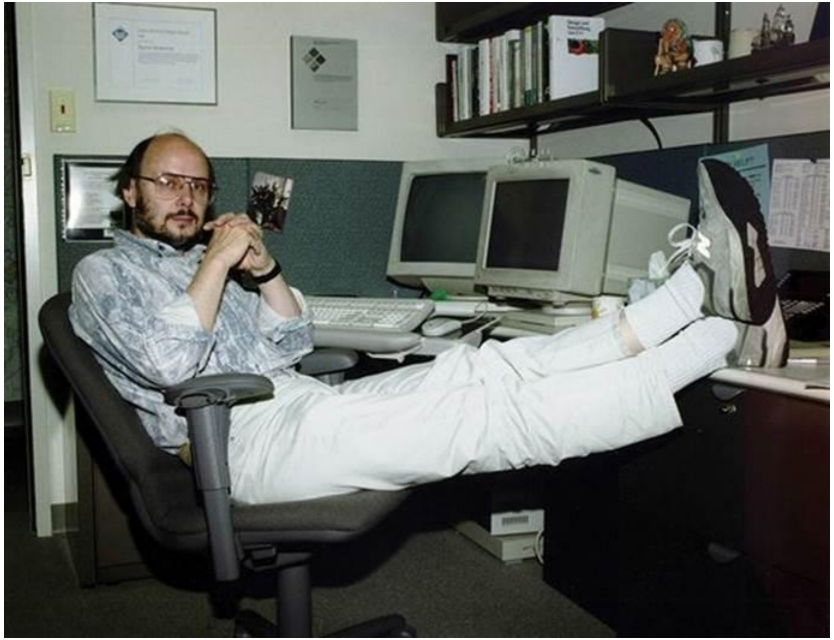
System Programming with C++

History, classes and encapsulation.

History



Bjarne Stroustrup

History

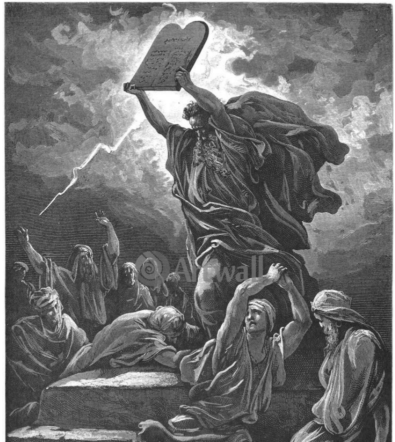
History

- 1972 - C Language creation
- 1980 - C with classes by Bjarne in Bell Labs.
  Added Simula features to C.

History

- 1972 - C Language creation
- 1980 - C with classes

- 1998 - ISO/IEC 14882:1998
  «Standard for the C++ Programming Language»
- 2003 - ISO/IEC 14882:2003



History

- 1972 - C Language creation
- 1980 - C with classes

- 1998 - ISO/IEC 14882:1998
  «Standard for the C++ Programming Language»
- 2003 - ISO/IEC 14882:2003

- 2011 - C++11 (C++0x)

History

- 1972 - C Language creation
- 1980 - C with classes

- 1998 - ISO/IEC 14882:1998
  «Standard for the C++ Programming Language»
- 2003 - ISO/IEC 14882:2003

- 2011 - C++11 (C++0x) ⇒ 2014 - C++14 ⇒ 2017 - C++17

History

- 1972 - C Language creation
- 1980 - C with classes

- 1998 - ISO/IEC 14882:1998
  «Standard for the C++ Programming Language»
- 2003 - ISO/IEC 14882:2003

- 2011 - C++11 (C++0x) ⇒ 2014 - C++14 ⇒ 2017 - C++17
- 2020 - C++20, 2023 - C++23
- 2026 - C++26

What’s wrong with C language?

What's wrong with C language?

Nothing! C is just perfect.

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

Pros?

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

Pros: fast, straightforward, low level ✓

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

Pros: fast, straightforward, low level ✓

Cons: ?

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

Pros: fast, straightforward, low level ✓

Cons: too straightforward! ✗

(hard to create abstractions)

What's wrong with C language?

Nothing! C is just perfect.

