Programming in C++

Session 5 – Pointers and Arrays
Iterators

Dr Christos Kloukinas

City, UoL
https://staff.city.ac.uk/c.kloukinas/cpp
(slides originally produced by Dr Ross Paterson)

Introduction

- Pointers and arrays: vestiges of C that survive in C++ (Savitch 10.1; Stroustrup 5.1–3; Horstmann 9.7).
- *Iterators*: objects that provide sequential access to the elements of containers (Savitch 17.3 and 19.2; Stroustrup 19.2).
- The interface offered by STL iterators is based on an analogy with pointers and arrays.
- The STL provides a number of generic functions that operate on iterators. In the STL, these are called *algorithms*.

Pointers and arrays

- C's arrays, pointers and pointer arithmetic survive in C++.
- Arrays are mostly superseded by vectors.
- C/C++ pointers support arithmetic, but this is little used in C++.
- Many uses of pointers are superseded by references, but they still have their uses:
  - Subtype polymorphism.
  - Dynamically allocated objects (sessions 8 and 9).
  - Dynamic data structures.
  - Legacy interfaces.
  - Accessing hardware directly.

Pointers in C and C++

- Pointer variables are declared with `*`

  ```
  int *ip;
  ```

  This does not initialize the pointer.
- The address of a piece of storage, obtained with `&`, is a pointer:

  ```
  int i;
  ip = &i;
  ```

- Pointers are dereferenced with `*`

  ```
  *ip = *ip + 3;
  ```

  In general, `*` and `&` are inverses.

`&` the ***address-of*** operator
`*` the ***dereference*** operator

Note: Beware of multiple variable definitions!

```
int *ip1, ip2; // ip1 is a pointer, ip2 is an int
```

Why? `*ip1` is an `int` – so is `ip2`. The `*` operator binds with the name, not the type.

Pointers vs References

Given the definition of two integer variables: `int i = 3, j = 4;`

	References	***Pointers***
Declaration	`int &ref = i;`	`int *pointr = &i;`
Reading the integer	`cout << ref;`	`cout << *pointr;`
Assigning the integer	`ref = 5;`	`*pointr = 5;`
Using another integer	***N/A***	`pointr = &j;`

- `pointr` is an ***actual variable***, allocated somewhere in memory.
- A `ref` is more like a `const` pointer (`int * const r = &i;`), with an easier interface (no `*` and `&`), and the additional assertion that `r != nullptr`.

```
        _______  (On a 16 bit computer:)
1024 |   3     | i, ref
1040 |   4     | j
1056 | 1024    | pointr (holds the address of i)
1072 | ...     | (other - possibly garbage)
1088 | ...     | (other - possibly garbage)
```

Undefined pointers

- The storage pointed to by a pointer may become undefined. There will be no warning from the compiler or runtime system:

```
int *p;
{
    int i = 5;
    p = &i;
}       // i ceases to exist
*p = 3; // undefined behaviour
```

  Like a telephone number that has gone out of use – calling it doesn't reach anyone (or may reach another person).

It is the ***programmer's responsibility*** to ensure that the pointer points at something ***valid*** whenever it is dereferenced.

- BTW, local variable pointers are ***not initialized*** (no basic type is).
  ⇒ `p`'s initial value is ***garbage*** .

Null pointers

- The value `0` in pointer types is distinct from any address.

  `int *ip = 0;`

  cf. **null** in Java.
- Since C++11 one should use ***`nullptr`*** instead of `0` – avoid using ***`NULL`*** (comes from C).
- Pointers that are global variables are initialized to `nullptr`
- Again, pointers that are local variables are not initialized.

