

# Pointers, References & Iterators

---

Danilo Ardagna

07/10/2024

Politecnico di Milano

[danilo.ardagna@polimi.it](mailto:danilo.ardagna@polimi.it)



POLITECNICO  
DI MILANO

# Content

- Pointers and references
- *auto* specifier
- *const* Qualifier
- Iterators

# Pointers and references

- A reference is not an object. Hence, we may not have a pointer to a reference



# A reference to *const* may refer to an object that is not *const*

- A reference to *const* restricts only what we can do through that reference
- Binding a reference to *const* to an object says nothing about whether the underlying object itself is *const*
- Because the underlying object might be non *const*, it might be changed by other means

```
int i = 42;  
int &r1=i;           // r1 bound to i  
const int &r2 = i; // r2 also bound to i; but cannot be used to  
                  // change i  
r1 = 0;            // r1 is not const; i is now 0  
r2 = 0;            // error: r2 is a reference to const
```

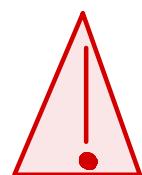
# References cannot be stored in a vector



```
std::vector<int> hello; // OK
```

```
std::vector<int &> hello; // Error! Pointer to reference  
// is illegal!
```

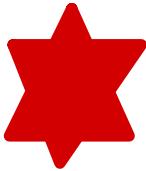
- Containers values **must be Assignable**
- References are *non-assignable* (you can only initialize them once when they are declared, and you cannot make them reference something else later)
- Other non-assignable types are also not allowed as components of containers, e.g. `vector<const int>` is not allowed





# Pointers to Pointers

- There are no limits to how many type modifiers can be applied to a declarator
- When there is more than one modifier, they combine in ways that are logical but not always obvious
- A pointer is an object in memory, so like any object it has an address. Therefore, we can store the address of a pointer in another pointer
- We indicate each pointer level by its own \*. That is:
  - **we write \*\* for a pointer to a pointer**
  - \*\*\* for a pointer to a pointer to a pointer, and so on



# Pointers to Pointers Example

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to an int
```



```
cout << "The value of ival\n"
<< "direct value: " << ival << "\n" << "indirect
value: " << *pi << "\n" << "doubly indirect value: "
<< **ppi << endl;
```

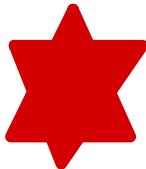
For // computing.

A real example of something we need, MPI!!

- **MPI\_Init( int\* argc, char\*\*\* argv)**

# *auto* Specifier

---



# The *auto* type specifier

- It is not uncommon to want to store the value of an expression in a variable declaration
  - To declare the variable, we have to know the type of that expression
- Since C++ 11, we can let the compiler figure out the type for us by using the `auto` type specifier
- Unlike type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce the type from the initializer
  - A variable that uses `auto` as its type specifier **must** have an initializer

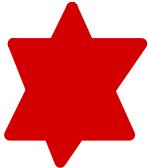




# The *auto* type specifier

Example

```
// the type of item is deduced from the type of the result of  
// adding val1 and val2  
auto item = val1 + val2; // item initialized to the  
// result of val1 + val2
```



# Traversing a vector

Use case !

```
vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
for (auto &i : v) // for each element in v (note: i is a  
// reference)
```

```
i *= i; // square the element value
```

I'm really changing the vector!

```
for (auto i : v) // for each element in v  
cout << i << " "; // print the element
```

```
cout << endl;
```

Here I would only be changing a copy.



# Useful use of references

- A range-for loop (**assume v is a vector of strings**):

- `for (string s : v) cout << s << "\n";` // s is a copy of all v[i]
- `for (string& s : v) cout << s << "\n";` // no copy
- `for (const string& s : v) cout << s << "\n";` // and we don't modify v

*"Read-only"*

- A range-for loop (**assume v is a vector with elements of any type**):

- `for (auto e : v) cout << e << "\n";` // e is a copy of all v[i]
- `for (auto& e : v) cout << e << "\n";` // no copy
- `for (const auto& e : v) cout << e << "\n";` // and we don't modify v



# Multiple dimensional arrays – From Slide deck 1

```
int ia[3][4]; // array of size 3; each element is an array of  
                // ints of size 4  
int ia[3][4] = {{0, 1, 2, 3}, // three elements; each element is an array of size 4  
                 {4, 5, 6, 7}, // initializers for the row indexed by 1  
                 {8, 9, 10, 11}} // initializers for the row indexed by 2  
};
```

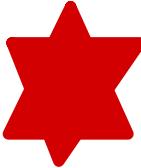
// equivalent initialization without the optional nested braces for each row

