20. Dynamic Data Structures I

Dynamic Memory, Addresses and Pointers, Const-Pointer Arrays, Array-based Vectors

Recap: vector<T>

- Can be initialised with arbitrary size **n**
- Supports various operations:

```
e = v[i];           // Get element
v[i] = e;           // Set element
l = v.size();       // Get size
v.push_front(e);    // Prepend element
v.push_back(e);     // Append element
...
```

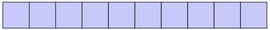
- A vector is a *dynamic data structure*, whose size may change at runtime

Our Own Vector!

- Today, we'll implement our own vector: **vec**
- Step 1: **vec<int>** (today)
- Step 2: **vec<**T**>** (later, only superficially)

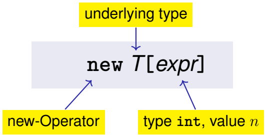
Vectors in Memory

Already known: A vector has a *contiguous* memory layout



Question: How to *allocate* a chunk of memory of *arbitrary* size during runtime, i.e. *dynamically*?

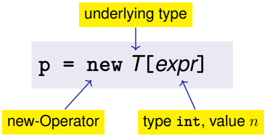
`new` for Arrays



- **Effect**: new contiguous chunk of memory n elements of type *T* is allocated

- This chunk of memory is called an *array* (of length n)

`new` for Arrays



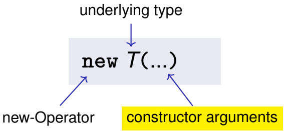
- **Type**: A pointer $T*$ (more soon)
- **Value**: the starting address of the memory chunk

Outlook: `new` and `delete`

`new` *T*[*expr*]

- So far: memory (local variables, function arguments) “lives” only inside a function call
- But now: memory chunk inside vector must not “die” before the vector itself
- Memory allocated with `new` is *not* automatically deallocated (= released)
- Every `new` must have a matching `delete` that releases the memory explicitly → **in two weeks**

`new` (Without Arrays)



- **Effect**: memory for a new object of type *T* is allocated . . .
- . . . and initialized by means of the matching constructor
- **Value**: address of the new T object, **Type**: Pointer $T*$
- Also true here: object “lives” until deleted explicitly (usefulness will become clearer later)

Pointer Types

T∗ Pointer type for base type `T`

An expression of type `T*` is called *pointer (to `T`)*

```
int* p; // Pointer to an int
std::string* q; // Pointer to a std::string
```


Pointer Types

T∗ Pointer type for base type `T`

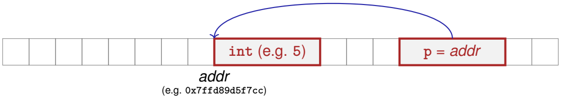
A $T*$ must actually point to a T

```
int* p = ...;
std::string* q = p; // compiler error!
```


Pointer Types

Value of a pointer to `T` is the *address* of an object of type `T`

```
int* p = ...;
std::cout << p; // e.g. 0x7ffd89d5f7cc
```



Address Operator

Question: How to obtain an object's address?

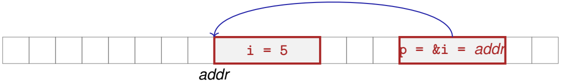
1. Directly, when creating a new object via `new`
2. For existing objects: via the *address operator* `&`

&expr ← expr: l-value of type T

- **Value** of the expression: the *address* of object (l-value) *expr*
- **Type** of the expression: A pointer $T*$ (of type T)

Address Operator

```
int i = 5; // i initialised with 5
int∗ p = &i; // p initialised with address of i
```



Next question: How to “follow” a pointer?

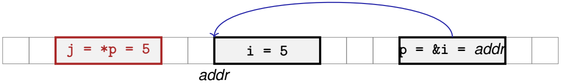
Dereference Operator

Answer: by using the *dereference operator* ∗

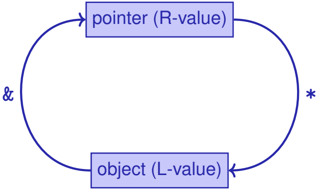
- **Value** of the expression: the *value* of the object located at the address denoted by *expr*
- **Type** of the expression: T

Dereference Operator

```
int i = 5;
int∗ p = &i; // p = address of i
int j = ∗p; // j = 5
```



Address and Dereference Operator



Mnenmonic Trick

The declaration

```
T* p;      // p is of the type “pointer to T”
```

can be read as

```
T *p;      // *p is of type T
```

Although this is legal, we do not write it like this!