More pointers

- The following declaration

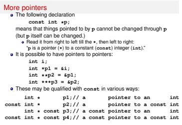
  `const int *p;`

  means that things pointed to by `p` cannot be changed through `p` (but `p` itself can be changed.)
  - Read it from right to left till the `*`, then left to right:
    "`p` is a pointer (`*`) to a constant (`const`) integer (`int`)."
- It is possible to have pointers to pointers:

```
int i;
int *p1 = &i;
int **p2 = &p1;
int ***p3 = &p2;
```

- These may be qualified with `const` in various ways:

```
      int *       p1;// a       pointer to an       int
const int *       p2;// a       pointer to a const  int
      int * const p3;// a const pointer to an       int
const int * const p4;// a const pointer to a const  int
```



More pointers

- The following declaration

```
const int *p;
```

means that things pointed to by **p** cannot be changed through **p** (but **p** itself can be changed.)
  - Read it from right to left till the *, then left to right: "**p** is a pointer (*) to a constant (**const**) integer (**int**)."
- It is possible to have pointers to pointers:

```
int i;
int *p1 = &i;
int **p2 = &p1;
int ***p3 = &p2;
```

- These may be qualified with **const** in various ways:

```
      int *       p1;// a       pointer to an      int
const int *       p2;// a       pointer to a const int
      int * const p3;// a const pointer to an      int
const int * const p4;// a const pointer to a const int
```

```
      int *       *       p1;// a pointer to a pointer to an int
const int *       *       p2;// ???
      int * const *       p3;// ???
      int *       * const p4;// ???
const int * const *       p5;// ???
const int *       * const p6;// ???
      int * const * const p7;// ???
const int * const * const p8;// ???
```

Pointers to objects

Given a class

```
class point {
public:
    int x, y;
    point (int xx, int yy) : x(xx), y(yy) {}
};
```

We can refer to members as follows:

```
point my_point(2, 3);
point *p = &my_point;
cout << (*p).x << '\n';
```

or equivalently as

```
cout << p->x << '\n';
```

and similarly for member functions.

Arrays

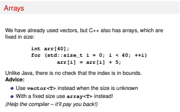
We have already used vectors, but C++ also has arrays, which are fixed in size:

```
int arr[40];
for (std::size_t i = 0; i < 40; ++i)
        arr[i] = arr[i] + 5;
```

Unlike Java, there is no check that the index is in bounds.
Advice:

- Use `vector<T>` instead when the size is unknown
- With a fixed size use `array<T>` instead!

(Help the compiler – it'll pay you back!)



Arrays

We have already used vectors, but C++ also has arrays, which are fixed in size:

int arr[40];
for (std::size_t i = 0; i < 40; ++i)
arr[i] = arr[i] + 5;

Unlike Java, there is no check that the index is in bounds.
Advice:
- Use vector<T> instead when the size is unknown
- With a fixed size use array<T> instead!
(Help the compiler – it'll pay you back!)

We can find the length of an array using the **sizeof** function:
int l = sizeof(arr) / sizeof(int);
Only works if **arr** is the name of the array, not if it's a pointer. . .

sizeof(Name of the array**)**
 / sizeof(Type of the elements**)**

Pointers and arrays

When assigning or initializing from an array, a pointer to the first element is copied, not the array:

```
int arr[40];
int *p = arr;     // What's arr ???
```

Now `*p` is equivalent to `arr[0]`, and indeed to `*arr`.
The following are all equivalent:

```
arr[0] = arr[0] + 5;
*p = *p + 5;
*arr = *arr + 5;
```

Parameter passing

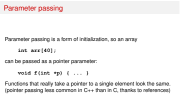
Parameter passing is a form of initialization, so an array

```
int arr[40];
```

can be passed as a pointer parameter:

```
void f(int *p) { ... }
```

Functions that really take a pointer to a single element look the same.
(pointer passing less common in C++ than in C, thanks to references)



Parameter passing

Parameter passing is a form of initialization, so an array

int arr[40];

can be passed as a pointer parameter:

void f(int *p) { ... }

Functions that really take a pointer to a single element look the same.
(pointer passing less common in C++ than in C, thanks to references)

But it might be used if we want to:

- re-use a C library; or
- write a C++ library that may be used by C programs as well.