```
1 int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```



## Accessing multiple dimensional arrays – From Slide deck 1

```
constexpr size_t rowCnt = 3, colCnt = 4;  
  
int ia[rowCnt][colCnt]; // 12 uninitialized elements  
                        // for each row  
for (size_t i = 0; i != rowCnt; ++i) {  
    // for each column within the row  
    for (size_t j = 0; j != colCnt; ++j) {  
        // assign the element's positional index as its value  
        ia[i][j] = i * colCnt + j; }  
}
```



# Range-for for accessing multiple dimensional arrays

```
constexpr size_t rowCnt = 3, colCnt = 4;  
  
int ia[rowCnt][colCnt]; // array of size 3; each element is an  
// array of ints of size 4  
  
size_t cnt = 0;  
for (auto &row : ia) // for every element in the outer array  
    for (auto &col : row) { // for every element in the inner array  
        col = cnt; // give this element the next value  
        ++cnt; // increment cnt  
    }
```

# Iterators

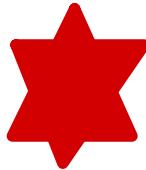
---



# Iterators

??

- We can use subscripts to access the characters of a string or the elements in a vector
- There is a more general mechanism — **iterators** — that we can use for the same purpose
- The library defines **several** other kinds of **containers**. All of the library containers have **iterators**, but only a **few** of them **support the subscript operator**
- You can think to an **iterator as a pointer to access any container**
- If we use **iterators instead of subscripts, we can change easily the container type without changing our code**
- Like pointers, iterators give us **indirect access** to an object
  - An iterator can be used to fetch an element
  - Iterators have also operations to move from one element to another
  - An iterator may be valid or invalid



# Using Iterators

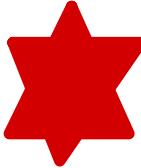
- Unlike pointers, we do not use the address-of operator to obtain an iterator
- Instead, types that have iterators have members that return iterators: **begin()** and **end()**
  - The begin member returns an iterator that denotes the first element, if there is one
  - The iterator returned by end is an iterator positioned “one past the end” of the associated container



# Using Iterators

```
// the compiler determines the type of b and e  
// b denotes the first element and e denotes one past the  
// last element in v  
auto b = v.begin(), e = v.end(); // b and e have the same  
// type
```

- If the container is empty, the iterators returned by begin and end are equal, they are both off-the-end iterators



# Using Iterators

```
string s("some string");
```

```
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}
```

// process characters in s until we run out of characters or

// we hit a whitespace

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
```

Equivalent to  
string::iterator it = ....

Important use != instead of <  
If we change the type it might not work

Means go to  
the next  
element

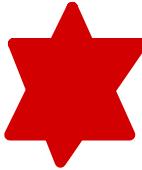


# Standard container iterator operations

*iter	Returns a reference to the element denoted by the iterator <i>iter</i>
iter->memb	Dereferences <i>iter</i> and fetches the member <i>memb</i> from the underlying element
(*iter).memb	
++iter	Increments <i>iter</i> to refer to the next element in the container
--iter	Decrements <i>iter</i> to refer to the previous element in the container
iter1==iter2	Compares two iterators. Two iterators are equal if they denote the same element or if they are the <b>off-the-end</b> iterator for <b>the same container</b>
iter1!=iter2	



## Operations supported by **vector** and **string** iterators (Only by vector and strings! Be careful!!)



iter + n	Adding (subtracting) an integral value $n$ from the iterator $iter$ yields an iterator $n$ elements forward of backward than $iter$ within the container
iter - n	
iter1 +=n	Assign to $iter1$ the value of adding (subtracting) $n$ to $iter1$
iter1 -=n	
iter1-iter2	Compute the number of elements between $iter1$ and $iter2$
>,>=,<,<=	One iterator is less than another if it denotes an element that appears in the container before the one referred to



# Iterator types

- The library types that have iterators define types named `iterator` and `const_iterator` that represent actual iterator types

```
vector<int>::iterator it; // it can read and write vector<int>  
                         // elements
```

```
string::iterator it2; // it2 can read and write characters in a  
                     // string
```

```
vector<int>::const_iterator it3; // it3 can read but not write  
                                // elements
```