Perfect macro assembler.

Pros: fast, straightforward, low level ✓

Cons: too straightforward! ✗

(hard to create abstractions)

=> poor standard library ✗

Let's try C

Let's try C

```
Task: implement "growable array"
      data structure in C
```

Let's try C

```
Task: implement "growable array" of ints
      data structure in C
```

Let's try C

Task: implement "growable array" of ints
      data structure in C

Functionality:

1. add element to the end,
2. remove the last element,
3. get element at index (random access)

```
struct Vector {




};
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    ...
}
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) {
        v->capacity = (v->capacity + 1) * 2;
        int* new_data = realloc(v->data, v->capacity);
        if (!new_data) { ... }
        v->data = new_data;
    }
    ...
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) {
        v->capacity = (v->capacity + 1) * 2;
        int* new_data = realloc(v->data, v->capacity);
        if (!new_data) { ... }
        v->data = new_data;
    }
    ...
```

Why pointer?

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};
```

Why pointer?

```
void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) {
        v->capacity = (v->capacity + 1) * 2;
        int* new_data = realloc(v->data, v->capacity);
        if (!new_data) { ... }
        v->data = new_data;
    }
    ...
```

Otherwise, Vector would be copied. Definitely not the thing we want.

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) {
    return v->data[--(v->size)];
}
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {
    .data = NULL,
    .size = 0,
    .capacity = 0
};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
```

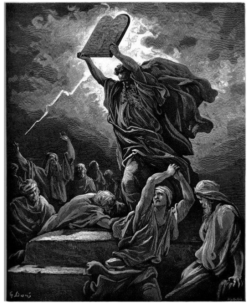
Something else?

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
```

Something else?

Freeing memory!



```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
free(v.data);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
free(v.data);
```

What would you improve (in C)?

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

int pop_back(struct Vector* v) { ... }

int get(struct Vector* v, size_t pos) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
free(v.data);
```

What would you improve (in C)?

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v = {0};
push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
free(v.data);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

Now it is quite good for C lang!

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

But what's wrong with it in general?

```
Task: implement "growable array" of ints
      data structure in C
```

```
Problems:
```

Task: implement "growable array" of ints
data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

   Hard to think about the logic, hard to read, tons of boilerplate code.

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

But what's wrong with it in general?

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

Boilerplate!

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user. Feel free to change!

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

Why should user care???

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

Who stops user from this?

```c
struct Vector v;
init(&v);

push_back(&v, 13);
v.capacity = 0; // lol

push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user.

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user.

3. Inconsistent state of an object.

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

```c
struct Vector v;
init(&v);

push_back(&v, 13);
push_back(&v, 42);

int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

What will happen?

```c
struct Vector v;
~~init(&v);~~

push_back(&v, 13);
push_back(&v, 42);

int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

What will happen?

Object not initialized => garbage in fields => UB

```c
struct Vector v;
~~init(&v);~~

push_back(&v, 13);
push_back(&v, 42);

int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

Task: implement "growable array" of ints
data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user.

3. Inconsistent state of an object.

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user.

3. Inconsistent state of an object.

4. Problems with generalization (macroses).

C++ to rescue

C++ to rescue

Task: implement "growable array" of ints
data structure in C++

Functionality:

1. add element to the end,
2. remove the last element,
3. get element at index (random access)

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

struct Vector v;
...
push_back(&v, 13);
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}

struct Vector v;
...
push_back(&v, 13);
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this is a pointer to the instance from which the function was called

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this is a pointer to the instance from which the function was called

How to implement?

https://godbolt.org/z/e4x6x6W6v

C++

```cpp
#include <stdlib.h>

int k;

struct Vector {
    int* data;
    std::size_t size;
    std::size_t capacity;
};

__attribute__((noinline)) int pop_back(struct Vector* v) {
    return v->data[--(v->size)];
}

int main() {
    Vector v;
    k = pop_back(&v);
    return 0;
}
```

x86-64 gcc 13.2 -O2

```asm
pop_back(Vector*):
        mov     rax, QWORD PTR [rdi+8]
        mov     rdx, QWORD PTR [rdi]
        sub     rax, 1
        mov     QWORD PTR [rdi+8], rax
        mov     eax, DWORD PTR [rdx+rax*4]
        ret
main:
        sub     rsp, 40
        mov     rdi, rsp
        call    pop_back(Vector*)
        mov     DWORD PTR k[rip], eax
        xor     eax, eax
        add     rsp, 40
        ret
k:
        .zero   4
```

https://godbolt.org/z/sP1973nxf

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```cpp
#include <stdlib.h>

int k;

struct Vector {
    int* data;
    std::size_t size;
    std::size_t capacity;