C-style strings

- In C, strings are stored in `char` arrays, with the end of the string marked by a `'\0'` character.

```
char name[]="Bill";//array of 5 chars
char *name2="Fred";//pointer to a *const* array of 5 chars
```

- Often `char *` indicates a C-style string, *e.g.*,

  ```
  int main(int argc, char **argv);
  ```

- C++'s `string` type is much safer.
- A C-style string can be used where a `string` is expected, and is automatically converted.
  That's done with constructor `string(char *s);`
- If you need a C-style string for some legacy interface, use the method `c_str()` of `string`.
  For example, `string s; char *p = s.c_str(); foo(p);`

Pointer arithmetic

When `p` has type `T *`, and points to the i^{th} element of an array of `T`s:

```
T   arr[N];
T  *p = arr + i; // MUST: i < N !
```

Then:

- `p +` k is a pointer to the $(i+k)^{\text{th}}$ element.
- `++p` is equivalent to `p = p+1`
- `p -` k is a pointer to the $(i-k)^{\text{th}}$ element.
- `--p` is equivalent to `p = p-1`
- `p[`k`]` is equivalent to `*(p+`k`)`

Again, there are no checks that anything is in bounds.

Can also subtract two pointers (`ptrdiff_t`), which should be pointers to the same array (*NOT* checked of course. . .).

```
T          *p1 = arr + i; // MUST: i < N !
T          *p2 = arr + j; // MUST: j < N !
ptrdiff_t diff = p2 - p1; // = j - i
```

A Game!!!

Consider:

```
int arr[] = {1, 2, 3, 4, 5};
int *p = arr;
```

Which are ***legal***, which are ***illegal***?

1. `p[2]`
2. `2[p]`
3. `p + 2`
4. `arr[2]`
5. `2[arr]`
6. `arr + 2`

ONLINE QUIZ NOW! `t.ly/zZDlQ`

???

What do the legal ones mean?

Looping over an array

Given an array of integers:

```
int arr[40];
```

The following are (functionally) equivalent:

- Using indices *(slower)*:

```
for (std::size_t i = 0; i < 40; ++i)
    arr[i] = arr[i] + 5;
```

- Using pointers *(faster)*:

```
int *end = arr + 40;
for (int *p = arr; p != end; ++p)
    *p = *p + 5;
```

Notes:

- `arr + 40` ***SHOULDN'T*** be dereferenced.
- Pointer loop is ***faster!*** *(why?)*

Iterators

Iterators are objects providing sequential access to container elements

- The Java interface is analogous to a linked list or a stream:

```
public interface java.util.Iterator {
    boolean hasNext();
    Object next();
    void remove();   // not always supported
}
```

- C++ STL iterators are modelled after array pointers

Iterators in the STL

Iterating over a list of strings:

```
list<string> names;
...
for (list<string>::iterator p = names.begin();
        p != names.end(); ++p)
    cout << *p << ’\n’;
```

Sequences include a type `iterator` and two iterators:

begin() positioned ***at the start*** of the sequence, and

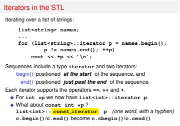
end() positioned ***just past the end*** of the sequence.

Each iterator supports the operators `==`, `++` and `*`.

- For `int *p` we now have `list<int>::iterator p`.
- What about `const int *p` ?

`list<int>::`***`const_iterator`*** `p` *(one word, with a hyphen)*

`c.begin()`/`c.end()` become `c.cbegin()`/`c.cend()`



- Prefer using `begin(container)` and `end(container)`
- Instead of `container.begin()` and `container.end()`
  - The former form works with arrays as well; *and*
  - It selects `container.begin()` or `container.cbegin()` automatically, depending on whether `container` is `const` or not.

A variation: typedefs

In C++ we can define new names for types using `typedef`:

```
typedef int time;
typedef char * cstr;
typedef deque<string> phrase;
typedef vector<vector<double> > matrix;
```

(We can also do this in C, but only outside functions.)

