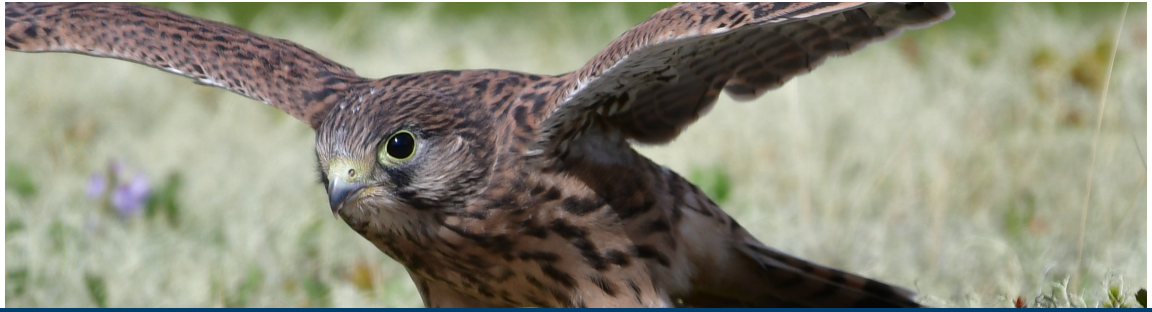


PROGRAMMING IN C++

Jülich Supercomputing Centre

8 – 12 May 2023 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Day 4

How do I write a generic function to calculate $f(a, x, y) = ax + y$?

- (A)

```
1  auto fma(int a, int x, int y) { return a * x + y; }
2  auto fma(float a, float x, float y) { return a * x + y; }
3  auto fma(double a, double x, double y) { return a * x + y; }
4  // and so on for all the types I can think of
```

- (B) `auto fma(auto a, auto x, auto y) { return a * x + y; }`
- (C) `template <class T> auto fma(T a, T x, T y) { return a * x + y; }`

How do I ensure that that generic function is only considered for 3 floating point inputs of the same type?

- (A)

```
1  // Please, only use this with floating point inputs!!!
2  template <class T>
3  auto fma(T a, T x, T y) { return a * x + y; }
```

- (B)

```
1  template <class T> requires std::floating_point<T>
2  auto fma(T a, T x, T y) { return a * x + y; }
```

- (C) The two above are equivalent, because `requires` is a fancy annotation for the programmer, not actual code
- (D) It can not be done

STANDARD TEMPLATE LIBRARY

- Utilities
  - `pair`, `tuple`
  - `optional`, `variant`, `any`
  - `bitset`, `bit`, `endian`, `bit_cast`
  - `type_traits`, `concepts`, safe integral comparisons
  - `initializer_list`
  - `system`, `atexit`
  - `bind`, `placeholders`, `apply`, `invoke` ...
- Date and Time
- Random numbers
- Smart pointers
- Filesystem
- Regular expressions
- Containers, `span`
- Algorithms, ranges
- Iterators
- Strings and string view
- Fast character conversions
- Multi-threading, atomic types
- Parallel algorithms
- Text formatting

UNIQUE POINTER

```
1  // examples/uniqueptr.cc
2  auto main() -> int
3  {
4      auto u1 = std::make_unique<MyStruct>(1);
5      //auto u2 = u1; //won't compile
6      auto u3 = std::move(u1);
7      std::cout << "Data value for u3 is u3->v1 = " << u3->v1 <<'\n';
8      auto u4 = std::make_unique<MyStruct[]>(4);
9  }
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Exclusive access to resource
- Can not be copied (deleted copy constructor and assignment operator)
- Data ownership can be transferred with `std::move`
- Can create single instances as well as arrays through `make_unique`

SHARED POINTER

```cpp
// examples/sharedptr.cc
auto main() -> int
{
    auto u1 = std::make_shared<MyStruct>(1);
    std::shared_ptr<MyStruct> u2 = u1; // Copy is ok
    std::shared_ptr<MyStruct> u3 = std::move(u1);
    std::cout << "Reference count of u3 is "
              << u3.use_count() << '\n';
}
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Can share resource with other shared/weak pointers
- Can be copy assigned/constructed
- Maintains a reference count `ptr.use_count()`

WEAK POINTER

```
1  // examples/weakptr.cc
2  auto main() -> int
3  {
4      auto s1 = std::make_shared<MyStruct>(1);
5      std::weak_ptr<MyStruct> w1(s1);
6      std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
7      std::shared_ptr<MyStruct> s3(s1);
8      std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
9  }
```

- Does not own resource
- Can "kind of" share data with shared pointers, but does not change reference count

Exercise 1.1: uniqueptr.cc, sharedptr.cc

Read the 3 smart pointer example files, and try to understand the output. Observe when the constructors and destructors for the data objects are being called.

MEMORY MANAGEMENT ERRORS

```
1  auto somefunc(inputpars) -> outputtype
2  {
3      auto* heapblock = new double[1024];
4
5      // calculations
6      // calculations
7      // calculations
8
9      return res;
10     // Oops! Forgot to delete heapblock!
11 }
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.

MEMORY MANAGEMENT ERRORS

```
1  auto somefunc(inputpars) -> outputtype
2  {
3      auto* heapblock = new double[1024];
4
5      // calculations
6      // calculations
7      // calculations
8
9      delete [] heapblock;
10     return res;
11 }
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code

MEMORY MANAGEMENT ERRORS

```
1  auto somefunc(inputpars) -> outputtype
2  {
3      auto* heapblock = new double[1024];
4
5      // calculations
6      // throw an exception!
7      // calculations
8
9      delete [] heapblock;
10     return res;
11 }
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`

MEMORY MANAGEMENT ERRORS

```
1  auto somefunc(inputpars) -> outputtype
2  {
3      auto heapblock =
4          std::make_unique<double[]>(1024);
5
6      // calculations
7      // throw an exception!
8      // => unique_ptr cleans up
9
10     return res;
11     // unique_ptr cleans up
12 }
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`
- Use RAII for resource management instead.