    __attribute__((noinline)) int pop_back() {
        return this->data[--(this->size)];
    }

};


int main() {
    Vector v;
    k = v.pop_back();
    return 0;
}
```

x86-64 gcc 13.2 -O2

A Output... Filter... Libraries Overrides + Add new... Add tool...

```asm
Vector::pop_back():
        mov     rax, QWORD PTR [rdi+8]
        mov     rdx, QWORD PTR [rdi]
        sub     rax, 1
        mov     QWORD PTR [rdi+8], rax
        mov     eax, DWORD PTR [rdx+rax*4]
        ret
main:
        sub     rsp, 40
        mov     rdi, rsp
        call    Vector::pop_back()
        mov     DWORD PTR k[rip], eax
        xor     eax, eax
        add     rsp, 40
        ret
k:
        .zero   4
```

https://godbolt.org/z/sP1973nxf

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```cpp
#include <stdlib.h>

int k;

struct Vector {
    int* data;
    std::size_t size;
    std::size_t capacity;

    __attribute__((noinline)) int pop_back() {
        return this->data[--(this->size)];
    }

};


int main() {
    Vector v;
    k = v.pop_back();
    return 0;
}
```

x86-64 gcc 13.2 -O2

A Output... Filter... Libraries Overrides + Add new... Add tool...

```asm
Vector::pop_back():
        mov     rax, QWORD PTR [rdi+8]
        mov     rdx, QWORD PTR [rdi]
        sub     rax, 1
        mov     QWORD PTR [rdi+8], rax
        mov     eax, DWORD PTR [rdx+rax*4]
        ret
main:
        sub     rsp, 40
        mov     rdi, rsp
        call    Vector::pop_back()
        mov     DWORD PTR k[rip], eax
        xor     eax, eax
        add     rsp, 40
        ret
k:
        .zero   4
```

this is just an invisible first argument of any inner function, code generation doesn't suffer => zero-cost abstraction!

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (this->size == this->capacity) { ... }
        this->data[this->size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this is a pointer to the instance from which the function was called

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value) {
        if (size == capacity) { ... }
        data[size++] = value;
    }
};

Vector v;
...
v.push_back(13);
```

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this is a pointer to the instance from which the function was called
- avoid explicit usage of this where possible

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value);
};

void Vector::push_back(int value) {
    if (size == capacity) { ... }
        data[size++] = value;
    }
}
```

No need to place everything inside code of the struct, only declaration.

- move function to the structure itself
- now structure is not only several fields, but also logic, how to work with it!
- this is a pointer to the instance from which the function was called
- avoid explicit use of this where possible

Task: implement "growable array" of ints
data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct.

2. Implementation details are accessible to the user.

3. Inconsistent state of an object.

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct. ✓

2. Implementation details are accessible to the user.

3. Inconsistent state of an object.

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

Who stops user from this?

```c
struct Vector v;
init(&v);

push_back(&v, 13);
v.capacity = 0; // lol

push_back(&v, 42);
int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```
struct Vector {
    int* data;
    size_t size;
    size_t capacity;

    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

```cpp
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:
    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

```
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:
    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

internal part of structure, can be accessed only from functions of the structure

```
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:
    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

internal part of structure, can be accessed only from functions of the structure

public API, can be used anywhere

```cpp
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:
    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

Who stops user from this?

```cpp
Vector v;

v.push_back(13);
v.capacity = 0; // lol

v.push_back(42);
int k = v.pop_back();
int p = v.get(0);
...
v.dispose();
```

```cpp
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:
    void push_back(int value);
    int pop_back();
    void init();
    void dispose();
};
```

Who stops user from this?

Compiler!

