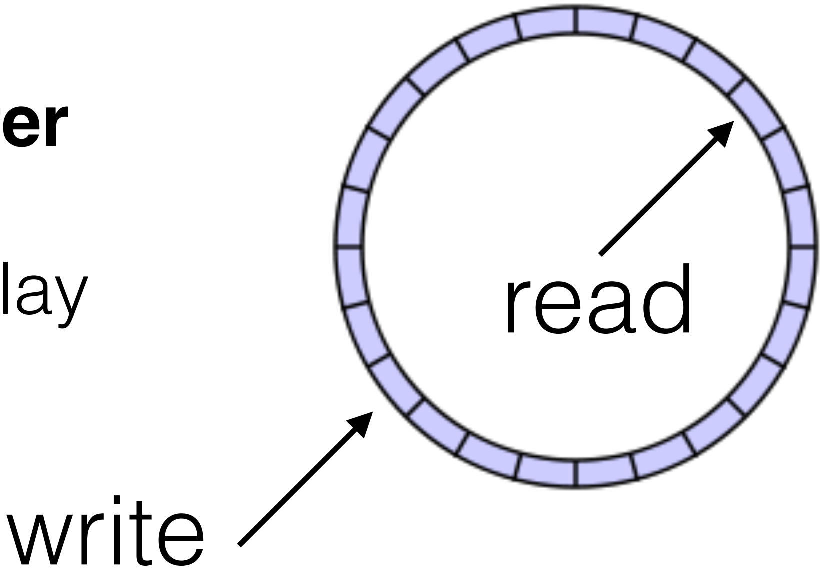
  ```
  int x[10];
  int table[50][100];
           rows cols
  ```
- Hash Tables

Circular List

- Instead of finishing list with NULL pointer…
- Be careful to stop at some point when searching

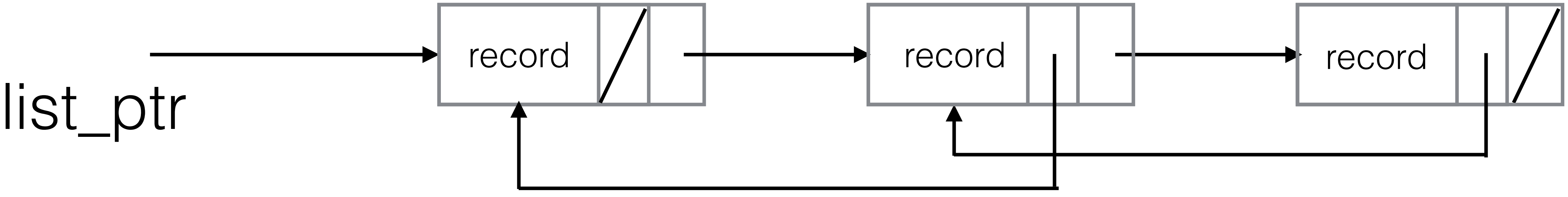


- this is one way to implement a **circular buffer**
  - useful for streaming. e.g. "continuously play audio" whilst downloading