With `typedef` we can introduce an abbreviation for the iterator type:

```
typedef list<string>::iterator iter;
for (iter p = begin(names), e = end(names);
                                       p != e; ++p)
    cout << *p << ’\n’;
// Or *better*: USE auto!
for (auto p = begin(names), e = end(names);
                                       p != e; ++p)
    cout << *p << ’\n’;
```

The analogy

C		STL – C++98	
array	`arr`	container	`c`
pointer	`p`	iterator	`p`
start pointer	`arr`	start iterator	`c.begin()/cbegin()`
end pointer	`arr + LENGTH`	end iterator	`c.end()/cend()`
increment	`++p`		`++p`
dereference	`*p`		`*p`

Since C++11 – One API for all!

array	`arr`	container	`c`
pointer	`p`	iterator	`p`
start pointer	`begin(arr)`	start iterator	`begin(c)`
end pointer	`end(arr)`	end iterator	`end(c)`
increment	`++p`		`++p`
dereference	`*p`		`*p`

`begin(c)` returns a const/non-const iterator as appropriate! `:-)`

Iterator is a concept

- Iterator is an STL concept, not a C++ class.
- All iterators support the same operations in the same way:
  - Switching representations is relatively easy.
  - Generic code can be written using these operations.
- Special kinds of iterators support more operations.
- Checking is done when generic code is instantiated.

Iterator concepts in the STL

Different containers have different kinds of iterator, belonging to a hierarchy of iterator concepts:

Input Iterator supports `==`, `++`, (unary) `*` and `->`
e.g., the `iterator` of `forward_list` (née `slist`, see issue: https://stackoverflow.com/a/6885508)

Bidirectional Iterator supports all these as well as `--`
e.g., the `iterator` of `list`.

Random Access Iterator supports all these as well as `<`, `+`, `-` and `[]`, which should behave similarly to operations on pointers.
e.g., the `iterator` of `vector` or `deque`.

- Why isn't `<` supported for input/bidirectional iterators?
- What does `iter[3]` stand for?

A generic function

```
template <typename Iterator, typename Elem>
int count(Iterator start,
          Iterator finish, const Elem & v) {
    int n = 0;
    for (Iterator p = start; p != finish; ++p)
        if (*p == v)
            n++;
    return n;
}
```

There are several type requirements here (checked at instantiation):

- `Iterator` must be at least an ***input*** iterator type;
- `Iterator` must be an iterator with element type `Elem`; and
- The `Elem` type must support `==`.

Using the generic count function

The `count` function is defined in `<algorithm>`.
Here is an example of its use:

```
list<string> names;
string s;
....
std::size_t n = count(begin(names), end(names), s);
cout << s << " occurs " << n << " times\n";
```

In the above use,

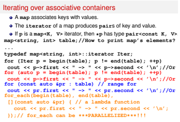
- `Iterator` is `list<string>::iterator`
- `Elem` is `string`.

Check `<algorithm>` *out!* en.cppreference.com/w/cpp/algorithm

Iterating over associative containers

- A **`map`** associates keys with values.
- The **`iterator`** of a map produces **`pair`**s of key and value.
- If **`p`** is a **`map<K, V>`** iterator, then **`*p`** has type **`pair<const K, V>`**.

```
map<string, int> table;//How to print map’s elements?
...
typedef map<string, int>::iterator Iter;
for (Iter p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << ’\n’;//Or
for (auto p = begin(table); p != end(table); ++p)
 cout << p->first << " -> " << p->second << ’\n’;//Or
for (const auto &pr : table) // range for
 cout << pr.first << " -> " << pr.second << ’\n’;//Or
for_each(begin(table), end(table),
 [](const auto &pr) { // a lambda function
   cout << pr.first << " -> " << pr.second << ’\n’;
 });// for_each can be ***PARALLELIZED***!!!
```



```
#include <string>
#include <iostream>
#include <algorithm>
#include <execution>

std::map<std::string, int> table;
std::for_each(std::execution::par_unseq,
                //instance of parallel_unsequenced_policy
              std::begin(table), // start from.
              std::end(table), // end before.
              // a lambda (anonymous) function
              [](const auto &pair) {
                 std::cout << pair.first
                           << " -> "
                           << pair.second
                           << std::endl;
              });// std::for_each ***PARALLELIZED***!!!
```