```cpp
Vector v;

v.push_back(13);
v.capacity = 0; // lol

v.push_back(42);
int k = v.pop_back();
int p = v.get(0);
```

```
<source>:21:7: error: 'std::size_t Vector::capacity' is private within this context
   21 |     v.capacity = 0;
      |       ^~~~~~~~
<source>:9:17: note: declared private here
    9 |     std::size_t capacity;
      |                 ^~~~~~~~
Compiler returned: 1
```

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct. ✓

2. Implementation details are accessible to the user. ✓

3. Inconsistent state of an object.

Access modifiers

1. `private` defines internal code (methods and fields) of a structure

Access modifiers

1. `private` defines internal code (methods and fields) of a structure: you can access it only from code of your functions

```
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:

    size_t total_size(Vector* another) {
        return size + another->size;
    }
};
```

```
struct Vector {
private:
    int* data;
    size_t size;
    size_t capacity;

public:

    size_t total_size(Vector* another) {
        return this->size + another->size;
    }
};
```

We have access not only to private fields/methods of this, but of any other Vector!

Access modifiers

1. private defines internal code (methods and fields) of a structure: you can access it only from code of your functions. Add getters/setters if you need to access private.

```cpp
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:

    size_t size() const {
        return size_;
    }
};
```

```cpp
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:

    size_t size() const {
        return size_;
    }
};
```

getters for (read only!) access to private fields

```cpp
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:

    size_t size() const {
        return size_;
    }
};
```

naming of private fields depends on your code style, Google CC suggests underscore at the end

getters for (read only!) access to private fields

```
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:

    size_t size() const {
        return size_;
    }
};
```

naming of private fields depends on your code style, Google CC suggests underscore at the end

getters for (read only!) access to private fields

const will be discussed later

Access modifiers

1. `private` defines internal code (methods and fields) of a structure: you can access it only from code of your functions

2. Why to have `private` fields and methods?

Access modifiers

1. `private` defines internal code (methods and fields) of a structure: you can access it only from code of your functions

2. Why to have `private` fields and methods? To prevent breaking the invariants (by other developers who will use your code).

```
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

What invariants here?

```cpp
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

What invariants here?

- `size_ <= capacity_`
- `data_` allocated (or `nullptr` when `size_ == 0`)

```cpp
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

What invariants here?

- `size_ <= capacity_`
- `data_` allocated (or `nullptr` when `size_ == 0`)

That's why user shouldn't modify those fields by himself!

That's why fields are private.

Access modifiers

1. private defines internal code (methods and fields) of a structure: you can access it only from code of your functions

2. Why to have private fields and methods? To prevent breaking the invariants.

3. Default access modifier for structures is public. Why?

Access modifiers

1. `private` defines internal code (methods and fields) of a structure: you can access it only from code of your functions

2. Why to have `private` fields and methods? To prevent breaking the invariants.

3. Default access modifier for structures is `public`. Why? Backward compatibility with C!

```
struct Vector {
private:
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

```
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

```cpp
class Vector {

    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }

    size_t size() const { return size_; }
};
```

```
struct - default is public,
class  - default is private.
```

Access modifiers

1. private defines internal code (methods and fields) of a structure: you can access it only from code of your functions

2. Why to have private fields and methods? To prevent breaking the invariants.

3. Default access modifier for structures is public. Default access modifier for classes is private.

Access modifiers

Q: Do we still need struct in C++? Except for backward compatibility.

Access modifiers

Q: Do we still need struct in C++? Except for backward compatibility.

A: Sure!

```
struct Point {
    int x;
    int y;
};

Point p = {3, -5};
```

Access modifiers

Q: Do we still need struct in C++? Except for backward compatibility.

A: Sure!

```
struct Point {
    int x;
    int y;
};

Point p = {3, -5};
```

struct is exactly what we need here:

- No invariants at all
- No boilerplate getters/setters
- Just a couple of ints

Access modifiers

Q: Should all fields in classes by always private?

Access modifiers

Q: Should all fields in classes by always private?