MEMORY MANAGEMENT ERRORS

```
1  auto somefunc(inputpars) -> outputtype
2  {
3      auto heapblock =
4          std::make_unique<double[]>(1024);
5
6      // calculations
7      // throw an exception!
8      // => unique_ptr cleans up
9
10     return res;
11     // unique_ptr cleans up
12 }
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`
- Use RAII for resource management instead.
- Delegate explicit life time management of heap resources to smart pointers, e.g., `std::unique_ptr`

DANGERS OF DANGLING POINTERS AND REFERENCES

```
1  {
2      int* ptr = nullptr;
3      if (something) {
4          auto i = std::stoi(argv[1]);
5          ptr = &i;
6          std::cout << "ptr is pointing at "
7              << *ptr << "\n";
8      }
9      // ptr still in scope, but i isn't!
10     std::cout << *ptr << "\n";
11     // dangling --> dereference -->
12     // undefined behaviour!
13 }
```

- Other forms of memory errors exist, and are harder to eliminate

DANGERS OF DANGLING POINTERS AND REFERENCES

```
1  {
2      int* ptr = nullptr;
3      if (something) {
4          auto i = std::stoi(argv[1]);
5          ptr = &i;
6          std::cout << "ptr is pointing at "
7              << *ptr << "\n";
8      }
9      // ptr still in scope, but i isn't!
10     std::cout << *ptr << "\n";
11     // dangling --> dereference -->
12     // undefined behaviour!
13 }
```

- Other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.

DANGERS OF DANGLING POINTERS AND REFERENCES

```
1  auto calc(double inp) -> double&
2  {
3      auto loc = inp * inp;
4      // Returning ref to local:
5      return loc; // Bad idea!
6  }
7  void elsewhere()
8  {
9      auto&& res = calc(4);
10     std::cout << res << "\n";
11 }
```

- Other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.
- If we return a reference from a function, we must make sure, it is not a reference to a temporary object.

DANGERS OF DANGLING POINTERS AND REFERENCES

```
1  {
2      std::vector v{1, 2, 3};
3      auto& vstart = v.front();
4      v.push_back(4); // may invalidate refs
5      v.push_back(5);
6      v.push_back(6);
7      v.push_back(7);
8      std::cout << vstart << "\n";
9  }
```

- Other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.
- If we return a reference from a function, we must make sure, it is not a reference to a temporary object.
- If we store references to heap object, there is always the danger that operations on the owning entity will invalidate the reference

DANGLING –> DEREFERENCE –> UNDEFINED BEHAVIOUR

Example 0.1:

The folder `examples/dangling_pr` contains examples of the 3 kinds of memory bugs mentioned in this section. Study them, and check what, if any, errors or warnings the compiler generates for them. Try compiling with `-Wall Wextra`. Run them and examine the results. Try compiling with `-fsanitize=address`.

AVOID DANGLING POINTERS AND REFERENCES

- Ensure that pointers and references do not outlive the referenced objects
- Prefer short lived non-owning pointers
- Do not return references to temporary objects
- Avoid storing references to objects on the heap

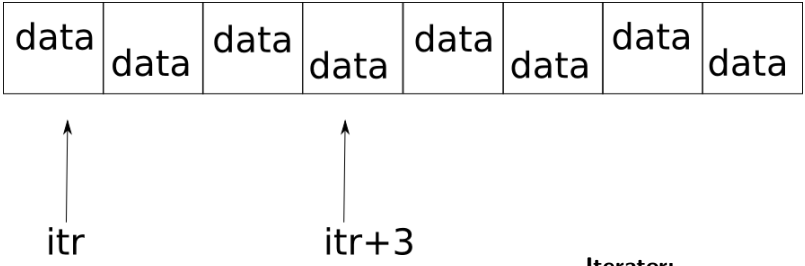
STL CONTAINERS