Null-Pointer

- Special pointer value that signals that no object is pointed to
- represented b the literal `nullptr` (convertible to `T*`)

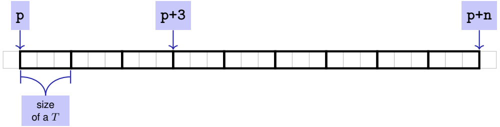
```
int* p = nullptr;
```

- Cannot be dereferenced (runtime error)
- Exists to avoid undefined behaviour

```
int* p; // p could point to anything
int* q = nullptr; // q explicitly points nowhere
```


Pointer Arithmetic: Pointer plus `int`

```
T* p = new T[n]; // p points to first array element
```

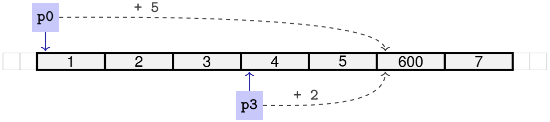


How to point to rear elements? → *Pointer arithmetic*:

- `p` yields the *value* of the *first* array element, `*p` its *value*
- `*(p + i)` yields the value of the `i`*th* array element, for $0 \leq$ `i` $<$ `n`
- `*p` is equivalent to `*(p + 0)`

Pointer Arithmetic: Pointer plus `int`

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
int* p3 = p0 + 3; // p3 points to 4th element
*(p3 + 2) = 600; // set value of 6th element to 600
std::cout << *(p0 + 5); // output 6th element’s value (i.e. 600)
```



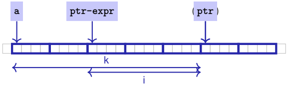
Pointer Arithmetic: Pointer minus `int`

- If *ptr* is a pointer to the element with index k in an array a with length n
- and the value of *expr* is an integer i, $0 \leq k - i \leq n$,

then the expression

ptr - expr

provides a pointer to an element of a with index $k - i$.



Pointer Subtraction

- If *p1* and *p2* point to elements of the same array `a` with length n
- and $0 \leq k_1, k_2 \leq n$ are the indices corresponding to *p1* and *p2*, then

p1 - p2 has value k_1 - k_2

Only valid if *p1* and p2 point into the same array.

- The pointer difference describes "how far away the elements are from each other"

Pointer Operators

Description	Op	Arity	Precedence	Associativity	Assignment
Subscript	[]	2	17	left	R-value → L-value
Dereference	*	1	16	right	R-Wert → L-Wert
Address	&	1	16	rechts	L-value → R-value

Precedences and associativities of +, -, ++ (etc.) as in Chapter 2

Pointers are not Integers!

- Addresses can be interpreted as house numbers of the memory, that is, integers
- But integer and pointer arithmetic behave differently.

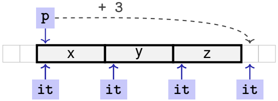
`ptr + 1` is *not* the next house number but the s-next, where s is the memory requirement of an object of the type behind the pointer `ptr`.

- Integers and pointers are not compatible

```
int* ptr = 5; // error: invalid conversion from int to int*
int a = ptr;  // error: invalid conversion from int* to int
```

Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

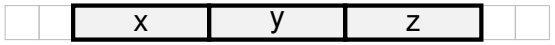



```
for (char* it = p;        // it points to first element
     it != p + 3;         // Abort if end reached
     ++it) {              // Advance pointer element-wise

  std::cout << *it << ' '; // Output current element: 'x'
}
```


Random Access to Arrays

```
char* p = new char[3]{'x', 'y', 'z'};
```



- The expression `*(p + i)`
- can also be written as `p[i]`
- E.g. `p[1] == *(p + 1) == 'y'`

Random Access to Arrays

iteration over an array via indices and *random access*:

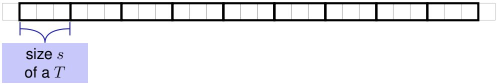
```
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
  std::cout << p[i] << ' ';
```

But: this is less *efficient* than the previously shown *sequential* access via pointer iteration

Random Access to Arrays

