EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF SCIENCE

Applied Bioinformatics

Programming in C/C++

- Data Structures -

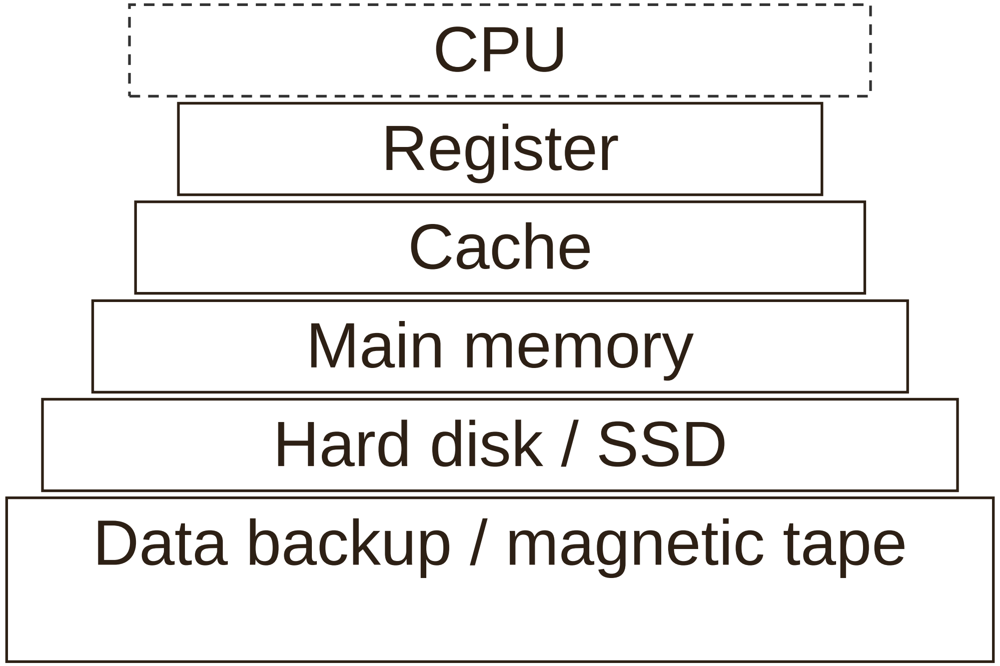
Memory

- Memory hierarchy
- Virtual address space
- C++

Data in Memory

- Users and programmers would love to have infinite memory that is fast and, if possible, persistent.

- **A practical solution**: Memory hierarchy



Order of magnitude / `Access time`

kB	`< ns`
MB	`ns`
GB	`50ns`
TB	`9ms / .25ms`
10xTB	`s`

Physical Memory and Virtual Address Space

- On a modern systems, many programs run simultaneously and share memory.

- Each program has its **own virtual address space** with 32 or 64bit addresses
  - Address space is usually significantly larger than the working memory
  - Address ranges are provided by the **OS** in physical memory on tiles (mapping with page table, details in *Computer Engineering*).
  - **Physical storage space is limited**

- C++ has different built-in mechanisms to manage memory inside its virtual address space.

- There is **no garbage collector** for dynamically managed memory (heap) in C++
  - you must clean up yourself!
  - since C11: **smart pointers** offer simple means for automated clean up of resources

Memory Areas and Allocation

Depending on how a C++ uses data different areas of the virtual address space are used.

- **static memory:**
  - Fixed address relative to the program code
  - Global variables, static variables
  - Memory requirements determined during compilation/linking, allocated on program start and released on exit

- **automatic memory:** On the **stack** (last in first out)
  - Local variables, function parameters, return values
  - Allocation and deallocation happens automatically during execution of program

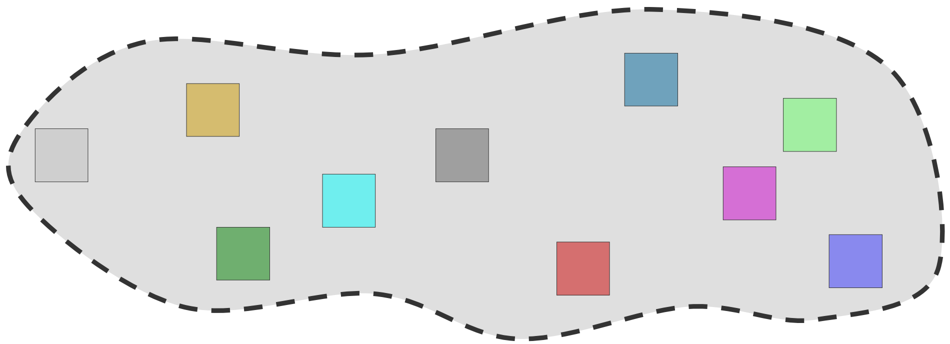
- **dynamic memory:** On the **heap**
  - Manual allocation during the execution of the program with explicit instructions
  - In C++: with `new, new[], delete, delete[]`
  - In C: with `malloc, free, …`

arrays

- How to store many elements in memory
- Low-level C-style arrays
- How arrays and pointers are related
- The two types of memory: stack and heap
- A modern alternative for allocation of arrays on the stack: `std::array`

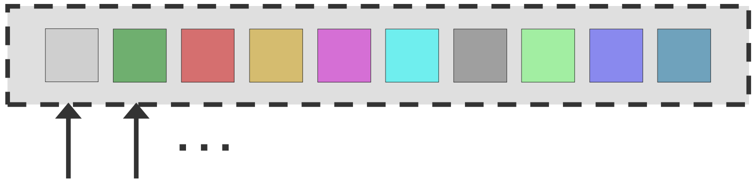
Motivation

- Given a set of elements with different values = colors but same type



- **Goal: Combining** many elements into a **structure**.

- If we put elements into a **linear arrangement** we can access them by an **index.**



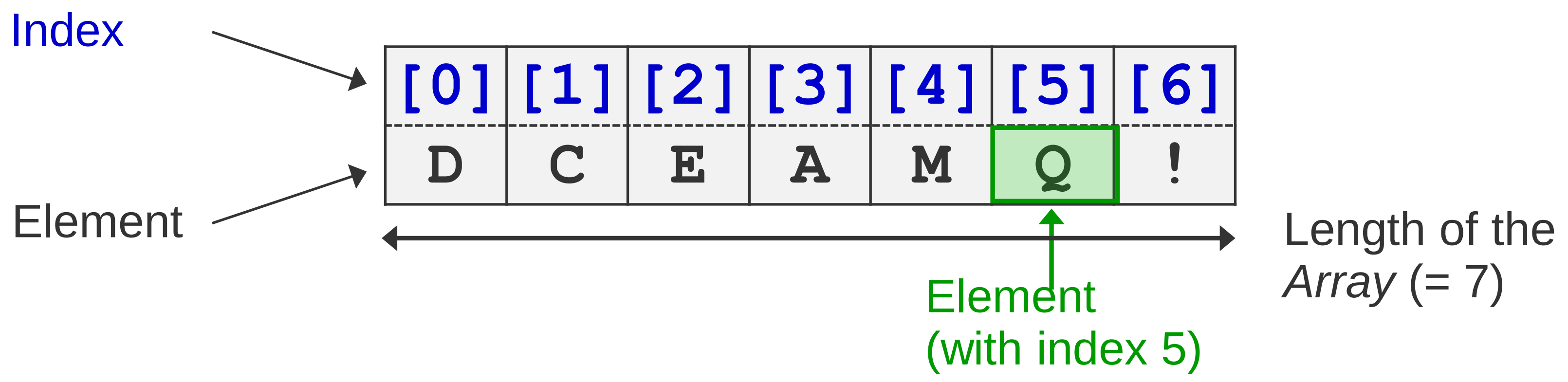
- Once we can access them by index, we can **process all in a loop**.