```
1  using namespace std;
2  int sz;
3  cin >> sz;
4  // vector<double> B(sz,3.0); // <- C++17 ->
5  vector B(sz, 3.0); // C++17 ->
6  vector c{1, 2, 3, 4};
7  c.push_back(5); // append
8  list l{1, 2, 3, 4};
9  l.insert(find(l.begin(),l.end(),2), 14);
10 // insert in the middle
11 map<string, int> rank;
12 rank["Sirius"] = 1;
13 rank["Canopus"] = 2;
14 for (auto el : B) cout << el << "\n";
15 for (auto el : l) cout << el << "\n";
16 for (auto el : rank)
17     cout << el.first <<" -> "
18          << el.second << "\n";
```

- Form: `container<datatype>`. Include file `containername`

- Many easy-to-use sequence types available in the STL
  - `vector` : Dynamic array type
  - `list` : Linked list
  - `map` : Sorted associative container
  - `unordered_map` : Hash table
- Not always necessary to explicitly state the element type. If there is an initialiser, element type can be inferred.
- Store a fixed kind of elements, determined at the point of declaration.
- They can grow at run time (except `std::array`)
- Whenever possible, prefer `array` or `vector`

VECTOR: DYNAMIC ARRAY CLASS TEMPLATE



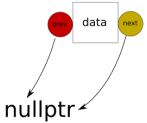
- Element type is a template parameter
- Consecutive elements in memory
- Can be accessed using an "iterator"

Iterator:

- Iterators are classes which pretend to be pointers
- They can be dereferenced with overloaded `*` and `->` operators to retrieve an element
- They can be moved forward or backward using overloaded `++` and `--` operators
- They can be compared for equality or inequality

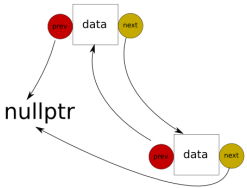
A LINKED LIST

A linked list is a collection of connected nodes. Each node has some data, and one or two pointers to other nodes. They are the "next" and "previous" nodes in the linked list. When "next" or "previous" does not exist, the pointer is set to `nullptr`



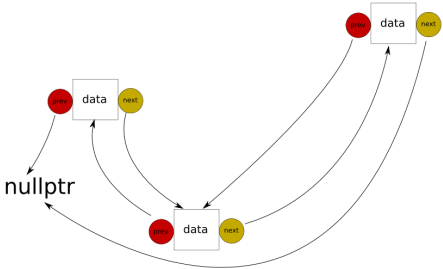
A LINKED LIST

When a new element is added to the end of a list, its "previous" pointer is set to the previous end of chain, and it becomes the target of the "next" pointer of the previous end.



A LINKED LIST

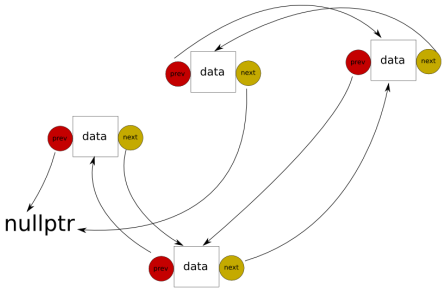
New elements can be added to the front or back of the list with only a few pointers needing rearrangement.



JÜLICH Forschungszentrum

A LINKED LIST

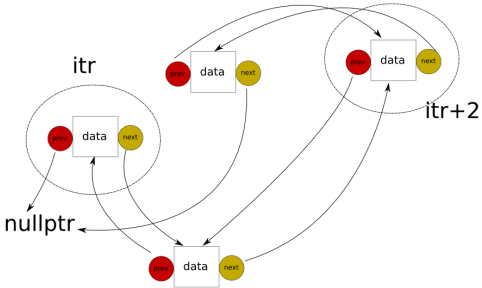
Any element in the list can be reached, if one kept track of the beginning or end of the list, and followed the "next" and "previous" pointers.



JÜLICH Forschungszentrum

A LINKED LIST

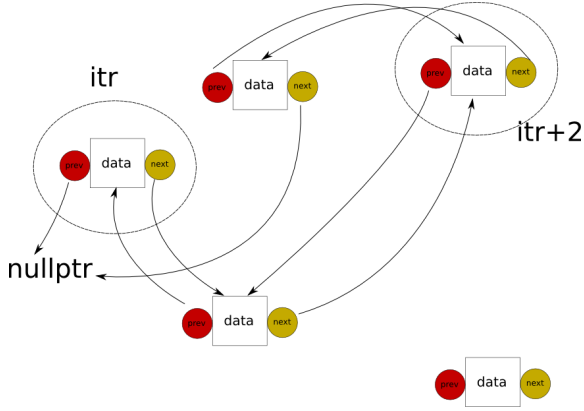
A concept of an "iterator" can be devised, where the `++` and `--` operators move to the next and previous nodes.



JÜLICH Forschungszentrum

A LINKED LIST

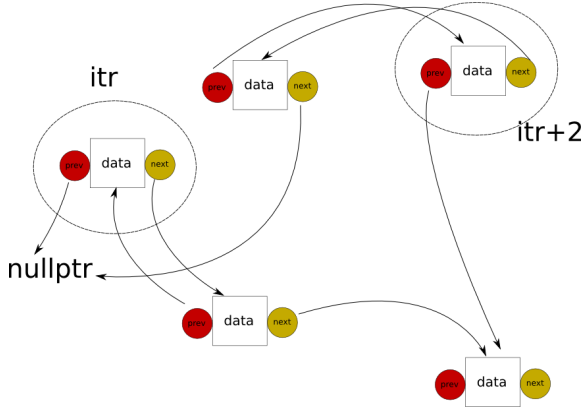
Inserting a new element in the middle of the list does not require moving the existing nodes in memory.



JÜLICH Forschungszentrum

A LINKED LIST

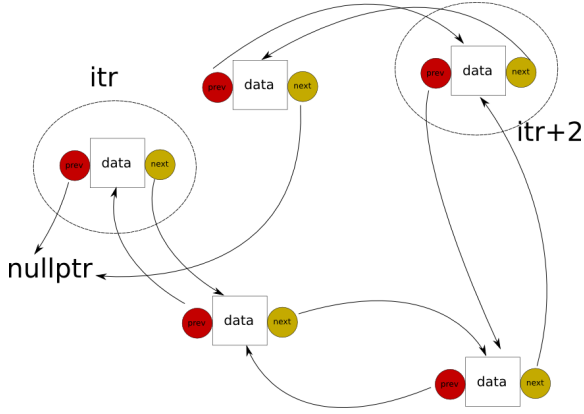
Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.



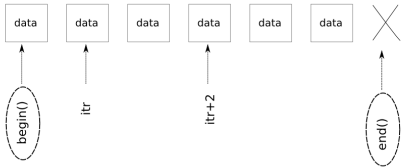
JÜLICH Forschungszentrum

A LINKED LIST

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.



GENERIC "CONTAINERS"



- Generic data holding constructions
- Can be accessed through a suitably designed "iterator"
- The data type does not affect the design $\Longrightarrow$ template

- Similarity of interface is by design
- With a standard container `c` of type `C`, it's always possible to use `std::begin(c)` to access the start and `std::end(c)` to access the end
- `std::begin()` and `std::end()` return `C::iterator` or `C::const_iterator` depending on whether `c` is const qualified.
- `std::cbegin(c)` and `std::cend(c)` return `C::const_iterator` types irrespective of whether `c` is a const
- Similarly, `std::size(c)` always returns the size of the container, i.e., the number of elements it contains

STL CONTAINERS

- `std::vector<>` : dynamic arrays
- `std::list<>` : linked lists
- `std::queue<>` : queue
- `std::deque<>` : double ended queue
- `std::map<A,B>` : associative container

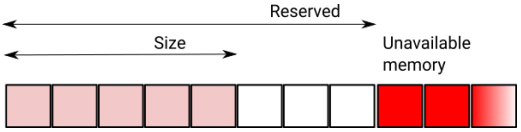
- Structures to organise data
- Include file names correspond to class names
- All of them provide corresponding iterator classes
- If `iter` is an iterator, `*iter` is data.
- All of them provide member functions like `begin()`, `end()`, `size()`, initializer list constructors, deduction rules for class template argument deduction