```
string::const_iterator it4; // it4 can read but not write  
                          // characters
```

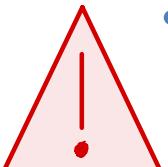


# The cbegin and cend operations

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin(); // it1 has type vector<int>::iterator
auto it2 = cv.begin(); // it2 has type vector<int>::const_iterator
```

*r&w*      *only*

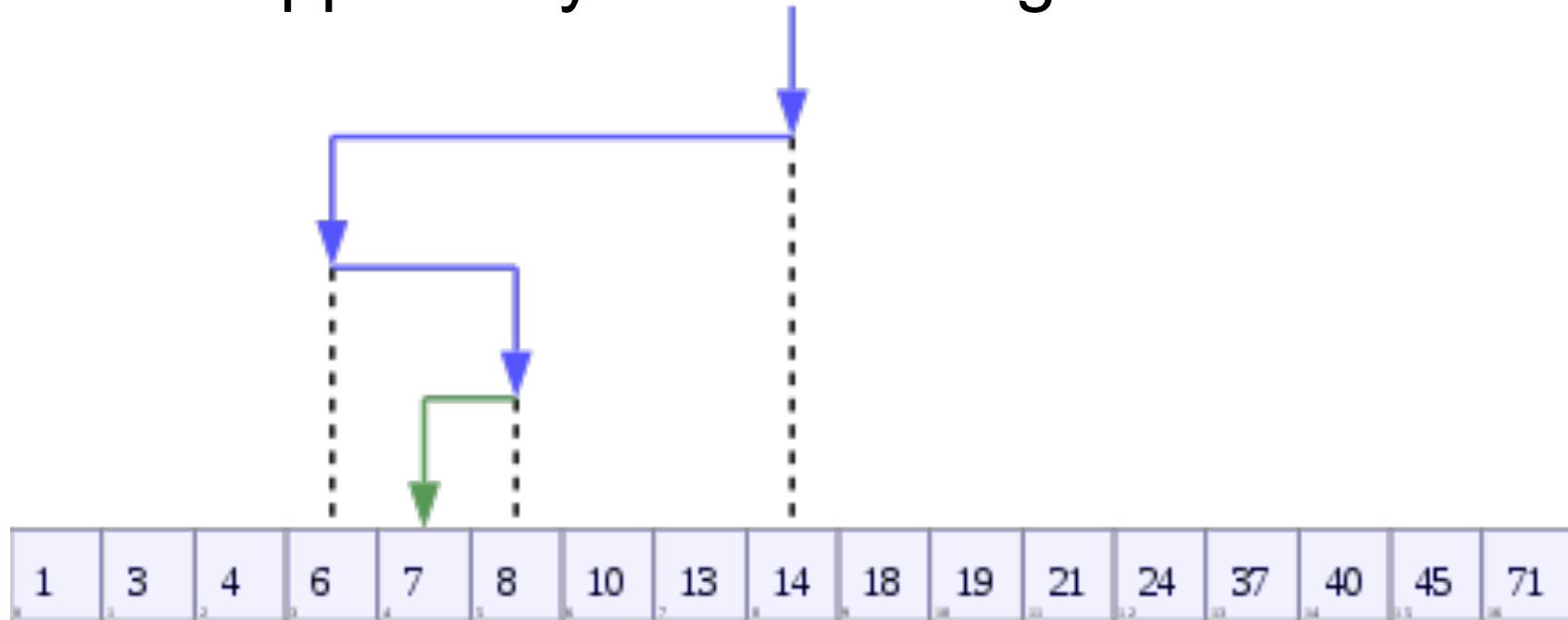
- It is usually best to use a const type (such as `const_iterator`) when we need to read but do not need to write to an object
- To let us ask specifically for the `const_iterator` type, since C++ 11, two new functions named `cbegin()` and `cend()` are available

 auto it3 = v.cbegin(); // it3 has type vector<int>::const\_iterator

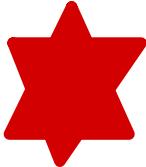
# Binary Search

Recall on how binary search works:

- Sorted arrays
- Looking for 7
- What happens if you are looking for 5?