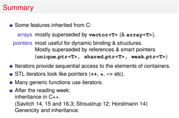
Check out en.cppreference.com/w/cpp/algorithm/reduce

Summary

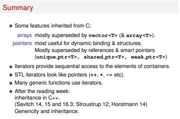
- Some features inherited from C:

arrays	mostly superseded by **`vector<T>`** (& **`array<T>`**).
pointers	most useful for dynamic binding & structures. Mostly superseded by references & smart pointers (**`unique_ptr<T>, shared_ptr<T>, weak_ptr<T>`**)

- Iterators provide sequential access to the elements of containers.
- STL iterators look like pointers (**`++`**, **`*`**, **`->`** etc).
- Many generic functions use iterators.
- After the reading week:
  inheritance in C++.
  (Savitch 14, 15 and 16.3; Stroustrup 12; Horstmann 14)
  Genericity and inheritance.



(Area left empty on purpose)



Summary

- Some features inherited from C:

arrays	mostly superseded by `vector<T>` (& `array<T>`).
pointers	most useful for dynamic binding & structures. Mostly superseded by references & smart pointers (`unique_ptr<T>, shared_ptr<T>, weak_ptr<T>`)

- Iterators provide sequential access to the elements of containers.
- STL iterators look like pointers (++, *, -> etc).
- Many generic functions use iterators.
- After the reading week:
  inheritance in C++.
  (Savitch 14, 15 and 16.3; Stroustrup 12; Horstmann 14)
  Genericity and inheritance.

Final Notes – I:

- Pointers are used with operators `&` (address-of) and `*` (dereference).
  - `&` returns the memory address where a variable/object(/function. . .) can be found.
  - `*` takes an address and returns the item at that address.
- Pointers are declared as
  `type * p = nullptr; // Not 0/NULL!!! C++11`
  Such declarations are read right-to-left: "`p` is a pointer (`*`) to a `type`". So given some integer `i`:
  - `const int *        p1 = &i;`
    `p1` is a pointer to a constant `int` (can point to another integer `j` but cannot be used to modify any of them)

```
int j = 3;
*p1 = 4; // attempt to modify i - invalid
p1 = &j; // attempt to point elsewhere - valid
```

  - `int * const p2 = &i;`
    `p2` is a constant pointer to an `int` (cannot point to another integer but *can* be used to modify the integer it's pointing at)

```
int j = 3;
*p2 = 4; // attempt to modify i - valid
p2 = &j; // attempt to point elsewhere - invalid
```

  - `const int * const p3 = &i;`
    `p3` is constant pointer to a constant `int` (cannot point to another integer nor be used to modify the integer it's pointing at)

```
int j = 3;
*p3 = 4; // attempt to modify i - invalid
p3 = &j; // attempt to point elsewhere - invalid
```

  - We can have pointers to pointers (to represent things like multi-dimensional arrays):

```
int ** pp1 = &p1;
```

    `pp1` is a pointer to a pointer to an `int` (or `pp1` is a double pointer to an `int`).
    `const` can be sprinkled around quite freely as before:
    `int * const * const pp2 = &p1;`
    Read it right-to-left: `pp2` is a constant pointer to a constant pointer to an `int`.
- The null pointer is `nullptr` since C++11 – use that instead of `0` or `NULL` (C language). See an article on "enums and nullptr in C++11" (https://www.cprogramming.com/c++11/c++11-nullptr-strongly-typed-enum-class.html),
- An array's name can be used as a pointer to the first element of the array. `int arr[40]; int *p = arr;`
- Pointers support arithmetic operators (slide 14). Incrementing a pointer takes you to the next address that represents an object of the type you're pointing at (so it's address+1 for a char, address+4 for a 32 bit `int`, address+432 for an object that's 432 bytes long, etc.)
- Array elements can be accessed with pointers (more efficient than indexes – slide 16):

```
for (int *p = arr, *end = arr+40; p != end; ++p)
  *p = *p + 5;
```