```
1       list L{1, 2, 3, 4, 5}; // std::list<int>, initialized to 1, 2, 3, 4, 5
2       auto pp = partition(begin(L), end(L), [](auto i){ return i % 3 == 0; });
3       decltype(L) M;
4       M.splice(end(M), L, begin(L), pp);
```

USING STD::VECTOR

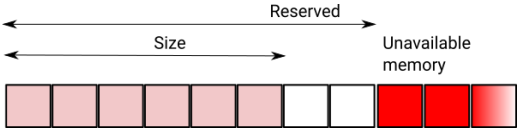
- `vector<int> v(10);` makes a dynamic array of 10 integers, `vector v(10, 0.)` creates a vector of 10 `doubles` initialized to 0, `vector v{1u, 2u, 3u}` creates a vector of `unsigned int` with values 1, 2 and 3.
- Efficient indexing operator `[]`, for unchecked element access
- `v.at(i)` provides range checked access. An exception is thrown if `at(i)` is called with an out-of-range `i`
- `std::vector<std::list<userinfo>> vu(10) ;` array of 10 linked lists.
- Supports `push_back` and insert operations, but sometimes has to relocate the all the elements because of one `push_back` operation (next slide)

STD::VECTOR



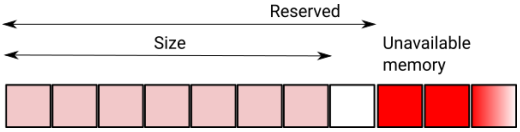
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



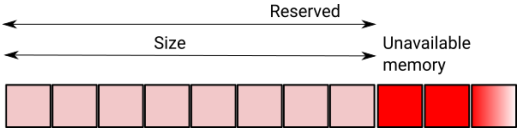
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



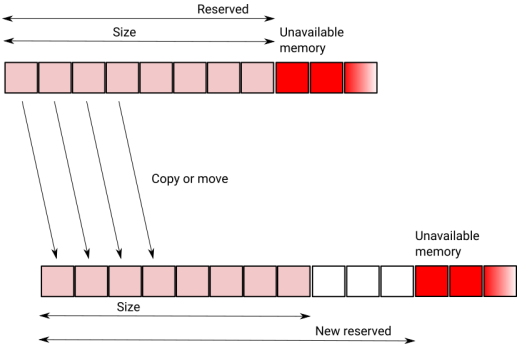
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



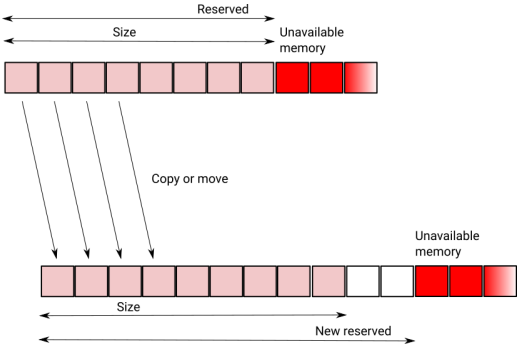
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



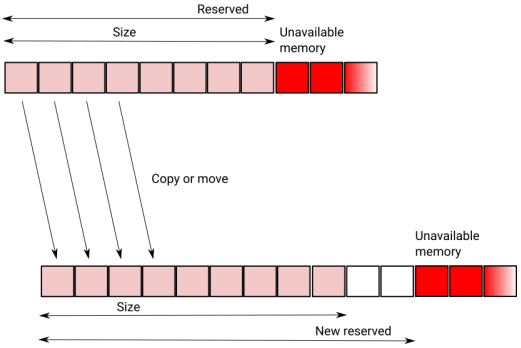
- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When `push_back` is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

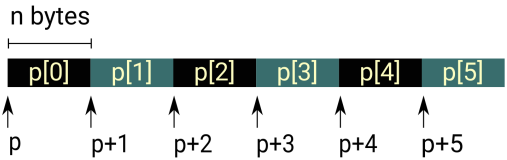
Exercise 1.2:

Construct a list and a vector of 3 elements of the `Vbose` class from your earlier exercise. Add new elements one by one and pause to examine the output. This aspect was also demostrated in the notebook `CtorDtorDemo.ipynb` .

STD::ARRAY : ARRAYS WITH FIXED COMPILE TIME CONSTANT SIZE

- `std::array<T,N>` is a fixed length array of size N holding elements of type `T`
- It implements functions like `begin()` and `end()` and is therefore usable with STL algorithms like `transform`, `generate` etc.
- The array size is a template parameter, and hence a compile time constant.
- `std::array<std::string,7> week{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};`

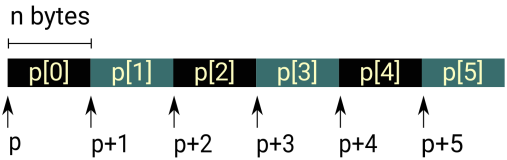
ARRAYS




```
1  double A[10]; // Built-in or C-style array
2  int sz;
3  std::cin >> sz;
4  int M[sz]; // Not allowed!
5  #include <array>
6  ...
7  std::array<double,10> A; // On stack
8  // Like a built-in array, but obeys
9  // C++ standard library conventions.
10 for (size_t i = 0; i < A.size(); ++i) {
11     P *= A[i];
12 }
13 std::vector<double> B(sz,3.0);
```

- Sequence of N objects stored consecutively in memory, with no gaps
- If `p` is a pointer to the first object of such a sequence, `p+1`, `p+2` etc, will point to the subsequent elements. Elements of the sequence can therefore be accessed as `*(p+0)`, `*(p+1)`, `*(p+2)` ... another notation for that is `p[0]`, `p[1]` ...

ARRAYS