Collections of Data Objects in Arrays

- An array is a **container** (data structure) with:
  - **a fixed number of values**
  - **of a defined type**

- The elements in an array can be accessed by an index starting at 0.

- The index is implicit and doesn‘t need extra memory.



Disclaimer:

- Slides dealing with pointer and arrays contains code that is error prone.
- Memorize the concept of pointers to be able to understand C or older C++ code but try to avoid them in your code.
- Many errors in C++ programs can be traced back to the use of pointers.
- Most pointers you encounter in actual C++ code could be avoided.
- Still useful to learn about pointers as they teach us a lot about internals.

Declaration, Creation and Initialization of Arrays

1. **Declaration of** the pointer variables:

```
int* numberfield;
```

C-style
low-level memory allocation

2. **Create / reserve** the necessary memory (setting the length):

```
numberfield = (int*) alloca(5 * sizeof(int));
```

alloca
on the stack

3. **Access to** elements of the array

```
numberfield[3] = 4;
cout << numberfield[3];
```

4. **Releasing** the memory (automatic at end of function)

```
```

Declaration, Creation and Initialization of Arrays

C-style
low-level memory allocation

1. **Declaration of** the pointer variables:

```
int* numberfield;
```

2. **Create / reserve** the necessary memory (setting the length):

```
numberfield = (int*) malloc(5 * sizeof(int));
```

malloc
on the heap

3. **Access to** elements of the array

```
numberfield[3] = 4;
cout << numberfield[3];
```

4. **Releasing** the memory

```
free( (void*) numberfield);
```

Declaration, Creation and Initialization of Arrays

C++-style
low-level memory allocation

1. **Declaration of** the pointer variables:

```
int* numberfield;
```

2. **Create / reserve** the necessary memory (setting the length):

on the heap

```
numberfield = new int[5];
```

3. **Access to** elements of the array

```
numberfield[3] = 4;
cout << numberfield[3];
```

4. **Releasing** the memory (works for more complex types by calling **destructors**)

```
delete[] numberfield;
```

Declaration, Creation and Initialization of Arrays – Alternative

1. **Declaration of** the pointer variables and **creation / reservation of** the memory:

```
int numberfield[5];
```

C++
on the stack

2. **Access to** elements of the array

```
numberfield[3] = 4;
cout << numberfield[3];
```

3. **Releasing** of the memory (automatic at end of block)

Stack vs. Heap

- Stack memory:
  - **parameters** passed to functions, **local variables**
  - **automatic memory management**, First in Last out
  - **faster** (everything fits in CPU cache)
  - **small** (e.g., 1 MB)

- Heap memory:
  - **manual memory management**
  - **larger** (GBs or TBs)
  - **slower** (only parts fit/need to be loaded into CPU cache)

Now you know everything to pay tribute to the famous webpage and create a stack overflow in your program ;)

Example: Access to Elements of the Array

```
float prices[100];          // on stack, automatic memory management

prices[3] = 2.3f;           // write access, assign a value
float price = prices[3];    // read access
int a = 2;
price = prices[4 * a + 7]; // also works: read access via complex expression as index
```

```
float * prices = new float[100]; // on heap, manual memory management

// …

delete[prices];                  // free memory
```

- The pointer variable stores the memory address of the first element.
- Accessing an element with operator [] (internally) calculates the memory address of that element and returns the data it points to.

Array Creation and Initialization

Common pitfalls:

- The contents of a pointer variable **may not** be immediately accessed **after declaration.**

```
float * prices;
float p = prices[3]; // error!
```

- A **fixed length data field** for the elements needs to be **created/reserved in memory** first.

Correct construction of the array would have been:

```
float * prices = new float[4]; // new float array of size 4
float p = prices[3];           // now we can access the element
```

But: array elements are uninitialized. We don't know what value `p` will contain.

Incorrect Access to Array Elements

- **No check** is made when accessing the elements of an array, whether the index used is in the permissible range: **`0 <= index value < array length`**

If you try to access an illegal index, unallocated memory or memory used in a different way may be accessed!

```
int a[8];                    // Arrays with 8 ints
int b[8];
b[0] = 15;                   // set first element of b to 15
cout << b[0] << endl;

a[8 - 1] = 2; // OK, access to last element
a[8];         // may abort program or e.g., point to b[0]
cout << a[8] << endl;
```

```
Potential output
15
15 ??? What!?!
```

```
a[1000000];  // Program abort (Segmentation Fault)
```

Variants for the Generation and Initialization of Elements

- When an array is created, the individual elements are **default initialized**

```
int a[5]; // default initialization of int = indetermined
```

- Different forms of generation and initialization of data fields are possible:
  - **Creation of** an array **at the declaration of** the array variable;

```
char someWord[5];
```

  - **Create by specifying the set of elements** in curly brackets `{}` (*array initializer*);
  - Elements can be **constants**, **variables** or arbitrary **expressions**.
  - **The length is determined automatically by compiler.**

```
int primes[] = {2, 3, 5, varA, 11, 13+5, 17, 19};
```

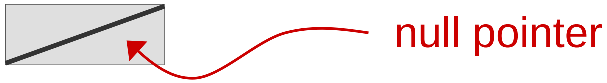
also:

```
int primes[] = new int[]{2, 3, 5, 7, 11, 13, 17, 19};
  ...
delete[] primes;
```

Arrays - Reference, Pointer Variables

- When declaring a variable of the array type, only the **pointer to the data is** introduced.

- Pointer should be set to **`nullptr`**, or NULL in C)




```
int* list = nullptr;
```

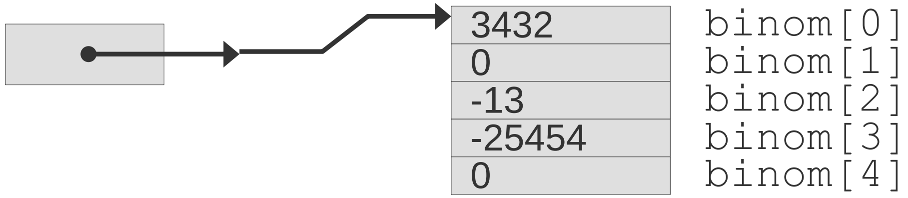
- The **`new`** operator requests and reserves **memory** for the field

  The elements of the array are default initialized which means: **not initialized** for primitive data types. They can be zero - but are not guarenteed to be.

```
int* binom;
binom = new int[5];
```

or shorter:

```
int* binom = new int[5];
```

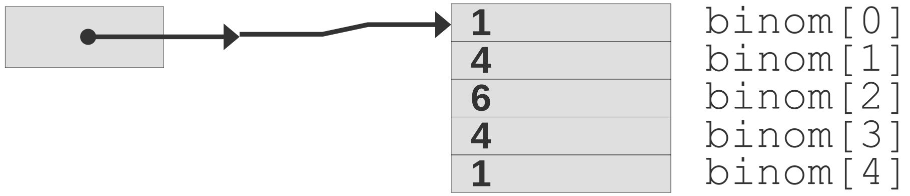


Arrays - Reference, Pointer Variables (2)

- **Explicit initialization** is used to assign **values** to the field elements.

