# Lecture 4

C-array,
Command line parameters,
STL, and lambda functions

## Pointer

- Remember?
- A variable that store the address of some other variable.
- A * before an address will go to that address.
- A & before a variable will get the address where the variable is stored.

## Pointer and const

```
int variable;
int * pointer{ &variable };
int *const pointer{ &variable };

int const* pointer{ &variable };
const int* pointer{ &variable };

int const* const pointer{ &variable };
const int* const pointer{ &variable };
// Read backwards!
```

## Reference and const

- Similar to how it works for pointers
  – But simpler, only one variable involved

```
int variable;

int& ref{ variable };
int const& reference{ variable };
const int& reference{ variable };
```

## Pointer arithmetic

- You can use plus (+) and minus (-) on an address to calculate a new address.
- You must make sure the calculated address is not used for anything else!

```
int x{2};
int y{7};
int* z{&x};
// Following both compiles and run:
int* what = z + 1; // ERROR!!!!!!
cout << *what << endl; // ??
```

## A *VERY* BAD c-array idea!

```
int main()
{
  int a, b, c, d, e;
  int* x{ &a };

  for (int i{0}; i < 5; ++i)
    cin >> *(x+i);

  for (int i{4}; i >= 0; --i)
    cout << *(i+x) << endl;
}
```

## But that's essentially it!

```
int main()
{
  int a[5]; // a, b, c, d, e;
  int* x{ a }; // note the removed &

  for (int i{0}; i < 5; ++i)
    cin >> *(x+i);

  for (int i{4}; i >= 0; --i)
    cout << *(i+x) << endl;
}
```

## But we can beautify

```
int main()
{
  int a[5];
  int* x{ a };

  for (int i{0}; i < 5; ++i)
    cin >> x[i]; // much better!

  for (int i{4}; i >= 0; --i)
    cout << i[x] << endl; // Wait... what?
}
```

## To do it right

```
int main()
{
  int a[5];

  for (int i{0}; i < 5; ++i)
    cin >> a[i];

  for (int i{4}; i >= 0; --i)
    cout << a[i] << endl;
}
```

## Or do it old-time efficient

```
int main()
{
  int a[5];

  for (int* i{a}; i < &a[5]; ++i)
    cin >> *i;

  for (int* i{&a[4]}; i >= a; --i)
    cout << i[0] << endl; // What??
}
```

## Common errors

```
int a[5]{1,2,3,4,5}
int i{ -1 };
int j{ 5 };
int* x;
a[i] = 4711;
a[j] = 4711;
*x   = 4711;
x[1] = 4711;
```

## Create array at runtime

```
int main()
{
  int size;
  cin >> size;
  int* a{ new int[size] }; // borrow memory (allocate)

  for (int i{0}; i < 5; ++i)
    cin >> a[i];

  for (int i{4}; i >= 0; --i)
    cout << a[i] << endl;

  delete[] a; // return borrowed memory (deallocate)
}
```

## new[] and delete[]

- new type[ count ]
  - Allocate sequential space in memory for count variables of type
  - Initialize all of the variables to default value
  - Returns the adress of the first variable
- delete[] adress
  - Requires the adress of the first variable, the exact address that was returned from new
  - Destroys all variables in array
  - Deallocate the memory (that new allocated for that adress)

## C-string

- An array of characters.
- Lenght unknown.
- Always end with character '\0'.

```
char const* c_string = ”C-string”;
string cpp{ c_string };
char const* str{ cpp.c_str() };
// strlen sometimes useful
```

## Array of C-strings

```
// Read backwards!
char* c_string_array[5];

char** c_string_array;
```

## Command line arguments

```
a.out testing 1 2 3

int main(int argc, char* argv[])
{
  for (int i{0}; i < argc; ++i)
  {
    cout << argv[i] << endl;
  }
}
```

## Lambda function

- Normal function
  ```
  result name(parameters...) { body; }
  auto name(parameters...)->result { body; }
  ```
- Lambda function
  ```
  [ capture ]( parameters... )->result { body; }
  ```
- Capture section
  - Specification of any surrounding variables that should be accessible in the function
  - Capture take place when the lambda function is *created*
  - Capturing references poses a danger!
  ```
  [a, &b] // capture a by copy and b by reference
  &       // capture all by reference
  =       // capture all by copy
  []      // capture nothing (good default)
  ```