```
1  double A[10]; // Built-in or C-style array
2  int sz;
3  std::cin >> sz;
4  int M[sz]; // Not allowed!
5  #include <array>
6  ...
7  std::array<double,10> A; // On stack
8  // Like a built-in array, but obeys
9  // C++ standard library conventions.
10 for (size_t i = 0; i < A.size(); ++i) {
11     P *= A[i];
12 }
13 std::vector<double> B(sz,3.0);
```

- Built-in or "C-style" arrays consist of blocks of memory large enough to hold a fixed number of elements. The array, thought of as a pointer, points to the first element in the sequence. The elements are stored consecutively, but the number of elements is never stored anywhere
- `std::array<type,size>` is a compile-time fixed length array obeying STL conventions. The size is available through a function, although it does not have to be stored with the array data!
- `std::array<type,size>` retains its "personality" (does not decay into a pointer) when used as input to function or when received as the output from a function. This should be your default choice when you need fixed length arrays.

ASSOCIATIVE CONTAINERS: STD::MAP

```
1  std::map<std::string, int> flsize;
2  flsize["S.dat"]=123164;
3  flsize["D.dat"]=423222;
4  flsize["A.dat"]=1024;
```

- Think of it as a special kind of "vector" where you can have things other than integers as indices.
- Template arguments specify the key and data types
- Could be thought of as a container storing (key,value) pairs : $\{("S.dat", 123164), ("D.dat", 423222), ("A.dat", 1024)\}$
- The less than comparison operation must be defined on the key type
- Implemented as a tree, which keeps its elements sorted

A WORD COUNTER PROGRAM

Exercise 1.3:

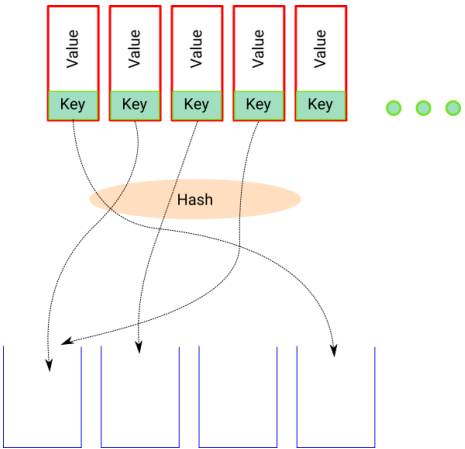
Fake exercise: Write a program that counts all different words in a text file and prints the statistics.

```
1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4  #include <string>
5  #include <map>
6  auto main(int argc, char *argv[]) -> int
7  {
8      std::ifstream fin(argv[1]);
9      std::map<std::string, unsigned> freq;
10     std::string s;
11     while (fin >> s) freq[s]++;
12     for (auto [key, val] : freq)
13         cout << std::setw(12) << key
14              << std::setw(4)  << ':'
15              << std::setw(12) << val <<"\n";
16 }
```

A quick histogram!

- `std::map<string, unsigned>` is a container which stores an integer, for each unique `std::string` key.
- The iterator for `std::map` "points to" a `pair<key,value>`

STD::UNORDERED_MAP AND STD::UNORDERED_SET



Unordered map

- Like `std::map<k,v>` and `std::set<v>`, but do not sort the elements
- Internally, these are hash tables, providing faster element access than `std::map` and `std::set`
- Additional template arguments to specify hash functions

STL ALGORITHMS

```
1   ...
2   std::vector<YourClass> vc(inp.size());
3   std::copy(inp.begin(), inp.end(), vc.begin());
4   //Copy contents of list to a vector
5   auto pos = std::find(vc.begin(), vc.end(), elm);
6   //Find an element in vc which equals elm
7   std::sort(vc.begin(),vc.end());
8   //Sort the vector vc. The operator "<"
9   //must be defined
10  ...
11  std::transform(inp.begin(), inp.end(), out.begin(), rotate);
12  //apply rotate() to each input element,
13  //and store results in output sequence
```

The similarity of the interface, e.g. `begin()` , `end()` etc., among STL containers allows generic algorithms to be written as template functions, performing common tasks on collections

STL ALGORITHMS

- Typically, the algorithms in the namespace `std` accept one or more ranges as (start, stop) pairs, some other inputs which may include callable objects
- New algorithms were introduced in C++20 in the namespace `std::ranges`, where the input ranges are given as single objects rather than iterator pairs. Think

```
std::ranges::for_each(v, [](auto&& elem){ std::cout << elem << "\n"; })
```

rather than

```
std::for_each(v.begin(), v.end(), [](auto&& elem){ std::cout << elem << "\n";
```

Exercise 1.4:

- The standard library provides a large number of template functions to work with containers
- Look them up in `www.cplusplus.com` or `en.cppreference.com`
- Use the suitable STL algorithms to generate successive permutations of the vector

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
         [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
         [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object
- `std::ranges::sort(range)` and `std::ranges::sort(range, lt)` are corresponding versions using a range as an argument instead of a pair of iterators

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  ranges::sort(v, [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STD::TRANSFORM

- `std::transform(begin_1 , end_1, begin_res, unary_function);`
- `std::transform(begin_1 , end_1, begin_2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The "result" container can be (one of the) input container(s)

```
1  std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
2  std::list L1(v.size(), 0), L2(v.size(), 0);
3  std::transform(v.begin(), v.end(), L1.begin(), sin);
4  std::transform(v.begin(), v.end(), L1.begin(), L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