A: The invariants of your code should not be broken.
Use private only for that. Avoid cargo cults!

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct. ✓

2. Implementation details are accessible to the user. ✓

3. Inconsistent state of an object.

```c
struct Vector {
    int* data;
    size_t size;
    size_t capacity;
};

void push_back(struct Vector* v, int value) {
    if (v->size == v->capacity) { ... }
    v->data[v->size++] = value;
}
int pop_back(struct Vector* v) { ... }
int get(struct Vector* v, size_t pos) { ... }
void init(struct Vector* v) { ... }
void dispose(struct Vector* v) { ... }
```

What will happen?

Object not initialized => garbage in fields => UB

```c
struct Vector v;
~~init(&v);~~

push_back(&v, 13);
push_back(&v, 42);

int k = pop_back(&v);
int p = get(&v, 0);
...
dispose(&v);
```

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    void push_back(int value) {
        if (v->size_ == v->capacity_) { ... }
        data_[v->size_++] = value;
    }
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    ...
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    ...
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

It is called constructor and used for initialization

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    ...
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

It is called constructor and used for initialization

Has some problems, will discuss them later

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    ...
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

```cpp
Vector v;
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    ...
};
```

To make it more interesting, let preallocate buffer with some initial size when Vector is created.

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16
```

```cpp
class Vector {
    ...
public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16
```

There can be several ctors with different arguments

```cpp
class Vector {
    ...
public:
    Vector() {
        size_ = 0;
        capacity_ = 16;
        data_ = new int[capacity_];
    }
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16

Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8
```

There can be several ctors with different arguments.

You can choose different constructors for initialization.

```cpp
class Vector {
    ...
public:

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16

Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8
```

Constructors can call each other before execution of their own body.

```cpp
class Vector {
    ...
public:
                    default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector v; // calls ctor
// object v is initialized!
std::cout << v.capacity(); // 16

Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8
```

Default (without arguments) ctors are special: if you have no ctors at all, the compiler will generate you an empty default ctor.

```
class Vector {
    ...
public:
                          default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```
Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8


Vector v2d[10];
std::cout << v2d[0].capacity();
```

How elements inside a flat array will be initialized?

```cpp
class Vector {
    ...
public:
                    default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

```cpp
Vector p{8}; // calls second ctor
// object p is initialized!
std::cout << p.capacity(); // 8


Vector v2d[10];
// default ctor called 10 times
std::cout << v2d[0].capacity();
                           // 16
```

How elements inside a flat array will be initialized?

Default constructor!

```cpp
class Vector {
    ...
public:
                        default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create objects in dynamic memory.

```cpp
class Vector {
    ...
public:
                 default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create objects in dynamic memory.

New operator:

- allocates memory,

```cpp
class Vector {
    ...
public:
                    default ctor
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create objects in dynamic memory.

New operator:

- allocates memory,
- checks the result (throws a special exception on failure)

```cpp
class Vector {
    ...
public:
                          default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create objects in dynamic memory.

New operator:

- allocates memory,
- checks the result (throws a special exception on failure)
- initialize an object via calling ctor
- returns pointer to the initialized object

```cpp
class Vector {
    ...
public:                default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create objects in dynamic memory.

```cpp
Vector v{8}; // calls ctor
// object v is initialized!
std::cout << v.capcity(); // 8
```

```cpp
class Vector {
    ...
public:
                    default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create and initialize (!) objects in dynamic memory.

```cpp
Vector* pv = new Vector{8};
// ctor is called =>
// object pointed by pv
// is initialized!
std::cout << pv->capacity(); // 8
```

```
class Vector {
    ...
public:
                    default ctor

    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

New operator is C++ way to create and initialize (!) objects in dynamic memory.

```
Vector* arr = new Vector[32];
// default ctor called 32 times
std::cout << arr[0].capacity();
                            // 16
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;

public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() { ←  destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {  ←  destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

Invoked when object is about to die.

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {  ←  destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

Invoked when object is about to die.

When object is about to die?

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {  ← destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

Invoked when object is about to die.

When object is about to die?
When it’s lifetime ends!

When does it end?

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() { ← destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

Invoked when object is about to die.