```
T* p = new T[n];
```



- Access `p[i]`, i.e. `*(p + i)`, "costs" computation $p + i \cdot s$
- Iteration via *random access* (`p[0]`, `p[1]`, ...) costs one addition and one multiplication per access
- Iteration via *sequentiall access* (`++p`, `++p`, ...) costs only one addition per access
- Sequential access is thus to be preferred for iterations

Reading a book . . . with random access . . . with sequential access

Random Access

- open book on page 1
- close book
- open book on pages 2-3
- close book
- open book on pages 4-5
- close book
- ....

Sequential Access

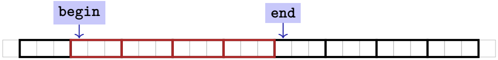
- open book on page 1
- turn the page
- turn the page
- turn the page
- turn the page
- turn the page
- ...

Static Arrays

- `int* p = new int[`*expr*`]` creates a dynamic array of size *expr*
- C++ has inherited *static* arrays from its predecessor language C: `int a[`*cexpr*`]`
- Static arrays have, among others, the disadvantage that their size *cexpr* must be a constant. I.e. *cexpr* can, e.g. be `5` or `4*3+2`, but kein von der Tastatur eingelesener Wert `n`.
- A static array variable `a` can be used just like a pointer
- Rule of thumb: Vectors are better than dynamic arrays, which are better than static arrays

Arrays in Functions

C++ *covention*: arrays (or a segment of it) are passed using two pointers



- `begin`: Pointer to the first element
- `end`: Pointer *past* the last element
- `[begin, end)` Designates the elements of the segment of the array
- `[begin, end)` is empty if `begin == end`
- `[begin, end)` must be a *valid range*, i.e. a (pot. empty) array segment

Arrays in (mutating) Functions: `fill`

```
// PRE: [begin, end) is a valid range
// POST: Every element within [begin, end) was set to value
void fill(int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
    *p = value;
}

...
int* p = new int[5];
fill(p, p+5, 1); // Array at p becomes {1, 1, 1, 1, 1}
```

Functions with/without Effect

- Pointers can (like references) be used for functions with effect. Example: `fill`
- But many functions don't have an effect, they only read the data
- ⇒ Use of `const`
- So far, for example:

```
const int zero = 0;
const int& nil = zero;
```


Positioning of Const

Where does the `const`-modifier belong to?

`const` T is equivalent to T `const` (and can be written like this):

```
const int zero = ...    ⟺   int const zero = ...
const int& nil = ...    ⟺   int const& nil = ...
```

Both keyword orders are used in praxis

Const and Pointers

Read the declaration from right to left

`int const p;`	`p` is a constant integer
`int const* p;`	`p` is a pointer to a constant integer
`int* const p;`	`p` is a constant pointer to an integer
`int const* const p;`	`p` is a constant pointer to a constant integer

Non-mutating Functions: `print`

There are also *non*-mutating functions that access elements of an array only in a read-only fashion

```
// PRE: [begin, end) is a valid range
// POST: The values in [begin, end) were printed
void print(
    int const* const begin,
    const int* const end) {

  for (int const* p = begin; p != end; ++p)
    std::cout << *p << ' ';
}
```

Const pointer to const int (`begin`)

Likewise (but different keyword order) (`end`)

Pointer, *not const*, to const int (`p`)

Pointer `p` may itself not be `const` since it is mutated (`++p`)

const is not absolute

- The value at an address can change even if a `const`-pointer stores this address.

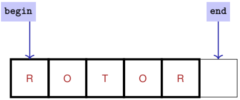
beispiel

```
int a[5];
const int* begin1 = a;
int*       begin2 = a;
*begin1 = 1;    // error *begin1 is const
*begin2 = 1;    // ok, although *begin will be modified
```

- `const` is a promise from the point of view of the `const`-pointer, not an absolute guarantee

Wow – Palindromes!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new` T`[`n`]` allocates a T-array of size n
- T`* p = new` T`[`n`]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new` T allocates memory for (and initialises) a single T-object, and yields a pointer to it
- Pointers can point to something (not) `const`, and they can be (not) `const` themselves
- Memory allocated by `new` is *not* automatically released (more on this soon)
- Pointers and references are related, both “link” to objects in memory. See also additional the slides `pointers.pdf`)

Array-based Vector

- Vectors ... that somehow rings a bell 🤔
- Now we know how to allocate memory chunks of arbitrary size ...
- ... we can implement a vector, based on such a chunk of memory
- `avec` – an array-based vector of `int` elements

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<`T`>` (später, nur kurz angeschnitten)

Array-based Vector `avec`: Class Signature