STD::RANGES::TRANSFORM

- `std::ranges::transform(range1, begin_res, unary_function);`
- `std::transform(range1, range2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The "result" container can be (one of the) input container(s)

```
1  std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
2  std::list L1(v.size(), 0), L2(v.size(), 0);
3  std::ranges::transform(v, L1.begin(), sin);
4  std::ranges::transform(v, L1, L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

ALL_OF, ANY_OF, NONE_OF

```
1  auto valid(std::string name) -> bool
2  {
3    return all_of(name.begin(),name.end(),
4        [](char c) { return (isalpha(c)) || isspace(c); });
5  }
```

- `std::all_of(begin_ , end_ , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::any_of(begin_ , end_ , condition)` checks if any single element in a given range satisfies `condition`
- `std::none_of(begin_ , end_ , condition)` returns true if not a single element in a given range satisfies `condition`

ALL_OF, ANY_OF, NONE_OF

```
1  auto valid(std::string name) -> bool
2  {
3    return all_of(name,
4        [](char c) { return (isalpha(c)) || isspace(c); });
5  }
```

- `std::ranges::all_of(range , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::ranges::any_of(range , condition)` checks if any single element in a given range satisfies `condition`
- `std::ranges::none_of(range , condition)` returns true if not a single element in a given range satisfies `condition`

ALGORITHMS

```
1  vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
2  vector<int>  x, y, z, m;
3  if (is_sorted(begin(v), end(v)))
4      cout << "The sequence is sorted in the increasing order.\n";
5  reverse(v.begin(), v.end());
6  rotate(v.begin(), v.begin() + 3, v.end());
7  sort(begin(v), end(v));
8  merge(v.begin(), v.end(), w.begin(), w.end(), back_inserter(m));
9  set_union(v.begin(), v.end(), w.begin(), w.end(), back_inserter(x));
10 set_intersection(w.begin(), w.end(), v.begin(), v.end(), back_inserter(y));
11 set_symmetric_difference(v.begin(), v.end(), w.begin(), w.end(), back_inserter(z));
12 if (is_permutation(z.begin(), z.end(), v.begin(), v.end())) // do something
```

Exercise 1.5:

A whole lot of operations available for sequence types. The file `seqops.cc` contains the operations shown here. Alternatively, (or, in addition,) use the jupyter notebook `intro_algorithms.ipynb` to examine the effects of the algorithms on sequences. Explore!

ALGORITHMS

- `for_each(start, end, operation)` : As it sounds
- `find(start, end, what)` : returns the location of the looked for value, "end" if not found
- `find_if(start, end, condition)` , find the first element satisfying a condition
- `copy(start1, end1, start2)` : As it sounds
- `copy_if(start1, end1, start2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(start1, end1, start2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `start2`

CONSTRAINED ALGORITHMS (RANGES)

- `for_each(range, operation)` : As it sounds
- `find(range, what)` : returns the location of the looked for value, "end" if not found
- `find_if(range, condition)` , find the first element satisfying a condition
- `copy(range1, iterator2)` : As it sounds
- `copy_if(range1, iterator2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(range1, iterator2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `iterator2`

ALGORITHMS

```
1  vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
2  vector<int>  x, y, z, m;
3  if (is_sorted(v))
4      cout << "The sequence is sorted in the increasing order.\n";
5  reverse(v);
6  rotate(v, v.begin() + 3);
7  sort(v);
8  merge(v, w, back_inserter(m));
9  set_union(v, w, back_inserter(x));
10 set_intersection(w, v, back_inserter(y));
11 set_symmetric_difference(v, w, back_inserter(z));
12 if (is_permutation(zv)) // do something
```

Exercise 1.6:

The file `seqops_range.cc` contains the operations shown here. Explore by making modifications. Try GCC 10.0+ compiler for this.

CHRONO: THE TIME LIBRARY

- `namespace std::chrono` defines many time related functions and classes (include file: `chrono`)
- `system_clock` : System clock
- `steady_clock` : Steady monotonic clock
- `high_resolution_clock` : To the precision of your computer's clock
- `steady_clock::now()` : nanoseconds since 1.1.1970
- `duration<double>` : Abstraction for a time duration. Uses `std::ratio<>` internally

Exercise 1.7: chrono_demo.cc

THE TIME LIBRARY

```
1  // examples/chrono_demo.cc
2  #include <iostream>
3  #include <chrono>
4  #include <vector>
5  #include <algorithm>
6  #include <ranges>
7  bool is_prime(unsigned n);
8  auto main() -> int
9  {
10     using namespace std::chrono;
11     namespace sr = std::ranges;
12     namespace sv = std::views;
13     std::vector<unsigned> primes;
14     auto t = steady_clock::now();
15     sr::copy(sv::iota(0UL, 10000UL) | sv::filter(is_prime), back_inserter(primes));
16     std::cout << "Primes till 10000 are ... " << '\n';
17     for (unsigned i : primes) std::cout << i << '\n';
18     auto d = steady_clock::now() - t;
19     std::cout<<"Prime search took " << duration<double>(d).count() << " seconds\n";
20 }
```

CALENDAR AND DATES WITH `STD::CHRONO`