# Arithmetic operations on iterators – Binary search



```

vector<string> text
// initialize text
// text must be sorted
sort(text.begin(), text.end());
// beg and end will denote the range we're searching
auto beg = text.cbegin(), end = text.cend();
auto mid = text.cbegin() + (end - beg)/2; // original midpoint
s=....
// while there are still elements to look at and we haven't yet // found s
while (mid != end && *mid != s) {
    if (s < *mid) // is the element we want in the first half?
        end = mid; // if so, adjust the range to ignore the
                    // second half
    else // the element we want is in the second half
        beg = mid + 1; // start looking with the ele
    mid = beg + (end - beg)/2; // new midpoint
}
if (mid!= text.cend() && s == *mid)
    cout <<"Yes I found "<<s<<" in text"<<endl;
else
    cout <<"Sorry I cannot find "<<s<<" in text"<<endl;

```

WE ARE LOOKING FOR  $s$  HERE

still lots to  
look at.

*we didn't find  $s$ .*

if ( $s < *mid$ ) // is the element we want in the first half?  
 $end = mid$ ; // if so, adjust the range to ignore the  
// second half

else // the element we want is in the second half  
 $beg = mid + 1$ ; // start looking with the ele

$mid = beg + (end - beg)/2$ ; // new midpoint

Complexity ?

Here we rely on **&& short-circuit**, no other way to write this while (if-else below) correctly

DEMO

# Last year quiz 1

# References

- Lippman Chapters 2 and 3

# Advanced readings

---

# Pointers and references

- A reference is not an object. Hence, we may not have a pointer to a reference
- However, because a pointer is an object, we can define a reference to a pointer

```
int i = 42;  
int *p;           // p is a pointer to int  
int *&r = p;    // r is a reference to the pointer p  
r = &i;          // r refers to a pointer; assigning &i to r makes  
                 // p point to i  
*r = 0;          // dereferencing r yields i, the object to which p points;  
                 // changes i to 0
```

# Pointers and Arrays

- In C++ pointers and arrays are closely intertwined. In particular, when we use an array the compiler ordinarily converts the array to a pointer
  - in most places when we use an array, the compiler automatically substitutes a pointer to the first element

```
string nums[] = {"one", "two", "three"}; // array of strings  
string *p = &nums[0]; // p points to the first element in nums
```

```
string *p2 = nums; // equivalent to p2 = &nums[0]
```

# Pointers and Arrays

- There are various implications of the fact that operations on arrays are often really operations on pointers
  - When we use an array as an initializer for a variable defined using `auto`, the deduced type is a pointer, not an array

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
auto ia2(ia); // ia2 is an int* that points to the first element in ia
ia2 = 42; // error: ia2 is a pointer, and we can't assign an int to a
           // pointer
```

# Pointers arithmetic

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr; // equivalent to int *ip = &arr[0]
int *ip2 = ip + 4; // ip2 points to arr[4] (which is 5!), the last element in arr
```

```
// ok: arr is converted to a pointer to its first element; p points one
// past the end of arr
int *p = arr + sz; // use caution -- do not dereference!
int *p2 = arr + 10; // error: arr has only 5 elements; p2 has
                    // undefined value
```

# Pointers arithmetic

- Although we can compute an off-the-end pointer, doing so is error-prone. To make it easier and safer to use pointers, C++ 11 library includes two functions, begin and end

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
```

```
int *beg = begin(arr); // beg points to the first element in arr
```

```
int *last = end(arr); // pointer just past the last element in arr
```

```
for (int *b = beg; b != last; ++b)
```

```
    cout << *b << endl; // print the elements in arr
```

# Pointers arithmetic

- Subtracting two pointers gives us the distance between those pointers. The pointers must point to elements in the same array

```
auto n = end(arr) - begin(arr); // compute the number of  
// elements in arr
```

# Pointers arithmetic

- We can use the subscript operator on any pointer, as long as that pointer points to an element (or one past the last element) in an array

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *p = &ia[2]; // p points to the element indexed by 2
int j = p[1];    // p[1] is equivalent to *(p + 1),
                  // p[1] is the same element as ia[3]
int k = p[-2];   // p[-2] is the same element as ia[0]
```

- This last example points out an important **difference** between **arrays** and **library types** that have subscript operators
  - The **library types** force the index used with a subscript to be an **unsigned value**. The **built-in subscript** does **not**
  - The **index** used with the **built-in subscript** operator can be a **negative** value. Of course, the resulting address must point to an element in (or one past the end of) the array to which the original pointer points

# Credits

- Bjarne Stroustrup. [www.stroustrup.com/Programming](http://www.stroustrup.com/Programming)