When object is about to die?
When it’s lifetime ends!

When does it end?

It depends…

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {  ←  destructor
        delete[] data_;
    }
};
```

Finally, we need to somehow deallocate memory for data_ when Vector object is dead.

Here come destructors!

Invoked when object is about to die.

When object is about to die?
When it’s lifetime ends!

When does it end?

It depends… on the type of memory where object was created!

Object lifetime (first approximation, C-like)

When does object die? Depends on its storage duration:

Object lifetime (first approximation, C-like)

When does object die? Depends on its storage duration:

- static => when program terminates (return from main or call exit)

Object lifetime (first approximation, C-like)

When does object die? Depends on its storage duration:

- static => when program terminates
  (return from main or call exit)

- automatic => at the end of the scope

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```cpp
int main() {
    Vector v;
    for (int i = 0; i < 10; i++) {
        Vector p{2};
        cout << p.capacity();
        cout << "end of iteration";
    }
    cout << "end of the loop";
    cout << v.capacity();
    cout << "end of method";
    return 0;
}
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```cpp
int main() {
    Vector v;
    for (int i = 0; i < 10; i++) {
        Vector p{2};
        cout << p.capacity();
        cout << "end of iteration";
    }
    cout << "end of the loop";
    cout << v.capacity();
    cout << "end of method";
    return 0;
}
```

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```
int main() {
    Vector v;
    for (int i = 0; i < 10; i++) {
        Vector p{2};
        cout << p.capacity();
        cout << "end of iteration";
    }
    cout << "end of the loop";
    cout << v.capacity();
    cout << "end of method";
    return 0;
}
```

lifetime of p

dstr is called

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```cpp
int main() {
    Vector v;
    for (int i = 0; i < 10; i++) {
        Vector p{2};
        cout << p.capacity();
        cout << "end of iteration";
    }
    cout << "end of the loop";
    cout << v.capacity();
    cout << "end of method";
    return 0;
}
```

lifetime of v

```
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```
int main() {
    Vector v;
    for (int i = 0; i < 10; i++) {
        Vector p{2};
        cout << p.capacity();
        cout << "end of iteration";
    }
    cout << "end of the loop";
    cout << v.capacity();
    cout << "end of method";
    return 0;
}
```

lifetime of v

dstr is called

https://godbolt.org/z/3M1cqnoKo

https://godbolt.org/z/3M1cqnoKo

C++ source #1

```cpp
int main() {
    Vector v;
    {
        if (v.capacity() > 10) {
            Vector p{2};
        } else {
            Vector k{13};
        }
    }
    return v.capacity();
}
```

x86-64 gcc 13.2 (Editor #1) — x86-64 gcc 13.2, -O2

```asm
        push    rbp
        mov     esi, 16
        push    rbx
        sub     rsp, 72
        mov     rdi, rsp
        lea     rbp, [rsp+32]
        call    Vector::Vector(unsigned long) [complete object constructor]
        mov     rbx, QWORD PTR [rsp+16]
        cmp     rbx, 10
        jbe     .L20
        mov     esi, 2
        mov     rdi, rbp
        call    Vector::Vector(unsigned long) [complete object constructor]
.L26:
        mov     rdi, rbp
        call    Vector::~Vector() [complete object destructor]
        mov     rdi, rsp
        call    Vector::~Vector() [complete object destructor]
        add     rsp, 72
        mov     eax, ebx
        pop     rbx
        pop     rbp
        ret
.L20:
```

https://godbolt.org/z/3M1cqnoKo

C++ source #1 — C++

```cpp
int main() {
    Vector v;
    {
        if (v.capacity() > 10) {
            Vector p{2};
        } else {
            Vector k{13};
        }
    }
    return v.capacity();
}
```

x86-64 gcc 13.2 (Editor #1) — x86-64 gcc 13.2 — -O2

```asm
        push    rbp
        mov     esi, 16
        push    rbx
        sub     rsp, 72
        mov     rdi, rsp
        lea     rbp, [rsp+32]
        call    Vector::Vector(unsigned long) [complete object constructor]
        mov     rbx, QWORD PTR [rsp+16]
        cmp     rbx, 10
        jbe     .L20
        mov     esi, 2
        mov     rdi, rbp
        call    Vector::Vector(unsigned long) [complete object constructor]
.L26:
        mov     rdi, rbp
        call    Vector::~Vector() [complete object destructor]
        mov     rdi, rsp
        call    Vector::~Vector() [complete object destructor]
        add     rsp, 72
        mov     eax, ebx
        pop     rbx
        pop     rbp
        ret
.L20:
```