- The **initialization must be** done **simultaneously** with the **declaration** or **allocation**.

```
int* binom;
binom = new int[] { 1, 4, 6, 4, 1 };
// or:
int* binom = new int[] { 1, 4, 6, 4, 1 };
// or:
int binom[] = { 1, 4, 6, 4, 1 };
```



Note:

```
binom = { 1, 4, 6, 4, 1 }; // assignment doesn't work
```

No Initialization of Array Elements

Example: allocation of an `Array` with **primitive types** does not initialize the values.

```
const int l{100000};
int a[l];                 // allocates but does not initialize elements to zero
int count{0};
for (int i = 0; i < l; i++) {
  if (a[i] != 0) {        // count non-zero elements
    count++;
  }
}
cout << count << " non-zero values!" << endl; // count typically > 0 !!!
```

outlook: but for classes / objects `new` calls the constructor automatically… (later more)

Assignments - Arrays vs. element of the array

- **We already saw: Assignment to an element** of the array

```
int *list = new int[7];
list[5] = 12; // assign 12 to the 6th element of list
```

- **Now: Assignment to the pointer variable** itself

```
int b[12];
int *list;
list = b; // *list also points to b’s data
```

Explanation:

- Assigning a pointer variable to another array variable causes both variables to point to the same data!

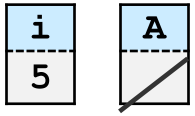
Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

```
int   i = 5;
int*  A;
```

Declaration of an **`int variable`** `i`, initialization with 5

Declaration of an *array variable* `A` (pointer = **`null`**)

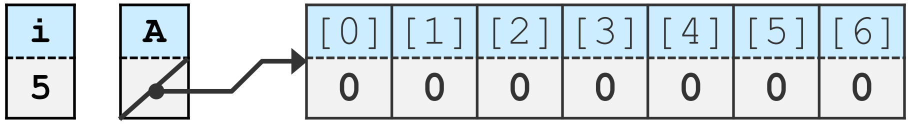


Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

```
int    i = 5;
int*   A;
A    = new int[7];
```

Generating an *array* for the variable `A` with 7 **`int elements`**.



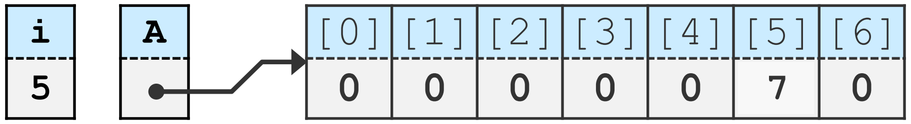
Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

```
int    i = 5;
int*   A;
A      = new int[7];

A[i] = 7;
```

The element with index `i = 5 of` the *array* `A` is assigned the value 7.



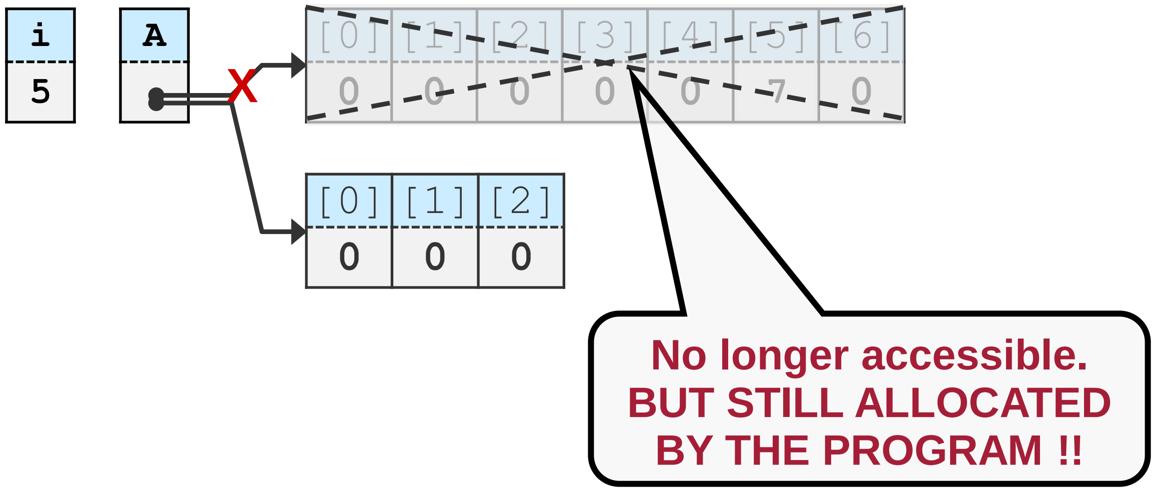
Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

```
int    i = 5;
int*   A;
A     = new int[7];

A[i] = 7;
A     = new int[3];
```

Generate a new *array* for variable `A` with 3 **`int elements`**.



Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

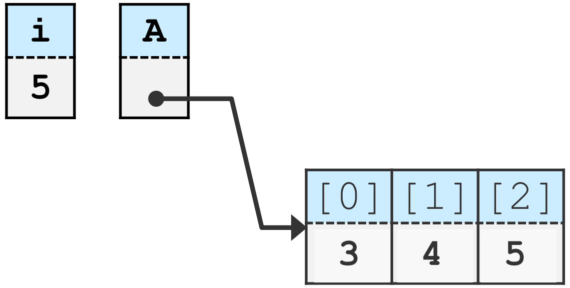
```
int   i = 5;
int*  A;
A     = new int[7];

A[i] = 7;
A     = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;
```

The element with index `0` of the (new) *array* `A` is assigned the value 3...

... the element with index `1` the value 4 ...

... the element with the index `2` the value 5 ...



Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

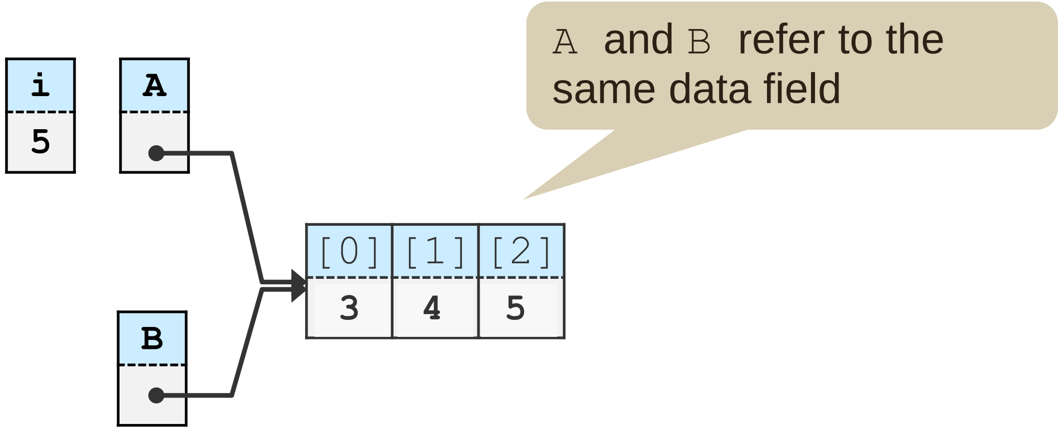
```
int    i = 5;
int*   A;
A      = new int[7];

A[i] = 7;
A      = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;

int* B;
B      = A;
```

Declaration of an *array variable* `B` (pointer = **`null`**).