## Function pointer

- Or looking back how we did it in C

```
double line(double x) { return 3.0*x; }

void plot(double (*fn)(double))
{
  for ( ... )
    draw_dot(x, fn(x));
}

int main()
{
  plot(line);
  plot(sin);
  plot(tan);
}
```

## Using & typedef

- About setting a simple name on something that else looks very awkward.
- Allows to define aliases for type names
- Useful to increase readability

```
// typedef void (*function_pointer)(int)); // old way
using function_pointer = void (*)(int);
void call_me(function_pointer fn) { fn(5); }
```

- Useful to increase changeability

```
// typedef int money; // we do not use the old way
using money = int; // money is a new type name for int
money bank_account;
```

- Compare:
  - Using a const variable instead of literals let you change that value at one place instead of dozen places.
  - Using a type alias instead of the real type let you change that type at one place instead of dozen places.

## Using templates

- With templates you can write generic code
- You add placeholders instead of data types

```
template<typename PlaceHolder>
void swap(PlaceHolder& a, PlaceHolder& b);
int a, b;
double c, d;
string e, f;
swap<int>(a, b);    // PlaceHolder => int
swap<double>(c, d); // PlaceHolder => double
swap<string>(e, f); // PlaceHolder => string
swap<int>(a, f);    // Error: f is not int
swap(c, d); // Ok, PlaceHolder => int automatic
```

## Standard Template Library

- Motivation
  - Easy access to *efficient* and *type safe* implementations of basic data structures
  - Easy access to common algorithms
  - Uniform interface to data structures and algorithms
- Implementation
  - Uses C++ *templates* for everything
  - Containers provide various types of data collections
  - Iterators provide iterable access to data in containers
  - Algorithms work on iterators
  - Adapters and utilities tie up loose ends

## Basic STL containers

```
std::pair
std::tuple
std::function
std::array
std::vector
std::string
std::forward_list
std::list
std::set
std::unordered_set
std::map
std::unordered_map
// Reference: www.cplusplus.com/reference
```

## std::pair

- Stores (groups) two data items
- They do not have to be of same type
- All comparisions based on first element

```
#include <utility>
pair<string, int> my_pair;
// can be initialized with {}

my_pair.first;  get<0>(my_pair);
my_pair.second; get<1>(my_pair);

my_pair = make_pair(”text”, 4711);
```

## std::tuple

- Stores (groups) any fix number of data items
- They do not have to be of same type

```
#include <tuple>
tuple<string, int, float> my_tup;
// can be initialized with {}

get<0>(my_tup);
get<1>(my_tup);
get<2>(my_tup);

my_tup = make_tuple(”text”,4711,3.14);
```

## std::function

- Stores a function in an object
- You can pass the function object as parameter
- You can call the function from the object

```
#include <functional>

function<void(string)> my_fun;

my_fun = [](string msg)->void
{
  cout << msg;
};
my_fun("hello world");
```

## std::array

- Store a fix length seqence in sequential memory
- All data items must have same type

```
#include <array>
array<int, 5> my_array;

my_array.size();
my_array.at(3); // validation
my_array[3]; // no validation
```

## std::vector

- Store a dynamic length seqence
- All data items must have same type
- Optimized for random access
- Inefficient inner insertion

```
#include <vector>
vector<int> my_vector;
my_vector.push_back(4711);
my_vector.size();
my_vector.at(3); // validation
my_vector[3]; // no validation
```

## std::string

- Store a sequence of characters
- Essentially a vector<char> with specialized features

## std::forward_list and std::list

- Stores a dynamic lenght sequece
- All elements must be of same type
- Optimized for inner insertion
- Not optimized for random access
- Forward list iterates only one way, but use less memory

```
forward_list<int> my_fwd_list;
my_fwd_list.push_front(4711);
my_fwd_list.sort();

list<int> my_list;
my_list.push_front(4711);
my_list.push_back(4712);
my_list.sort();
```