```
1  auto current_year() -> std::chrono::year
2  {
3      using namespace std::chrono;
4      year_month_day date { floor<days>(system_clock::now()) };
5      return date.year();
6  }
7  auto main(int argc, char* argv[]) -> int
8  {
9      using namespace std::chrono;
10     using namespace std::chrono_literals;
11     auto Y0 { current_year() };
12     auto Y1 = Y0 + years{100};
13     if (argc > 1) Y1 = year{std::stoi(argv[1])};
14     if (argc > 2) Y0 = year{std::stoi(argv[2])};
15     if (Y1 < Y0) std::swap(Y1, Y0);
16
17     for (auto y = Y0; y < Y1; ++y) {
18         auto d = y / February / Sunday[5];
19         if (d.ok())
20             std::cout << static_cast<int>(y) << "\n";
21     }
22 }
```

CALENDAR...

Example 0.2:

The programs `examples/feb.cc` and `examples/advent.cc` demonstrate the use of the calendar facilities of the C++ standard library. Familiarize yourself with them.

RANDOM NUMBER GENERATION

- Convenient, flexible, powerful random number library providing high quality (pseudo-)random numbers in standard C++ without any external libraries.
- Include `random`. Namespace `std::random`

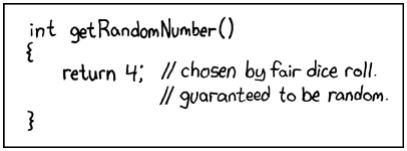


Figure: Source XKCD: `http://xkcd.com`

RANDOM NUMBER GENERATION

- Share a common structure
- Uniform random generator engine with (hopefully) well tested properties
- Distribution generator which adapts its input to a required distribution

$$p(n) = \frac{m^n e^{-m}}{n!}$$

Random distribution

Randomness engine

glue with lambda

```
1  auto gen = [
2      engine = std::mt19937_64{},
3      dist=std::poisson_distribution<>{8.5}
4             ]() mutable {
5      return dist(engine);
6  };
7  r = gen();
```

```
1  std::mt19937_64 engine;
2  std::poisson_distribution<> dist{8.5};
3  auto gen = [&dist, &engine] {
4      return dist(engine);
5  };
6  r = gen();
7  // if engine or dist are required elsewhere
```

RANDOM NUMBER GENERATORS

```
1  #include <random>
2  #include <iostream>
3  #include <map>
4  auto main() -> int
5  {
6      auto gen = [ dist=std::poisson_distribution<> {8.5}, engine=std::mt19937_64{} ]
7          () mutable { return dist(engine); };
8      std::map<int,unsigned> H;
9      for (auto i = 0UL; i < 5000000UL; ++i) H[gen()]++;
10     for (auto [i, fi] : H) std::cout << i << " " << fi << '\n';
11 }
```

- `std::mt19937_64` is a 64 bit implementation of Mersenne Twister 19937
- The template `std::poisson_distribution` is a functional implementing the Poission distribution

RANDOM NUMBER GENERATORS

```
1  std::normal_distribution<> G{3.5, 1.2}; // Gaussian mu = 3.5, sig = 1.2
2  std::uniform_real_distribution<> U{3.141, 6.282};
3  std::binomial_distribution<> B{13};
4  std::discrete_distribution<> dist{0.3, 0.2, 0.2, 0.1, 0.1, 0.1};
5  // The following is an engine like std::mt19937, but is non-deterministic
6  std::random_device seed; // int i = seed() will be a random integer
```

- Lots of useful distributions available in the standard
- With one or two lines of code, it is possible to create a high quality generator with good properties and the desired distribution
- `std::random_device` is a non-deterministic random number generator.
  - It is good for setting seeds for the used random number engine
  - It is slower than the pseudo-random number generators

RANDOM NUMBER GENERATOR: EXERCISES

Exercise 1.8:

Make a program to generate normally distributed random numbers with user specified mean and variance, and make a histogram to demonstrate that the correct distribution is produced. Start from `exercises/normal_distribution.cc`.

Exercise 1.9:

Make a program to implement a "biased die", i.e., with user specified non-uniform probability for different faces. You will need `std::discrete_distribution<>` Start from `exercises/weighted_die.cc`.

EXERCISES

Exercise 1.10:

For a real valued random variable X with normal distribution of a given mean μ and standard deviation σ, calculate the following quantity:

$$K[X] = \frac{\left\langle (X-\mu)^4 \right\rangle}{(\langle (X-\mu)^2 \rangle)^2}$$

Fill in the random number generation parts of the program `exercises/K.cc`. Run the program a few times varying the mean and standard deviation. What do you observe about the quantity in the equation above ?

Exercise 1.11: Probabilities with playing cards

The program `examples/cards_problem.cc` demonstrates many topics discussed during this course. It has a `constexpr` funtion to create a fixed length array to store results, several standard library containers and algorithms as well as the use of the random number machinery for a Monte Carlo simulation. It has extensive comments explaining the use of various features. Read the code and identify the different techniques used, and run it to solve a probability question regarding playing cards.

STD::OPTIONAL

- `std::optional<T>` manages an optional value of type `T`, which may or may not be present
- Another way to handle errors during computations to determine a value of some kind
- If the `optional` object has a value, the value resides in the object, i.e., the `optional` type does not do any dynamic memory allocation of its own
- The operators `*` and `->` are given for convenience, so that we can pretend we are dealing with a pointer type when using an `optional`
- If converted to a `bool`, we get **true** if there is a value, **false** otherwise
- Default initialisation as well as initialisation with `nullopt_t` create `optional` objects without value.

```
1  auto solve_quadratic(double a, double b,
2                       double c)
3  {
4      using namespace std;
5      optional<pair<double, double>> solution;
6      auto D = b * b - 4 * a * c;
7      if (D >= 0) {
8          auto q = -0.5 * ( b +
9                            copysign(sqrt(D), b) );
10         solution = make_pair(q / a, c / q );
11     }
12     return solution;
13 }
```

Exercise 1.12:

`examples/opt_qsolve.cc` is a small program demonstrating the use of `std::optional`.

STD::VARIANT : A TYPE SAFE UNION

- A `union` is a special kind of class where all the members occupy the same bytes in memory

```
union sameplace { size_t ulong; double real; };
static_assert(sizeof(sameplace) ==
              sizeof(double));
sameplace s;
s.ulong = 0UL;
s.real = 1.0;
cout << s.ulong << "\n";
```

- We can access the elements of a `union` the same way as a `struct` (above).
- Since both members occupy the same bytes, changes to one can affect the other
- If the union contains, e.g., `std::string`, such overriding of bytes would be dangerous.

- `std::variant` is a type safe `union`.
- Unlike the `union`, we don't get to name the different members. The different "members" can be accessed through functions like `std::get<int>(V)`, i.e., we can use the types to select the stored type. We also don't need to say what we are assigning to, since that can be deduced from the type of the object on the right of the `=`
- A `variant` knows what type is currently stored, and calls the destructors etc. when we assign something that would change the stored type

```
variant<double, int, long, string> V;
V = "let's assign a string";
V = 3.141;
// call string destructor and store a double
```

STD::VARIANT : A TYPE SAFE UNION

- A variant type stores one value of any one of a few pre-specified alternatives. To create a `variant` with an integer, a long, a string and a boolean, we would write

```
std::variant<int, long, string, bool> V;
```

- A variant can be assigned a value of any one of its contained types. The variant then remembers the value and the type of the value.