Assign pointer `A` to `B`; thus `B` also points to array `A` and its contents.



Example – Pointers and Arrays

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

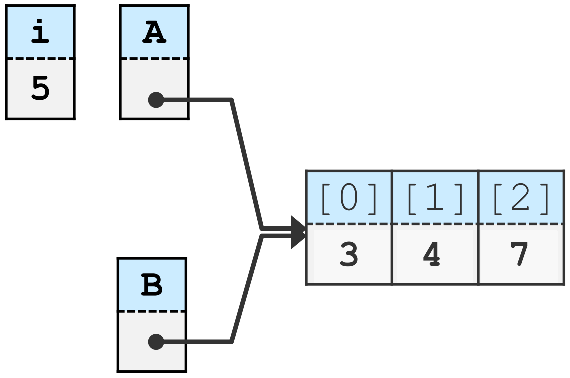
- Sequence of declarations and statements with state changes caused by the data objects and their values.

```
int    i = 5;
int*   A;
A      = new int[7];

A[i] = 7;
A     = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;

int* B;
B     = A;
B[2] = A[0] + 4;
```

The element with index `2` of the *array* pointing to `B` `is` assigned the value of the element with index 0 of the *array* pointing to `A` incremented by 4.



identical with `B[0]`

Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

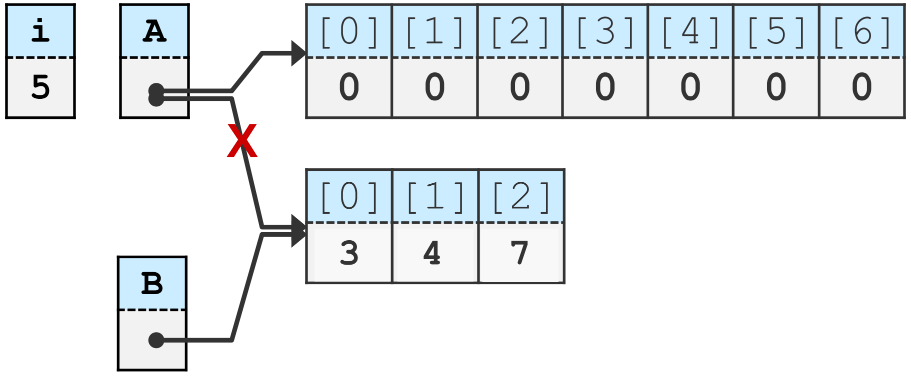
```
int   i = 5;
int*  A;
A     = new int[7];

A[i] = 7;
A     = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;

int* B;
B     = A;
B[2] = A[0] + 4;

A     = new int[7];
```

Generate a new *array* for variable `A` with 7 **`int elements`**.



Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

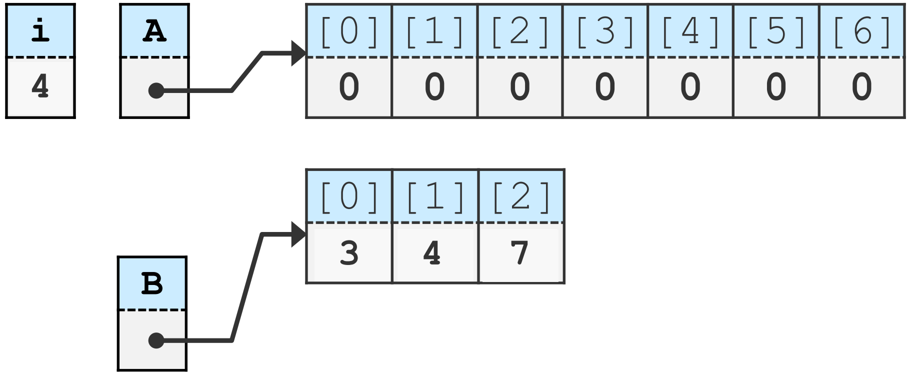
```
int    i = 5;
int*   A;
A     = new int[7];

A[i] = 7;
A     = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;

int* B;
B     = A;
B[2] = A[0] + 4;

A     = new int[7];
i     = B[0] + 1;
```

The value of the index variable `i` is changed; the new value is `i = 4`.



Example – Pointers and Arrays

- Sequence of declarations and statements with state changes caused by the data objects and their values.

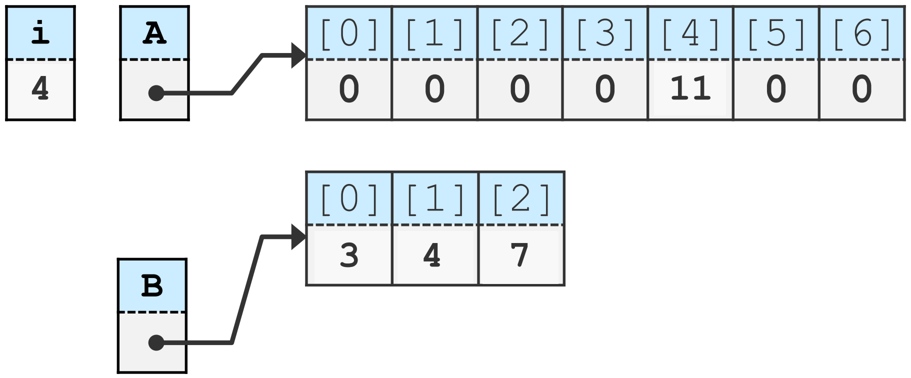
```
int   i = 5;
int*  A;
A    = new int[7];

A[i] = 7;
A    = new int[3];
A[0] = 3;
A[1] = 4;
A[2] = 5;

int* B;
B    = A;
B[2] = A[0] + 4;

A    = new int[7];
i    = B[0] + 1;
A[i] = B[2] + B[1];
```

The element with index `i = 4` of *array* `A is`
assigned the sum of the elements with index `2`
and `1 of` array `B.`



Standard Calculations on Arrays

Summation of elements

- **Goal**: **Addition of all values** in an array of doubles: `double arr[512]`

```
double sum = 0;
for (int i = 0; i < 512; i++) {
  sum += arr[i];
}
```

Counting elements

- **Goal**: **Count all elements with negative value in** an array, in which case we **set it to zero.**

```
int count = 0;
for (int i = 0; i != 512; ++i) {
  if (arr[i] < 0.0) {
    arr[i] = 0.0;
    count++;
  }
}
```

Standard Calculations on Arrays (2)

- **Goal**: Count all pairs of adjacent elements with the same value in an array.

```
int count = 0;
for (int i = 0; i < 512 - 1; i++) {
  if (arr[i] == arr[i + 1]) {
    count++;
  }
}
```

Finding the largest element

- **Goal**: determine the element with the **largest value** in an array

```
double maxVal = arr[0];
for (int i = 1; i < 512; i++) {
  if (arr[i] > maxVal)
    maxVal = arr[i];
}
```

Note: Some of these functions „reinvent the wheel“ and could be replaced by functions in the standard template library (later more).