## std::set and std::unordered_set

- Stores a collection of unique immutable values
- std::set provide efficient search (sorted)
- std::unordered_set provide efficient direct access

```
#include <set>
set<int> my_set;
my_set.insert(4711);
my_set.find(4711);
my_set.erese(4711);
#include <unordered_set>
unordered_set<int> my_uset;
```

## std::map and std::unordered_map

- Stores a collection of uniqe keys
- Each key is associated with a value
- Think of a set that stores pair<key, value>

```
#include <map>
map<string, int> my_map;
// awkard insertion
my_map.insert(make_pair(”key”, 4711));
// easy insertion or access
my_map[”key”] = 4712;
// easy access without insertion
my_map.at(”key”) = 4713;
my_map.find(”key”);
#include <unordered_map>
unordered_map<int, float> my_umap;
```

## Common operations

- Philosophy
  - Include only the operations that are efficient
  - Excluded operations does not make sense
  - Use another container type if you need them
- Commonly found operations

```
size(), empty(), cbegin(), cend(),
front(), back(), push_front(),
push_back(), pop_front(), pop_back(),
at(), operator[], fill(), swap(),
emplace(), emplace_front(),
emplace_back()
```

## Iterator concept

- To iterate a collection of data we need
  - A starting point (begin)
  - Some way to get to next data in the collection (next)
  - Some way to get from the iterator to the actual data
  - An ending point (end)
- Container iterators provide this
  - cbegin(), begin(), crbegin(), rbegin()
  - preincrement and postincrement (++)
  - dereference (*)
  - cend(), end(), crend(), rend()

## Forward iterator

- Begin
  - Refer to first element of container
  - Valid to dereference on non-empty container
  - Incremented toward last element (forward iteration)
- End
  - Refer to *just after* last element of container
  - Invalid to dereference
- Data type

  ::iterator

## Reverse iterator

- Rbegin
  - Refer to last element of container
  - Valid to dereference on non-empty container
  - Incremented toward first element (backward iteration)
- Rend
  - Refer to *just before* first element of container
  - Invalid to dereference
- Data type

  ::reverse_iterator

## Iterator over constant data

- begin(), end(), rbegin(), rend()
  - return mutable (non-const) iterators
  - *data in container* can be modified through iterator
  - None of the above refer to same position! Draw a picture!
- cbegin(), cend(), crbegin(), crend()
  - return immutable (const) iterators
  - *data in container* can only be read
  - type ::const_iterator or ::const_reverse_iterator

## Which iterator to use?

- Depend on what you want to do!
- A good safe default
  ::const_iterator, cbegin(), cend()
- If you *really* need to change data
  ::iterator, begin(), end()
- If you really need to go backwards
  ::const_reverse_iterator
  ::reverse_iterator (if you need mutable access)

## Iterator in C

- Modeled after pointers and pointer increment

```
int a[5]; // container
int* i; // iterator

//       begin       end    next
for (i = &a[0]; i != &a[5]; ++i)
  cin >> *i; // dereference
```

## Iterator in STL

- Modeled after pointers and pointer increment

```
vector<int> a(5); // container
vector<int>::iterator i;

for (i = a.begin(); i != a.end(); ++i)
  cin >> *i; // dereference
```

## Iterator loop

- Simplifies container iteration

```
vector<int> v(5); // container

for ( int& i : v )
  cin >> i;

for ( int const& i : v )
  cout << i << endl;
```

## Basic algorithms

```
#include <algorithm>
std::for_each
std::sort
std::transform
std::find
std::copy
std::swap
std::shuffle
std::min_element
std::max_element
```

## std::for_each

- Iterates a range and call a lambda function on each element

```
vector<int> v{1,2,3,4,5,6,7,8,9};
int sum{0};
function<void(int)> fun;
fun = [&sum](int i)->void
{
  sum += i;
}
for_each(v.cbegin(), v.cend(), fun);
cout << sum << endl;
```