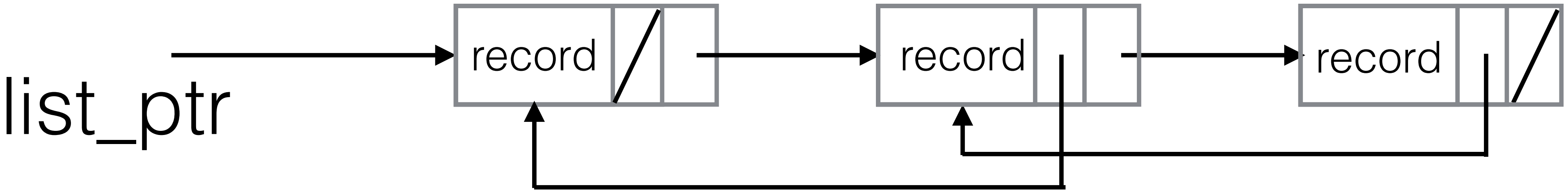


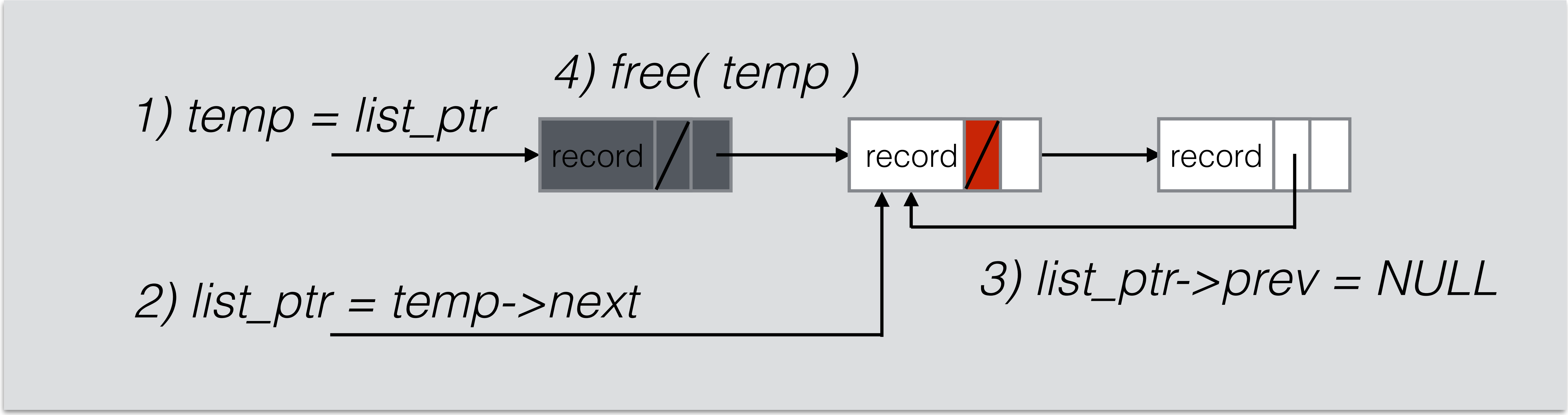
Doubly-Linked List

- next and previous pointers
- inserting and deleting is more complex



Doubly-linked List
Deleting head node

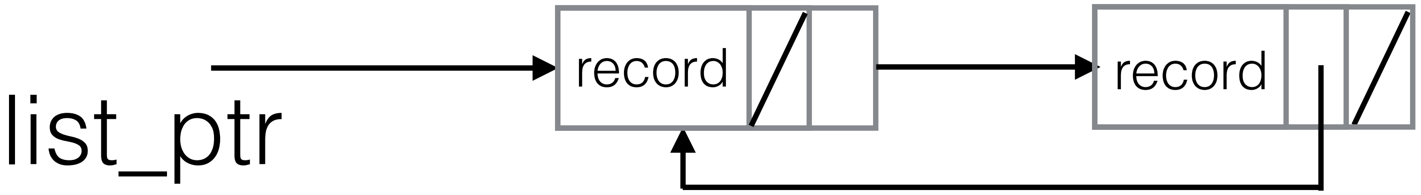



1) temp = list_ptr

2) list_ptr = temp->next

3) list_ptr->prev = NULL

4) free(temp)



Matrix (2d array)

- create a big 2d table
- stored in 1d in memory of course
  - shall we program a thing to find out what order?
- int table[rows][cols];
- a **sparse** matrix - most elements are 0 (unused)
  - memory inefficient. could do…
  - problems with this?