Explicitly Copy the Contents of an Array

- **Goal**: Create a complete copy of an existing array `double arr[512]`

- **Reminder:** The assignment `double B[512] = arr;` does not copy.
  `B` subsequently contains only the same address and both variables `B` and `arr` refer to the same memory

- **Solution:** element-wise copy

```
double B[512];
for (int i = 0; i < 512; i++) {
  B[i] = arr[i];
}
```

`std::array` - a **modern, better option (since C++11)**

- `std::array` class allocates on the stack

- Behaves like C-style arrays

`#include <array>`

```
// Initializing the array elements
array<int,6> arr = {1, 2, 3, 4, 5, 6};

for ( int i = 0; i < 6; i++)
   cout << arr[i] << " ";
```

- But:
  - Array classes know their size -> no need to pass size of array as a separate function parameter.
  - Extra functions like `.at(i)` that performs bounds checks
  - Has many additional helper functions to swap elements, fill values, iterators, etc.

- Still one downside: like C-arrays they can't grow (outlook: `std::vector` solves that problem)

strings

- Old-style C-strings are char arrays
- std::string as C++ alternative

Strings `char[]` - Old-style C-strings

C-strings: old style type for **strings**

- **`char` array**
- last character is always: `\0` (implicit)
- **fixed memory size**
- Global, C-style functions to work with strings (e.g., `strlen`)

```
#include <string.h>

char sentence[] = "Old style C-string";

cout << sentence << endl;
cout << sentence[2] << endl;
cout << strlen(sentence) << endl;
```

C++ std::string Class

`#include <string>`

- Our first **STL container**
- Dynamic memory management, can grow dynamically, request size with **size()**
- Common container methods **and** methods and operators to operate on strings (concatenate, insert, replace, …)

```
const string s1 = "Not a sentence";
string s2("This is");
s2 += s1;                                   // concatenate s2 and s1, grow!
s2.insert(7, " ");
s2.replace(8, 1, "n");

cout << s2 << endl;                         // ”This is not a sentence”
cout << sentence[2] << endl;
cout << sentence.size() << endl;

sentence.clear();                           // clear string -> now empty
cout << sentence.empty() << endl;
```

Recommendation: prefer **`std::string`** over char array

Special characters start with a backslash "\" e.g.:

- `\n` newline
- `\t` tabulator
- `\"` quote
- `\\` backslash

Example:

```
string s =  "\t\"Test\"Test\n";
cout << s << endl; // [Tab]"Test" Test
```

C++ character manipulation `#include <cctype>`

- Manipulation of single characters in strings with `cctype` header

- Useful functions:

  `char tolower(char)`

  and

  `char toupper(char)`

  convert character into lower/upper case

```
string s("Hello, world!");
for (size_t i = 0; i < s.size(); ++i)
{
    s[i] = toupper(s[i]);
}

// s = "HELLO, WORLD!"
```

C++ character classes

`#include <cctype>`

- Functions to **determine** if characters are part of a certain **character class**

- Return value **`bool`**:
  - **`isalnum(c)`** true, if c is a letter or digit
  - **`isalpha(c)`** true, if c is a letter
  - **`iscntrl(c)`** true, if c is a control character
  - **`isdigit(c)`** true, if c is a digit
  - **`islower(c)`** true, if c is a lower-case letter
  - **`isspace(c)`** true, if c is a white-space character
  - **`isupper(c)`** true, if c is an upper-case character
  - ...

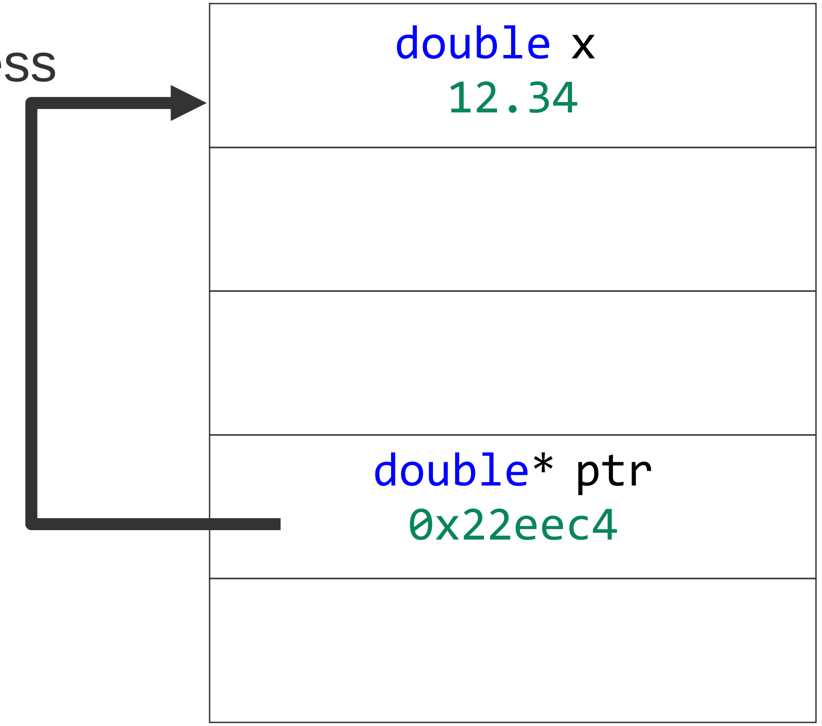
Pointers & Memory Management

- Calculate with pointers
- Where is the data located in the memory?

Pointer

- Data is stored somewhere in the **address space** of the program
- **Pointers** point to the corresponding location: they **hold the memory address of data**
- Address operator `&` returns the address
- Dereferencing operator `*` access data at the given address

```
double x{12.34};
double* ptr; // pointer on double
ptr = &x;    // assign address of x to ptr
```



Pointer

- Data is stored somewhere in the **address space** of the program
- **Pointers** point to the corresponding location: they **hold the memory address of data**
- Operators for arithmetic on pointers: `+ ++ - -- = += -= ==`

0x means hexadecimal

```
char sentence[] = "Old style C-string";
char* ptrSentence;
ptrSentence = &sentence[0]; // address operator& returns address to first character
cout << *ptrSentence << " and " << *(ptrSentence+2) << endl;

cout << std::hex << (size_t) ptrSentence << " to " << (size_t) ptrSentence+18 << std::dec << endl;
```