## std::sort

- Sort a range according to comparision lambda

```
vector<string> v{”12”, ”09”, ”31”};
function<bool(string const&,
              string const&)> less;
less = [](string const& a,
          string const& b)->bool
{
  return (a.at(1) < b.at(1));
}
sort(v.begin(), v.end(), less);
```

## std::transform

- Iterates a range and call a function on each element, putting the result in output range

```
vector<int> in{1,2,3,4,5,6,7,8,9};
vector<int> out(9); // set size 9
function<int(int)> fun;
fun = [](int i)->int
{
  return i*i;
}
transform(in.cbegin(), in.cend(),
          out.begin(), fun);
```

## std::find, std::copy, std::swap

- std::find
  - Find position of some value in a range
  - Return iterator to found element
  - Return end() if not found
- std::copy
  - Copy a range to another range
- std::swap
  - Swap two variables

## std::min_element, std::max_element and std_shuffle

- std::min_element
  - Find the minimum element in range
  - Returns an iterator to the minimum element
  - Custom compare function can be specified
- std::max_element
  - As min_element but finds maximum
- std::shuffle
  - Shuffles a range

## Iterator adaptors

- std::back_inserter (analogous for std::front_inserter)
  - gives an iterator that push_back() new items

```
#include <iterator>
vector<int> in{1,2,3,4,5}, out;
// out is empty, out.begin() invalid
transform(in.cbegin(), in.cend(),
          back_inserter(out), fun);
```

- std::istream_iterator and std::ostream_iterator

```
#include <iterator>
copy(istream_iterator<int>(cin),
     istream_iterator<int>(),
     ostream_iterator<int>(cout, “, “));
```

## using and/or auto

- STL types can get long and complex:

```
map<int, vector< tuple<int, double, string> > > my_map;
map<int, vector< tuple<int, double, string> > >::iterator it;
it = my_map.begin();
pair<int, vector< tuple<int, double, string> > data = *it;
```

- auto can simplify, but you can no longer see the type used:

```
map<int, vector< tuple<int, double, string> > > my_map;
auto it = my_map.begin();
auto data = *it;
```

- using let you name your types conveniently and (not here) descriptive:

```
using my_key_t = int;
using my_value_t = vector< tuple<int, double, string> >;
using my_map_t = map<my_key_t, my_value_t>;
my_map_t my_map;
my_map_t::iterator it = my_map.begin();
pair<my_key_t, my_value_t> data = *it;
```

## Random numbers in C++11 STL

```
#include <random>
random_device rand;
int r = rand(); // a random number
uniform_int_distribution<int> die(1,6);
int n = die(rand); // random in [1..6]
```

- Further reference:
  - your favorite book
  - cplusplus.com

## Random numbers (traditional C)

```
#include <cstdlib>
#include <ctime>
// important: call 'srand' once only!
srand(time(nullptr));
int r = rand(); // a random number
// a random number in [1..6]
int n = (rand() % 6 + 1);
```

## Using the debugger

- Needed to debug problems with invalid pointers and references:

```
segmentation fault, bus error
```

- Make sure your compilation command contain the '–g' flag:

```
g++11 -g my_buggy_program.cc
```

- Load your program in the debugger:

```
gdb a.out
```

- Start your program, add command line arguments if needed:

```
run arg1 arg2 arg3
```

- Do whatever causes your program to crash, and then retrieve a backtrace:

```
backtrace
```

- The backtrace will show where the program was executing, and how it gor there.

## A backtrace example

```
g++11 –g debug_example.cc
gdb a.out
(gdb) run 1234-56-89
Starting program: /home/klaar/Cplusplus/a.out 1234-56-89
[Thread debugging using libthread_db enabled]
[New Thread 1 (LWP 1)]
/home/klaar/Cplusplus/a.out is not a date
1234-56-89 is a date

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1 (LWP 1)]
0xff132d50 in strlen () from /lib/libc.so.1
(gdb) backtrace
#0  0xff132d50 in strlen () from /lib/libc.so.1
#1  0x00043554 in is_date (str=0x0) at debug_example.cc:10
#2  0x000436b0 in main (argc=2, argv=0xffbfe104) at debug_example.cc:29
```

## To be continued.

This page is not left blank. Intentionally.