```
class avec {
  // Private (internal) state:
  int* elements; // Pointer to first element
  unsigned int count; // Number of elements

public: // Public interface:
  avec(unsigned int size); // Constructor
  unsigned int size() const; // Size of vector
  int& operator[](int i); // Access an element
  void print(std::ostream& sink) const; // Output elems.
}
```


Constructor `avec::avec()`

```
avec::avec(unsigned int size)
        : count(size) {          ← Save size

  elements = new int[size];      ← Allocate memory
}
```

Side remark: vector is not initialised with a default value

Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
  this->elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the *pointer* `this`
- `elements` is a shorthand for `(*this).elements`
- Dereferencing a pointer (`*this`) followed by a member access (`.elements`) is such a common operation that it can be written more concisely as `this->elements`
- Mnemonic trick: "Follow the pointer to the member variable"

Function `avec::size()`

```
int avec::size() const {      ← Doesn't modify the vector
  return this->count;         ← Return size
}
```

Usage example:

```
avec v = avec(7);
assert(v.size() == 7); // ok
```

Function avec::operator[]

```
int& avec::operator[](int i) {
  return this->elements[i];   ← Return ith element
}
```

Element access with index check:

```
int& avec::at(int i) const {
  assert(0 <= i && i < this->count);

  return this->elements[i];
}
```


Function avec::operator[]

```
int& avec::operator[](int i) {
  return this->elements[i];
}
```

Usage example:

```
avec v = avec(7);
std::cout << v[6]; // Outputs a "random" value
v[6] = 0;
std::cout << v[6]; // Outputs 0
```


Function avec::operator[] is needed twice

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return elements[i]; }
```

- The first member function is *not const* and returns a *non-const* reference

```
avec v = ...; // A non-const vector
std::cout << v.get[0]; // Reading elements is allowed
v.get[0] = 123; // Modifying elements is allowed
```

- It is called on non-const vectors

Function avec::operator[] is needed twice

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return elements[i]; }
```

- The second member function *is const* and returns a *const* reference

```
const avec v = ...; // A const vector
std::cout << v.get[0]; // Reading elements is allowed
v.get[0] = 123; // Compiler error: modifications are not
   allowed
```

- It is called on const vectors

Also see the example attached to this PDF

Function `avec::print()`

Output elements using sequential access:

```
void avec::print(std::ostream& sink) const {
  for (int* p = this->elements;
       p != this->elements + this->count;
       ++p)
  {
    sink << *p << ' ';
  }
}
```

Pointer to first element

Abort iteration if past last element

Advance pointer element-wise

Output current element

Function `avec::print()`

Finally: overload output operator:

```
________________ operator<<(________________ sink,
                            ________________ vec) {
  vec.print(sink);
  return ________________;
}
```

```
std::ostream& operator<<(std::ostream& sink,
                         const avec& vec) {
  vec.print(sink);
  return sink;
}
```

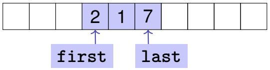

Further Functions?

```
class avec {
  ...
  void push_front(int e)      // Prepend e to vector
  void push_back(int e)       // Append e to vector
  void remove(unsigned int i) // Cut out ith element
  ...
}
```

Commonalities: such operations need to change the vector's *size*

Resizing arrays

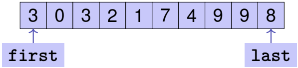
An allocated block of memory (e.g. `new int[3]`) cannot be resized later on



Possibility:

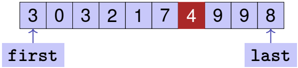
- Allocate more memory than initially necessary
- Fill from inside out, with pointers to first and last element

Resizing arrays

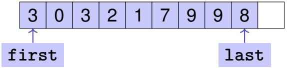


- But eventually, all slots will be in use
- Then unavoidable: Allocate larger memory block and copy data over

Resizing arrays



Deleting elements requires shifting (by copying) all preceding or following elements



Similar: inserting at arbitrary position