Pointer – Properties

- Pointers have a **fixed size** independent of the data type they point to (typical: 32bit or 64bit based on the architecture) because pointer hold a memory address.

```
int numbers[] = {0, 1,  2,  3,  4,  5,  6,  7,  8,
                  9, 10, 11, 12, 13, 14, 15, 16, 17};

int *ptrNumber = &numbers[0];

cout <<  ptrNumber << " to " <<  (ptrNumber + 17) << endl;   // address
cout << *ptrNumber << " to " << *(ptrNumber + 17) << endl;   // data

cout << "Calculate array length (Bytes): "
     << size_t(ptrNumber + 17) - size_t(ptrNumber) + 1 * sizeof(int)
     << endl;
```

Change Pointer Type (Cast)

- **We know already: casts change the type of variables**

```
double a = 222.0;
// int b = (int)a;              // old C-style cast
int b = int(a);                 // C++ type conversion
int c = static_cast<int>(a); // C++-style cast
```

- **Now: Casting pointers (danger zone)**

```
char str[] = "Try it out"; // length 10

// short *sPtr = (short *) str;  // old C-style cast, treat underlying data as short
short *sPtr = reinterpret_cast<short *>(str); // C++ style cast
// dangerous (short has 2Byte, therefore the length is now 5)
```

Operators new and delete - Details

- **`new T`** and **`new T[]`**
  - Allocates memory for objects, arrrays of the specified type
    - Free memory is newly reserved on the heap to hold the requested type
    - Constructs the object(s) (call of the constructor for non-primitive types)
    - Returns a pointer to the assigned address
  - If successful: not `nullptr`
    otherwise: Exception (**`std::bad_alloc`**)

- **`delete ptrT`** and **`delete[] ptrT`**
- Release of the memory once reserved.
    - Destroys the object(s) (calls destructor for non-primitive types)
    - Freed memory is added back to heap.
  - Any error when called on a pointer that was not allocated with **`new`** or **`new[]`**.

- **`new/delete`** or **`new[]/delete[]`** always use in pairs!

C-style memory management (malloc, free) - Details

- No constructors and destructors in C
  - are not called in `free` and `malloc`
- `malloc` returns a pointer of type `void*`.
  - **Memory is not initialized!**
  - Not type-safe
- `malloc` does not throw an exception in case of error
  - Program must test for null pointer
- `free` similar to `delete`
  - Pointer must have been previously reserved with `malloc`
  - Already released memory should not be released again (any result)
  - `free` on a zero pointer should be ok (ISO C)

`auto`, Function Overloading and Function Templates

`auto` – Deduced Variable Types

- You can have the compiler *deduce* the type of a variable when *initializing* it:

```
auto i = 7;           // i is of type int
auto d{3.3};          // d is of type double
auto x = foobar(); // x is of whatever type the function foobar() returns
```

- This is still **static typing**, the type is fixed at compile-time!
- This is handy when the **typename** is complex...
- ... but it also can make code less readable! Use with care ☺

Recommendation:

- Don't use `auto` for simple types like `int, bool, std::string` …

Function Overloading

```
double square(double const d) {
  return d * d;
}

uint32_t square(uint32_t const i) {
  return i * i;
}

uint64_t square(uint64_t const i) {
  return i * i;
}
...
uint64_t i = 7;
i = square(i); // picks third one
```

- You can have multiple functions with the same name, provided
  - they have a different number of parameters;
  - and/or the parameters have different types
  - (a different return type is not sufficient in C/C++!)
- This is called **function overloading**
- This is very useful when the functions operate differently on the input types, but...

Function Overloading

```cpp
float square(float const d) {
  return d * d;
}
double square(double const d) {
  return d * d;
}
uint8_t square(uint8_t const i) {
  return i * i;
}
uint16_t square(uint16_t const i) {
  return i * i;
}
```

```cpp
uint32_t square(uint32_t const i) {
  return i * i;
}

uint64_t square(uint64_t const i) {
  return i * i;
}

// and now for all integer types ...
```

- … you might never know which types might be needed in the future

Function Templates

```
template <typename T>
T square(T const n)
{
    return n * n;
}
// compiler generates no code
// for above template, until
// template is actually used:
int32_t i = 1;
i = square(i);
double d = 3.2;
d = square(d);
```

```
int32_t square(int32_t const i) {
  return i * i;
}
double square(double const d) {
  return d * d;
}
```

automatically instantiated

- The definition is called a **function template** (not "template function"**!**)
- On first use, the compiler generates / **instantiates** the overloads that we would have written otherwise
  → **reduces code duplication**
- it produces the same machine code as when we would have written the overloads ourselves
  → **zero-overhead, efficient**
- Only those overloads are generated that are actually used!

Function Templates vs. `auto`

```
template <typename T1, typename T2>
???? add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- But what type does `int32_t + double` return?
- How do you generalise that?

```
template <typename T1, typename T2>
auto add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- This is a good place to use **`auto`**!
- Only works for the return type...

```
auto add(auto const n1, auto const n2)
{
    return n1 + n2;
}
```

- C++20 introduces an abbreviated syntax with **auto**

Explicit Type Selection <type>

- When calling the function, you can also explicitly specify which arguments should be used for the template parameter(s)

```
template <typename T>
T min(const T& g, const T& d) {
    return ((g < d) ? g : d);
}

int i1,i2,i3;
double d1,d2,d3;

i3 = min(i1,i2);        // implicitly uses int version
d3 = min(d1,d2);        // implicitly uses double version

i3 = min<int>(i1,i2);   // explicitly use int version
d3 = min<int>(d1,d2);   // explicitly use int version
                        // (now the two double arguments
                        //  will be implicitly converted to int)
```

Explicit Instantiation of Function Templates

- Template with three parameters

```
template <class T1, class T2, class T3>
T1 add(const T2 &a, const T3 &b) {
  T1 res = a + b;
  return res;
}
```

- Usage and Instantiation
  - Function will be instantiated and compiled for the particular set of template parameters

- **Note:** Return type deduction is not possible in C++. One needs to specify it as template argument, others can be implicit.

```
int i1, i2;
short sum1 = add<short>(i1, i2); // add<short, int, int>
long sum2 = add<long>(i1, i2);   // add<long, int, int>
```

Behind the Scenes

- A given template function is **partially** checked for its syntax
- But: names, classes, attribute names etc. are only fixed after the instantiation
- Only when the function is called the template for the special type is fully instantiated and compiled completely

Testing of template libraries is difficult: Errors can become visible long after the implementation of the library just because e.g., a type was never used in any of the tests.

Integer as Template Parameter

```
template <int N, class T> void output(const T &_v) {
  for (int i = 0; i < N; i++) {
    cout << _v << " ";
  }
  cout << endl;
}

...

  int i = 3;
  double d = 4.44;
  output<3, int>(i);
  output<12>(i);
  output<5, double>(d);
  const int N = 3; // works, but int N = 3; would not work (dynamic)
  output<N>(i);
```

```
3 3 3
3 3 3 3 3 3 3 3 3 3
3 3
4.44 4.44 4.44 4.44
4.44
3 3 3
```

Overloaded Functions vs. Template

- **Template resolution:** if functions and template function exist with the same signature, the non-template function is evaluated first

```
inline float min(float g, float d) {
  cout << "non-template called" << endl;
  return ((g > d) ? g : d); // impl. wrong. returns max
}
template <typename T>
inline T min(T g, T d) {
  cout << "template called" << endl;
  return ((g < d) ? g : d); }
  ...
  int i1 = 3, i2 = 8;
  float f1 = 3, f2 = 8;
  int i3 = min(i1, i2);
  cout << "min(" << i1 << ", " << i2 << ") = " << i3 << endl;
  float f3 = min(f1, f2);
  cout << "min(" << f1 << ", " << f2 << ") = " << f3 << "-)" << endl;
```

```
template called
min(3, 8) = 3
non-template called
min(3, 8) = 8-)
```

Template specialization

Motivation: for some types it may be **necessary**, more **elegant** or more **efficient** to deviate slightly from the flow of the original template and provide a custom implementation.