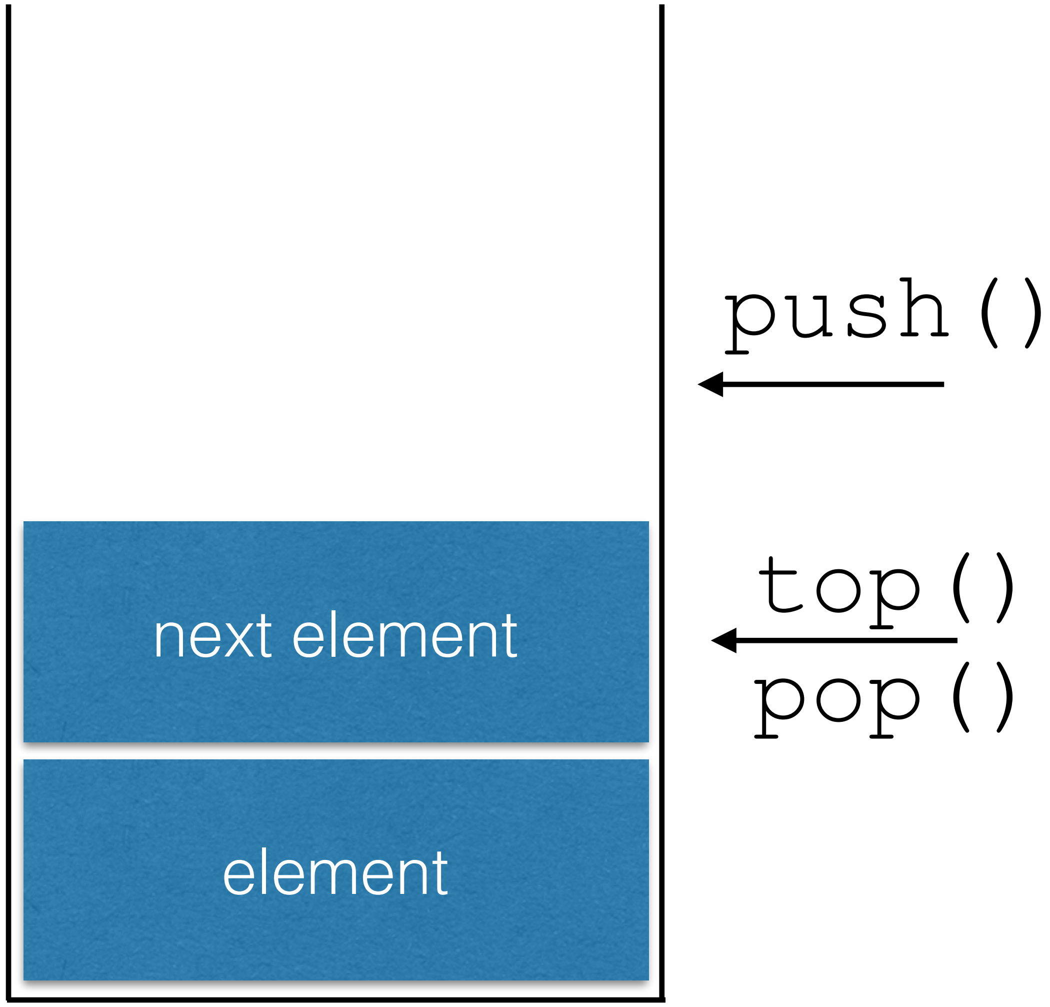
```
struct Sparse {
  int row, col;
  int data;
};

Sparse a[128];
int sparse_count = 0;

a[0].row = 10;
a[0].col = 0;
a[0].data = 222;
sparse_count++;
```

Abstract Data Types (ADT)

- *Implementation agnostic*
  - any data structure/algorithm underneath
  - user doesn't need/want details of how it works
  - might switch impl. behind scenes depending on context
- e.g. abstract **Stack** - has `push()`, `pop()`, `top()`
  - one big malloc(), pointer, and offset?
  - static array and counter?
  - linked list?