```
V = "0118 999 881 99 9119 725  3"s;
assert(std::holds_alternative<string>(V));
```

- The member function `index()` tells us the zero based index of the currently held type in the list of alternatives for the variant

```
assert(V.index() == 2);
```

- Since the type of the contained object can be changed by an assignment at run time, the variant can not simply have a function `get()` to return the contained value. We have to specify the type of value we want to read as a template argument:

```
cout << get<string>(V);
```

- Unlike the union, we can't store one type and read another

```
V = "0118 999 881 99 9119 725  3"s;
auto num = get<int>(V); //throws exception!
```

- There is also a non throwing version of the accessor:

```
if (auto iptr = get_if<int>(&V); iptr) {
// use iptr as pointer to int value
// Does not get here because get_if<int>
// returns a nullptr in this case.
}
```

STD::VARIANT : A TYPE SAFE UNION

```
1  using member_t = variant<int, long, string, bool>;
2  vector<member_t> pop{true, 91, "Monday"s};
3  for (auto & el : pop) {
4      if (auto iptr = get_if<int>(&el)) {
5      // *iptr is the int value in the variant el
6      } else if (auto lptr = get_if<long>(&el)) {
7      // *lptr is the long value in el
8      } else if (auto sptr = get_if<string>(&el)) {
9      // *sptr is the string value in el
10     }
11 }
```

- Variants can be made to model members of heterogeneous collections, much like pointers to base class in a class hierarchy. The difference is, we can even use built in type like `int`, `double` etc. in a variant based heterogeneous container, because it does not need a class hierarchy!

- Easiest way to model polymorphic behaviour is using a chain of `if ... else if ... else` statements using the `get_if<T>(&v)` function for the different types `T` in the `variant`. `get_if<T>(&v)` returns a valid `T *` if the variant `v` currently holds type `T`. Otherwise it returns `nullptr`.

Exercise 1.13:

The two example programs `examples/variant_0.cc` and `examples/variant_1.cc` demonstrate basic variant usage, such as assignment of values of different types, performing actions based on the content type.

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable fo handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  struct my_action {
2      auto operator()(int i) { // ... }
3      auto operator()(double x) { //  }
4  };
5  // ...
6  std::visit(my_action{}, V);
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable fo handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  std::visit([](auto upkd) {
2      if constexpr (is_same_v<int, decltype(upkd)>) {
3          // handle int input
4      } else if constexpr (is_same_v<double, decltype(upkd)>) {
5          // handle double input
6      }
7    }, V
8  );
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable fo handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
1  template <class ... Ts> struct stapler : Ts ... { using Ts::operator()... ; };
2  template <class ... Ts> stapler(Ts ...) -> stapler<Ts...>;
3  std::visit(stapler{
4      [](int i) { /* handle int input */ },
5      [](double d) { /* handle double */ }
6      }, V
7  );
```

USING VARIANTS WITH STD::VISITOR

Exercise 1.14:

Example programs `examples/variant_2.cc`, `examples/variant_3.cc` and `examples/variant_4.cc` demonstrate the use of `std::visit` to dispatch different actions depending on the type of the currently held value in a variant. They parallel the approaches in the 3 boxes in the previous slide.

STD::ANY : A TYPESAFE CONTAINER FOR SINGLE VALUES

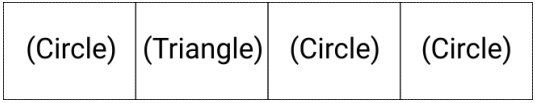
- A variable of type `std::any` can store 1 value of any type
- Simply by assigning a new value, the contained object is replaced with another of the new type. The variable of type `std::any` is like a box, whose type remains unchanged as the content is swapped. The contained object is indirectly accessed, leading to some overhead.

Exercise 1.15:

`examples/any_demo.cc` demonstrates basic usage of `std::any`.

```
1  any var = 1;
2  cout << "Reading int after storing int ... "
3       << any_cast<int>(var) << "\n"; // That works
4  try {
5      cout << "Reading float after storing an int ... "
6           << any_cast<float>(var) << "\n";
7           // This doesn't
8  } catch (const exception & err) {
9      cout << "Float cast after storing int failed. "
10          << "Error : " << err.what() << "\n";
11 }
12 var = "Europa"s;
13 map<string, any> config;
14 config["max_frequency_ghz"] = 3.3;
15 config["memory_MB"] = 16384;
16 config["fingerprint_reader"] = true;
```

SEQUENCES OF POLYMORPHIC OBJECTS



Exercise 1.16:

Sequences of objects with polymorphic behaviour is a frequently occuring programming problem. We have seen one example before, with a vector of `unique_ptr<Shape>`, filled with newly created instances of types inheritted from `Shape`, such as `Circle`, `Triangle` etc. The problem can be solved in many alternative ways. `examples/polymorphic` contains 4 subdirectories with different approaches to the geometric object example. (i) Inherittance with virtual functions (ii) `std::variant` with visitors (iii) Using `std.:any` (iv) Custom type erasure.