Template specialization: special implementation for a template parameter set.

```
template <typename T> inline T min(T g, T d) { return ((g < d) ? g : d); }

template <> // specialization for char* needed, comparing pointers would be wrong!
inline const char *min<const char *>(const char *g, const char *d) {
  if (strcmp(g, d) < 0) {
    return g;
  }
  return d;
}
...
const char strA[] = "A little bigger";
const char strB[] = "A little smaller";
const char *strResult = min(strA, strB);
cout << "min(" << strA << ", " << strB << ") = " << strResult << endl;
```

Sequence Containers

- The STL provides powerful containers
- Solves many of the problems we saw using manual memory management, pointers and arrays

Sequence Containers – `std::vector`

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- We already learned about `std::string`, an STL container for character sequences

- In contrast to `std::string`, `std::vector` can hold arbitrary objects.

  e.g., `std::vector<int>`

  `std::vector<std::string>`

  We can specify the type in the `< >` brackets -> **template class** (later more…).

- Think of `std::vector` as a better C++ array class:
  - it can **grow** dynamically.
  - **memory** is automatically **managed**.

Sequence Containers – vector

- `vector` can be **constructed** using one of its many constructors:

```
vector<int> v;                   // default construction: empty vector
vector<int> v(100);              // empty vector but memory for 100 integer pre-allocated
vector<double> v(100, 0.0); // vector with 100 integers initialized to zero
vector<double> v2(v);            // construct as copy of v

string s{"Hello!"};
vector<char> v(s.begin(), s.end()); // construct from other container type: string
```

- Elements can be **appended** to the end of the vector with `.push_back()`

```
vector<int> v(10);          // empty vector with space for 10 elements
v.push_back(1234);          // vector now contains one element (1234)
```

- If the capacity (here 10) is reached the vector **automatically grows** on the next `push_back` and the existing data is moved/copied over to the new memory location.
- How resize happens is implementation dependent.

Example: vector growth

```
vector<int> v;
vector<int>::size_type s = v.capacity();

for (int i = 0; i < 100000; ++i)
{
  v.push_back(1);
  if (s != v.capacity()) // did it grow?
  {
    cout << v.size() << " / "
         << v.capacity() << "\n";
    s = v.capacity();
  }
}
```

```
1 / 1
2 / 2
3 / 4
5 / 8
9 / 16
17 / 32
33 / 64
65 / 128
129 / 256
257 / 512
513 / 1024
1025 / 2048
2049 / 4096
4097 / 8192
8193 / 16384
16385 / 32768
32769 / 65536
65537 / 131072
```

Sequence Containers – vector

- One can **access** elements using the `operator[]` similar to C-arrays

```
vector<int> v(100, 1); // vector with 100 ones
v[0]                   // access first element
v[v.size()-1]          // access last element
v.at(0)                // like v[0] but with bounds check
```

- Container like `vector` and the contained elements get **destructed at end of scope**.
- Automatically **frees** the allocated memory on the heap.

https://en.cppreference.com/w/cpp/container/vector

Sequence Containers – Overview

- Other sequence container in the STL:

Container	Informal summary
`std::array`	*Fast access, but fixed number of elements*
`std::vector`	*Fast access, efficient insertion/deletion only at end*
`std::string`	*Optimized for character types*
`std::deque`	Efficient insertion/deletion at beginning and end
`std::list`	Efficient insertion/deletion also in the middle, no []
`std::forward_list`	Efficient insertion/deletion also in the middle, no []

- STL containers do not provide member functions for operations that would be slow
  - e.g. `std::vector` provides `.push_back()`, but not `.push_front()`, while `std::deque` and `std::list` provide both.

Sequence Containers – Iteration

- For vector, we have several possibilities to iterate over its elements.

```
vector<int> v{1, 2, 3, 4, 5};

for (auto const &elem : v)                  // "range-based": simple, prefer if possible
  cout << elem << " ";

for (size_t i = 0; i < v.size(); ++i)  // via [], more verbose, more flexible
  cout << v[i] << " ";

for (auto it = v.begin(); it != v.end(); ++it) // via so-called iterators
  cout << *it << " ";
```

- but `operator[]` is not available for all containers, e.g. lists.
- `std::forward_list` doesn't even have `.size()` 😱

Sequence Containers – Iterators

Iterators are objects that allow iterating over sequence containers.

- Iterators and the related range-for loops work with all sequence container:

```
cout << "container contains elements:";

for (auto it = cont.begin(); it != cont.end(); ++it) // "iterator-based"
  cout << *it << " ";

for (auto const &element : cont)                       // ”range-based”
  cout << element << " ";
```

- All STL containers return an iterator pointing to the first element when calling `.begin()`
- This iterator can be incremented **++** to move to the next element, or dereferenced ***** to retrieve the actual element in O(1)
- It can also be compared against the special iterator retrieved by calling `.end()`
  The iterator from end() acts as a sentinel for the end of the container and must not be dereferenced as it points **past-the-end** of the container!
- Iterators are light-weight objects and cheap to copy
- **iterators != pointers** (later more…)

Sequence Containers

- **`std::array`**

```
std::array<double, 2> df{3.1, 2.3};
std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
```

- **Size fixed** at compile-time; specified via second template argument. **Stack**.
- Provides *RandomAccessIterator*
- Use instead of built-in array in all serious projects, i.e. it has no drawbacks over built-in arrays.

- **`std::vector`**

```
std::vector<double> df{3.1, 2.3};
std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
df.push_back(2.2);  // append value
df.resize(42);      // resize
```

- "Dynamic" array. Can grow. **Heap.**
- Append values in O(1)
- Other inserts O(n)
- Fast access and no size overhead
- **If you are unsure, probably the right choice of container.**

Sequence Containers

- **`std::basic_string`**

```
std::string<char> str{"ABC"};
//== std::string
std::cout << str[0]; // prints 'A'
df[1] = 32.0;         // assigns value
```

- Like `std::vector<char>`, only supports character types
- Slightly slower access
- Optimizations for small strings
- Convenience functions for input/output

- **`std::deque`**

```
std::deque<double> df{3.1, 2.3};
std::cout << df[0]; // prints 3.1
df[1] = 32.0;       // assigns value
df.push_back(2.2);  // append value
df.push_front(1.1); // prepend value
```

- Like `vector`, but:
- Supports prepend in O(1)
- Faster `resizes`
- High overhead for `size`
- Slightly slower access