Object lifetime (first approximation, C-like)

When object dies? Depends on its storage duration:

- static   => when program terminates
  (return from main or call exit)
- automatic => at the end of the scope
- dynamic   => ?

Object lifetime (first approximation, C-like)

When object dies? Depends on its storage duration:

- static => when program terminates (return from main or call exit)
- automatic => at the end of the scope
- dynamic => when delete is called

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

delete operator is C++ way to deallocate objects previously allocated in dynamic memory (with new operator).

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

`delete` operator is C++ way to deallocate objects previously allocated in dynamic memory (with `new` operator).

`delete` operator:

- invokes destructor
- deallocates memory

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```cpp
int main() {
    Vector* v = new Vector{5};
    ...
    delete v;
    ...
    return 0;
}
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ~Vector() {
        delete[] data_;
        cout << "deleted" << endl;
    }
};
```

```cpp
int main() {
    Vector* v = new Vector{5};
    ...
    delete v;     dstr is called
    ...
    return 0;
}
```

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

`delete` operator is C++ way to deallocate objects previously allocated in dynamic memory (with `new` operator).

`delete` operator:

- invokes destructor
- deallocates memory

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

delete operator is C++ way to deallocate objects previously allocated in dynamic memory (with new operator).

delete operator:

- invokes destructor
- deallocates memory

delete[] operator:

- used only for arrays allocated with new!

- otherwise, UB

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

`delete` operator is C++ way to deallocate objects previously allocated in dynamic memory (with `new` operator).

`delete` operator:

- invokes destructor
- deallocates memory

`delete[]` operator:

- calls destructor for each element
- deallocates memory

```cpp
class Vector {
    int* data_;
    size_t size_;
    size_t capacity_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }

    ~Vector() {
        delete[] data_;
    }
};
```

```cpp
int main() {
    Vector* vs = new Vector[55];
    ...
    delete[] vs;   ← dstrs are called
    ...
    return 0;
}
```

Object lifetime (first approximation)

When object dies? Depends on its storage duration:

- static => when program terminates (return from main or call exit)
- automatic => at the end of the scope
- dynamic => when delete is called
- ??? => ???

Task: implement "growable array" of ints data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct. ✓

2. Implementation details are accessible to the user. ✓

3. Inconsistent state of an object.

Task: implement "growable array" of ints
data structure in C

Problems:

1. Code that works with the structure is separated. No connection to the struct. ✓

2. Implementation details are accessible to the user. ✓

3. Inconsistent state of an object. ✓

Task: implement "growable array" of ints
data structure in C

More problems?

Task: implement "growable array" of ints
data structure in C

More problems:

1. When and how else objects can be created?

Task: implement "growable array" of ints data structure in C

More problems:

1. When and how else objects can be created?

2. How to generalize it to other types?

3. Can we initialize Vector with some (variadic number of) elements in ctr?

Task: implement "growable array" of ints
data structure in C

More problems:

1. When and how else objects can be created?

2. How to generalize it to other types?

3. Can we initialize Vector with some
(variadic number of) elements in ctr?

4. Performance? Multithreading?

Takeaways

- Encapsulation:
  - bundling data (fields) and logic (method) together
  - hiding internals of classes to avoid breaking invariants by users

Takeaways

- Encapsulation in C++:
  - structs and classes
  - access modifiers
  - constructors and destructors
  - more to discuss!

Not So Tiny Task №1 (2 points)

Choose one of these well known data structures: AVL-tree, Treap or Fibonacci Heap.

Implement it on C++ and design a class that represents the data structure.

Add constructors, destructors, public and private fields to keep needed invariants.

Prepare tests and check your code with sanitizers.