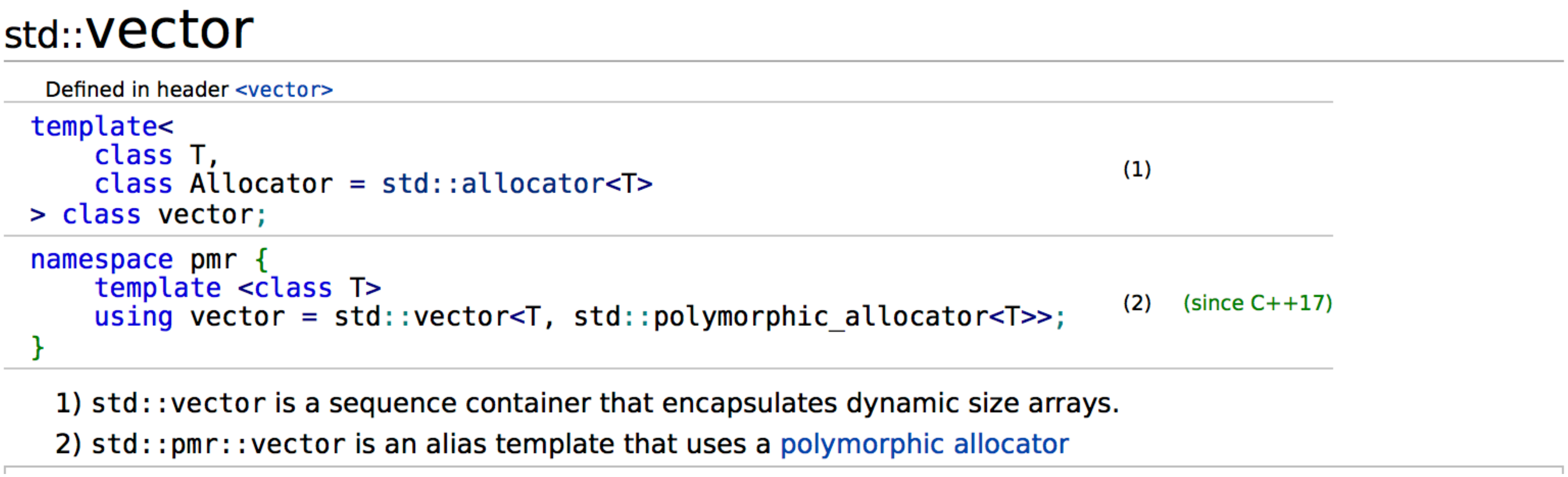


ADT and standard libraries

- **C++ STL** (standard template library) -1979
  - Alexander Stepanov
  - generic programming
    - extended by **Boost** libraries
- other common ADTs; **vector** data type, **dictionary**/ map (probably a hash table underneath).

example - std::vector

- every lazy C++ programmer's favourite!
- it's a pre-made ADT from the STL in C++
    - works like both a linked list and an array
    - store any data type or object or struct in it
    - don't have to touch any pointers directly
    - http://en.cppreference.com/w/cpp/container/vector