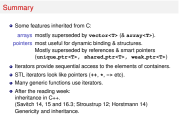
  This pattern is ***extremely important*** – it's how we use iterators to go over container elements.
  (Why more efficiently than indexes? Check slide 14 to see what `arr[i]` is translated to)
- Each container defines two types: `iterator` and `const_iterator`:

```
vector<int>::iterator i1;        // --->       int   *p1;
list<float>::const_iterator i2;  // ---> const float *p2;
```

  The looping pattern:

```
for (vector<int>::iterator
       p = begin(vi), end = end(vi); p != end; ++p)
  *p = *p + 5;
```

- Learn how to write generic functions that take iterators (slide 23)



Summary

- Some features inherited from C:
  arrays mostly superseded by `vector<T>` (& `array<T>`).
  pointers most useful for dynamic binding & structures. Mostly superseded by references & smart pointers (`unique_ptr<T>, shared_ptr<T>, weak_ptr<T>`)
- Iterators provide sequential access to the elements of containers.
- STL iterators look like pointers (++, *, -> etc).
- Many generic functions use iterators.
- After the reading week:
  inheritance in C++.
  (Savitch 14, 15 and 16.3; Stroustrup 12; Horstmann 14)
  Genericity and inheritance.

Final Notes – II:

- Also learn to use `auto` when your compiler supports C++11:
  The looping pattern:

```
 for (auto p = begin(vi), end = end(vi);
          p != end;
          ++p) {
      *p = *p + 5; // LEARN THIS!!!
  }
```

  - Functions `begin(c)` and `end(c)` work when `c` is either a container or an array (C++11), while `c.begin()` and `c.end()` only work with containers – use the former form rather than the latter.
    Both functions return the correct iterator (`const` or not) depending on whether `c` is `const` or not: watch out for this – might cause compilation errors if you try to store it in the wrong iterator variable:

```
void print( const vector<int> & v ) {
// for (vector<int>::const_iterator // CORRECT
   for (vector<int>::iterator // ERROR
          p = begin(v),
          end = end(v);
        p != end;
        ++p)
      cout << *p << ' ';
}
```

  - Crash course on auto:

```
int      i = 3;
auto     j = i; /* j is also an int, initialized as a
                   copy of i */
auto && k = i; /* k is a *reference* to an int (&& is
                   not a typo - use that with auto) */
const auto && m = i; /* m is a constant reference to
                         an int */
```

  - More on auto: https://www.cprogramming.com/c++11/c++11-auto-decltype-return-value-after-function.html
  - More on rvalue references (`&&`): https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
    (advanced – not to be examined. First time I read this I had to go and lie down – haven't read it again since. . .).

File `copy-string.cc` (*) contains four different implementations of a function that copies a source (`s`) C-style string (e.g., an array of characters) into a target (`t`) C-style string.

Version `strcpy3` is the canonical one – once you've understood why/how it works, your understanding of pointers should be quite good (and of the difference between `i++` and `++i`).
(*) https://www.staff.city.ac.uk/c.kloukinas/cpp/session-05/copy-string.cc

```
// *** The ONE, TRUE strcpy!!! ***
void strcpy3(const char *s, char *t) {
  while ((*t++ = *s++)) /* extra parentheses added
                           to get rid of warning */
    ; /* do nothing in the body - loop condition
         does the job */
}
/*
 * Source: Kernighan & Ritchie, The C Programming
 * Language, 2nd Edition, Prentice Hall PTR, 1988,
 * p. 106
 *
 * strcpy: copy s(ource) into t(arget).
 * ASSUMPTION: t(arget) has enough space for the
 * string inside s(source)!
 */
```