Sequence Containers

- **`std::list`**

```
std::list<double> df{3.1, 2.3};
std::cout << *df.begin(); // prints 3.1
df.insert(it, 2.2);        // insert value
df.push_back(2.2);         // append value
```

- A doubly-linked list
- Fast inserts/deletes anywhere
- No random access! ~~[]~~
- 128bit size overhead per element
- Provides `BidirectionalIterator`

- **`std::forward_list`**

```
std::forward_list<double> df{3.1, 2.3};

std::cout << *df.begin(); // prints 3.1
df.insert_after(it, 2.2); // append
```

- A singly-linked list
- Fast inserts/deletes anywhere
- No random access, no `.size()`!
- 64bit size overhead per element
- Provides `ForwardIterator`

Sequence Containers – Overview

Container	++it	--it	[]	←	↓	→	Space overhead
`std::array`	✓	✓	✓				0
`std::vector`	✓	✓	✓			✓	64bit per container
`std::basic_string`	✓	✓	✓			✓	Small per container
`std::deque`	✓	✓	✓	✓		✓	Large per container
`std::list`	✓	✓		✓	✓	✓	**128bit per element (!)**
`std::forward_list`	✓			✓	✓	✓	**64bit per element (!)**

- STL containers do not provide member functions for operations that would be slow
- e.g. `std::vector` provides `.push_back()`, but not `.push_front()`, while `std::deque` and `std::list` provide both.
- ← - insert front, ↓ - insert anywhere, → - insert at end

Quiz

- What container do you choose if you know you will need to add an unknown amount of new elements periodically, but never delete any?

- When would you prefer a `std::deque` over `std::vector`?

- When would you prefer `std::forward_list` over `std::list`?

Rule-of-thumb: don't use `std::deque`, `std::list` or `std::forward_list` of built-in types, because the overhead is just too high.

- `std::vector` **is efficient and works amazingly well in many situations**. They are like arrays under the hood with added convenience (e.g., being able to grow).

Note: if you observe performance problems with `std::vector` is often a result of frequent, expensive reallocation – can be fixed by preallocating enough space: `v.reserve(…)`

Tuples and Tie

Tuples

- **Tuples** are convenient to wrap multiple variables

```
std::tuple<std::string, std::string> cosmonaut{"Sigmund", "Jaehn"};
```

- Provides better encapsulations, makes interfaces more readable

```
void match(std::tuple<std::string, std::string> const &person0,
           std::tuple<std::string, std::string> const &person1);

// is more readable than:
void match(std::string const &person0_first_name,
           std::string const &person0_last_name,
           std::string const &person1_first_name,
           std::string const &person1_last_name);
```

Tuples – Access

```cpp
std::tuple<std::string, std::string> cosmonaut{"Sigmund", "Jaehn"};

// via get
std::cout << "First name: " << std::get<0>(cosmonaut) << ' '
          << "Last name: " << std::get<1>(cosmonaut) << '\n';

// via structured bindings
auto &[f, l] = cosmonaut; // std::string & f = std::get<0>(cosmonaut)…
std::cout << f << ' ' << l << '\n';
```

You can access tuple elements via

1. **`std::get<NUMBER>()`**
2. **`std::get<TYPE>()`** (only if the types are unique!)
3. **structured bindings**, i.e. declaring a set of variables of **`auto`** type (or **`auto &`** or **`auto const &`**...) and assigning the tuple.

Tuples (and Pairs)

- Can be used to return multiple values

```
auto make_tuple(size_t const i, std::string const &s) {
  return std::tuple{i, s}; // return type is std::tuple<size_t, std::string>
}
```

- There is also **`std::pair`**, a tuple of size 2, but it only exists for historical reasons – if possible, use **`std::tuple`** nowadays!

Tuples

Alternative way to bundle variables: **`struct`**

```
// our custom “Name” type
struct Name {
    std::string firstName;
    std::string lastName;
};
Name cosmonaut{"Sigmund", "Jaehn"};
cosomonaut.lastName = "Freud";
```

- Disadvantage of tuples:
  - Tuple members are *unnamed,* i.e. you have to know that first string refers to the first name and the second one to the last name.

- Advantages of tuples:
  - Use **`tuples`** for multiple return values, works together with **`std::tie`**
  - Usual operators are already defined on a per-element basis, i.e.
    `std::tuple<float, int>{1.1, 3} == std::tuple<float, int>{2.2, 3}`
    compares first the first element, then the second …

Tuples and Tie

```
  size_t const i = 7;
  std::string s{"foo"};
  std::tuple tup0{i, s};
  // == std::tuple<size_t, std::string>
```

- Creating tuples from existing variables copies the values and discards const.

```
  auto tup2 = std::tie(i, s);
  // == std::tuple<size_t const &,
  // std::string &>

  std::tie(a.i, a.s) < std::tie(b.i, b.s); //
compare two structs a and b (first i, then s)
```

- There is a convenience function for creating tuple of references: **std::tie()**

```
auto fillTuple() {
  return std::tuple( 3, "text",  4.3);
}
...
  int a;
  double b;
  std::tie( a, std::ignore, b) = fillTuple();   // a
== 3 and b == 4.3
```

- Fill existing variables
  - **`std::ignore`** to skip one return value

Extra: Arrays & Pointers 2

- Recap calling functions
- Passing pointers to functions

Parameter transfer - "pass by pointer"

- pass by value
  - The value of the argument is assigned to the formal parameter (copy)
  - The parameter is used as a local variable
  - no effect on the calling arguments
  - **Disadvantage**: potentially expensive copy

```
void f(int a)
```

- pass by reference
  - Passed argument can be uses like a local variable (no copy)
  - Values of the calling argument can be modified
  - **Disadvantage**: hard to see from call site that variable will be changed

```
void f(int &a)
```

- “pass by pointer"
  - Instead of a reference, you can also simply pass the address
  - Dereferencing allows changing the value
  - **Disadvantage**: dereferencing necessary
  - (Note: technically we are passing a pointer by value)

```
void f(int *a)
```

Call "by pointer"

```
void swapPointer(int *_a, int *_b) {
  int tmp = *_a;
  *_a = *_b;
  *_b = tmp;
}

int main() {
  int a{3}, b{2};
  cout << "before swap: " << a << " " << b << endl;
  swapPointer(a, b);
  cout << "after swap: " << a << " " << b << endl;
}
```

```
before swap: 3 2
after swap: 2 3
```

Functions and Arrays

- C-style arrays can be passed to functions
  - pass "by pointer" means: the **address** is passed by value
  - **size** must be passed as well

```
void printArray(int *arr, int n) {
  for (int i = 0; i < n; ++i) cout << arr[i] << " ";
  cout << endl;
}
int main() {
  int *myArr = new int[5]{0,1,2,3,4};
  printArray(myArr, 5);
  delete[] myArr;
}
```

```
0 1 2 3 4
```

- Arrays can be returned as pointer: `int * myFunction()`

Summary

Summary

- Memory, Virtual Address Space
- Arrays, Pointer, Strings
- `auto`
- Function Overloading
- Function Templates, Spezialization
- `std::vector`, Sequence Containers
- `std::tuple`
- Call by
  - Value
  - References
  - "Pointer" = call by value with an address