std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::polymorphic_allocator<T>>;
}
```

(2) (since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

Summary:

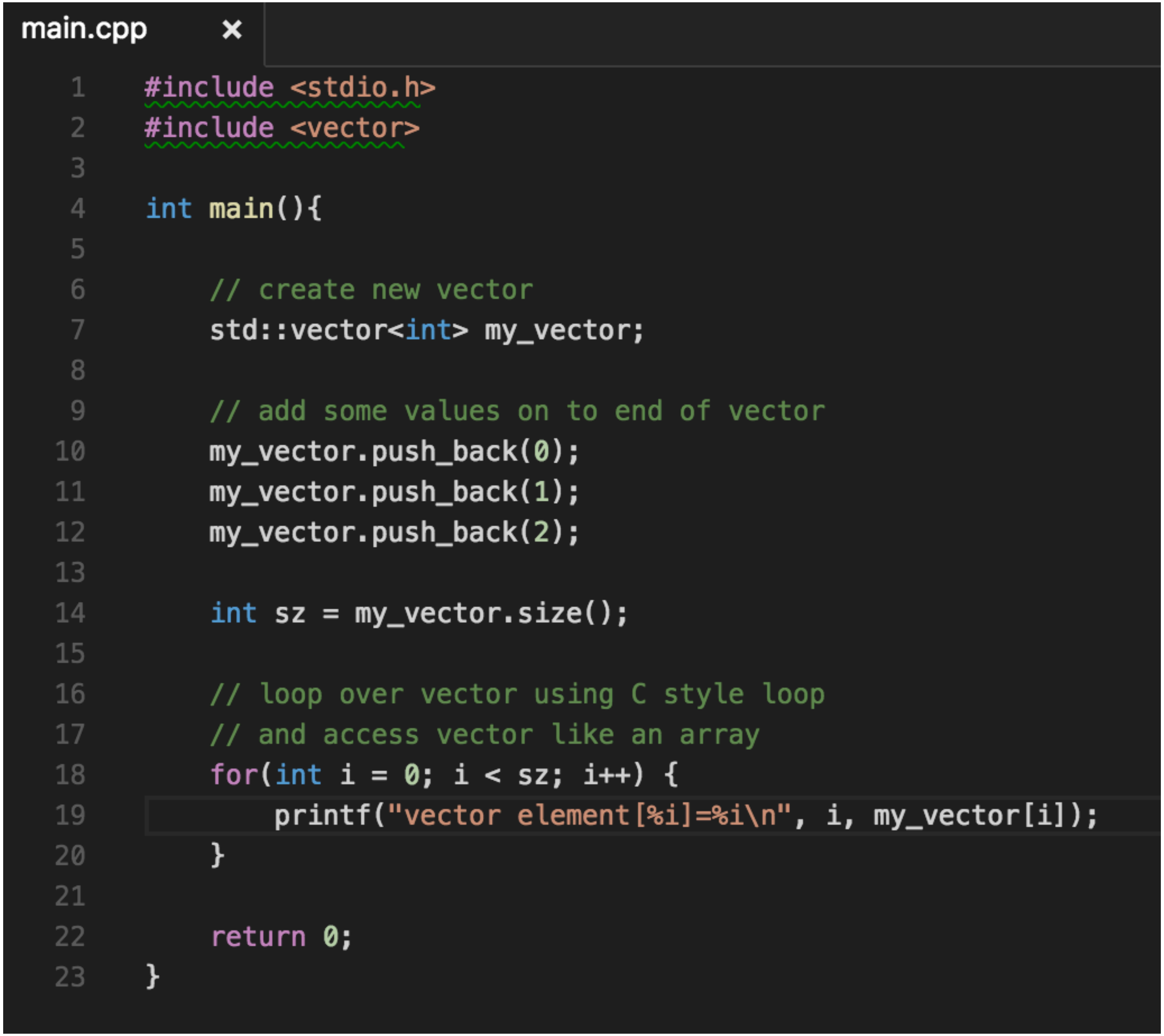
It tells us to **`#include <vector>`**

and we will use a **class** called **vector** with general form:

```
template<
  class T,
  class Allocator = std::allocator<T>
> class vector;
```

`<T>` means "specify your own data type here when you go to use vector".

live demo



```cpp
#include <stdio.h>
#include <vector>

int main(){

    // create new vector
    std::vector<int> my_vector;

    // add some values on to end of vector
    my_vector.push_back(0);
    my_vector.push_back(1);
    my_vector.push_back(2);

    int sz = my_vector.size();

    // loop over vector using C style loop
    // and access vector like an array
    for(int i = 0; i < sz; i++) {
        printf("vector element[%i]=%i\n", i, my_vector[i]);
    }

    return 0;
}
```

std::vector use

- the vector class is under the std namespace
- has functions
  - `push_back()  empty()  size()  reserve()`
  - `front()  back()  insert()` … more
- can use the `[]` operators to access specific element

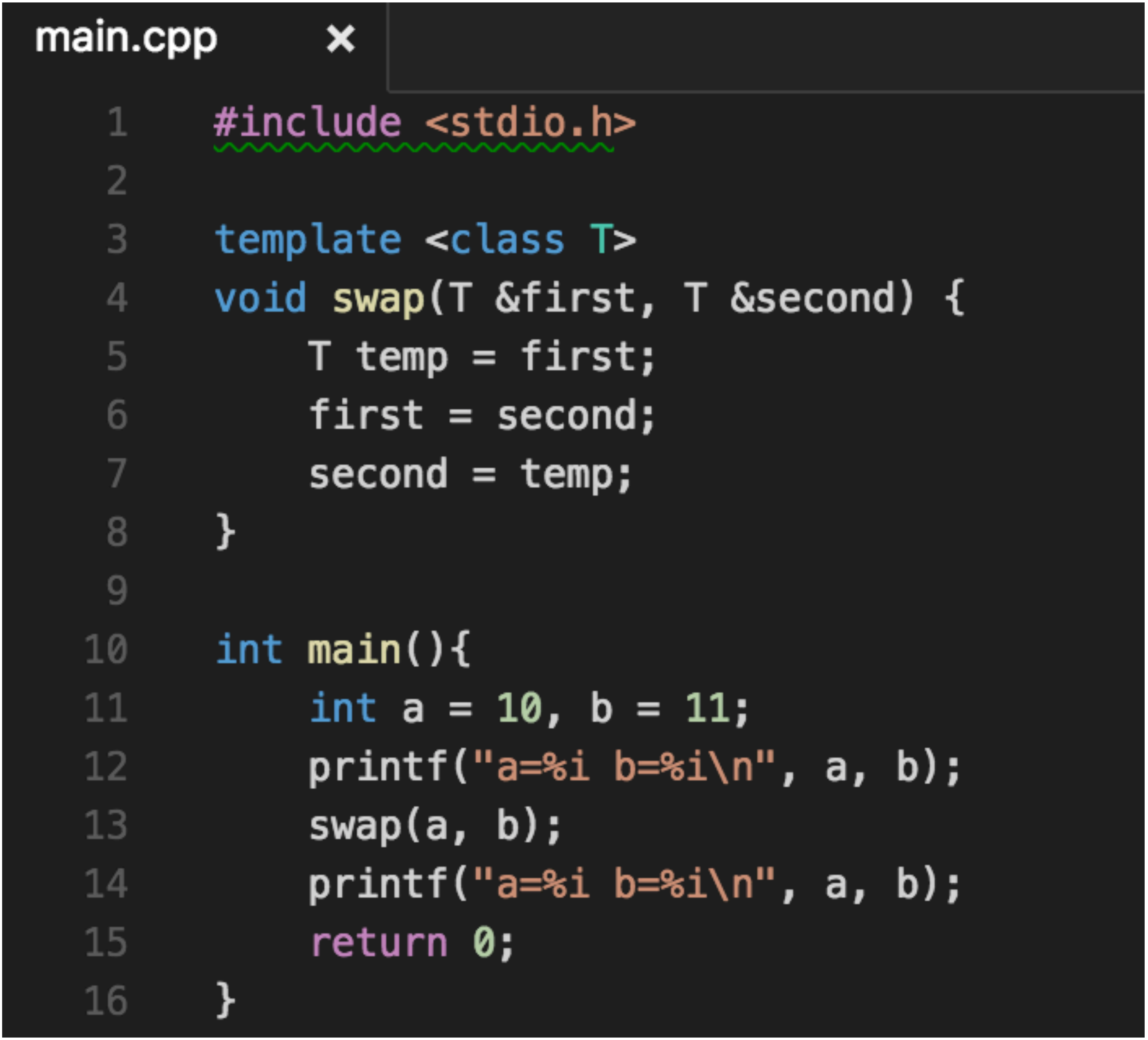
You can make your own template classes or functions

- e.g. sorting functions - work with any data type
- quite ugly/difficult to do in plain C
- if using *objects* - operators used in function must be overloaded. i.e. { **<**, **>**, **==**, **=**, … }
- put line above function declaration or definition:
  `template <class T>`
- then use `T` as an argument's data type
  `void myfunc( T myarg );`

live demo

`&first` is a C++ '**reference**'

(it doesn't let me use pointers for a template)



```cpp
#include <stdio.h>

template <class T>
void swap(T &first, T &second) {
    T temp = first;
    first = second;
    second = temp;
}

int main(){
    int a = 10, b = 11;
    printf("a=%i b=%i\n", a, b);
    swap(a, b);
    printf("a=%i b=%i\n", a, b);
    return 0;
}
```

Generic Programming

- Practical downsides can include
  - very poor performance / very slow compile time
    - inspect assembly of template/generic code
  - code hard to read/follow
  - useless compiler error messages
  - hard to step-through debug
  - give away control over memory allocation - `reserve()`