Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

3 Data structures and libraries

3.1 Co-design data & code
3.2 Shared libraries
3.3 CMake
3.4 Containers
3.5 Linked data
3.6 Graphs and trees

25th February 2025

Creating a dynamic (shared) library

Norwegian University of Life Sciences

A **shared object file** can be created from an object file using g++ **-shared**:

Example: **shared-library.zip**

```
g++ -c -fPIC -o first.o first.cpp
g++ -c -fPIC -o second.o second.cpp
g++ -fPIC -shared -o libname.so first.o second.o
```

needed on some systems, when reusing certain libraries, to enforce "position-independent code" (PIC) object files

this is a capital i, not a lower-case L

The library header location can be passed to g++ at compile time with -I…, and the shared object is found by the linker with the -L and -l options.

this time it is a lower-case L

But the library also needs to be found at execution time. For that to work, it must be in the appropriate path, or one of the environment variables for library paths must be set to include the location of the shared object.

this can be $LD_LIBRARY_PATH

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

3 Data structures

3.3 CMake

3.4 Containers

Generating makefiles using CMake

CMake is used by many complex C/C++ projects that require developers or users to compile code on their systems, which may be very diverse. Typically:

cmake . && **make** && sudo make install

There, **CMake** generates the Makefile that is then used by **GNU make**.
We have done this before when we looked into the C++ interface to ROS.

Instructions for CMake are communicated through a file called **CMakeLists.txt**.

- CMake documentation: https://cmake.org/cmake/help/latest/
- CMake tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/

CMake can be helpful if your project has a complex system of dependencies, or compile-time case distinctions are needed beyond what you can implement in a simple way using GNU make; *e.g.*, embedded system cross-compiling.

Generating makefiles using CMake

CMake project example (**cmake-dirgraph.zip**):

- Working folder (unpack into that folder)
- Code in subdirectory **./src**.
- Data in subdirectory **./data**.
- **CMakeLists.txt** (see commands[1]) in the main folder and in the **./src** folder.
- Calling "**cmake .**" in the main working folder generates Makefiles in both folders.

./CMakeLists.txt

```
cmake_minimum_required(
  VERSION 3.17
)
project(
  dirgraph
    VERSION 1.0.0
    LANGUAGES CXX
)

set(CMAKE_CXX_STANDARD 20)
add_subdirectory(src)
```

src/CMakeLists.txt

```
set(EXECUTABLE_OUTPUT_PATH ../bin)
add_executable(dirgraph graph.cpp query.cpp run-graph.cpp)
```

Now **make** will automatically call g++ with the right options and flags.

[1]See: https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html

CMake support for unit tests

Unit tests are generally a helpful debugging tool in complex development projects. Here they can also help the user verify that everything worked well.

CMakeLists.txt (**cmake-dirgraph.zip**)

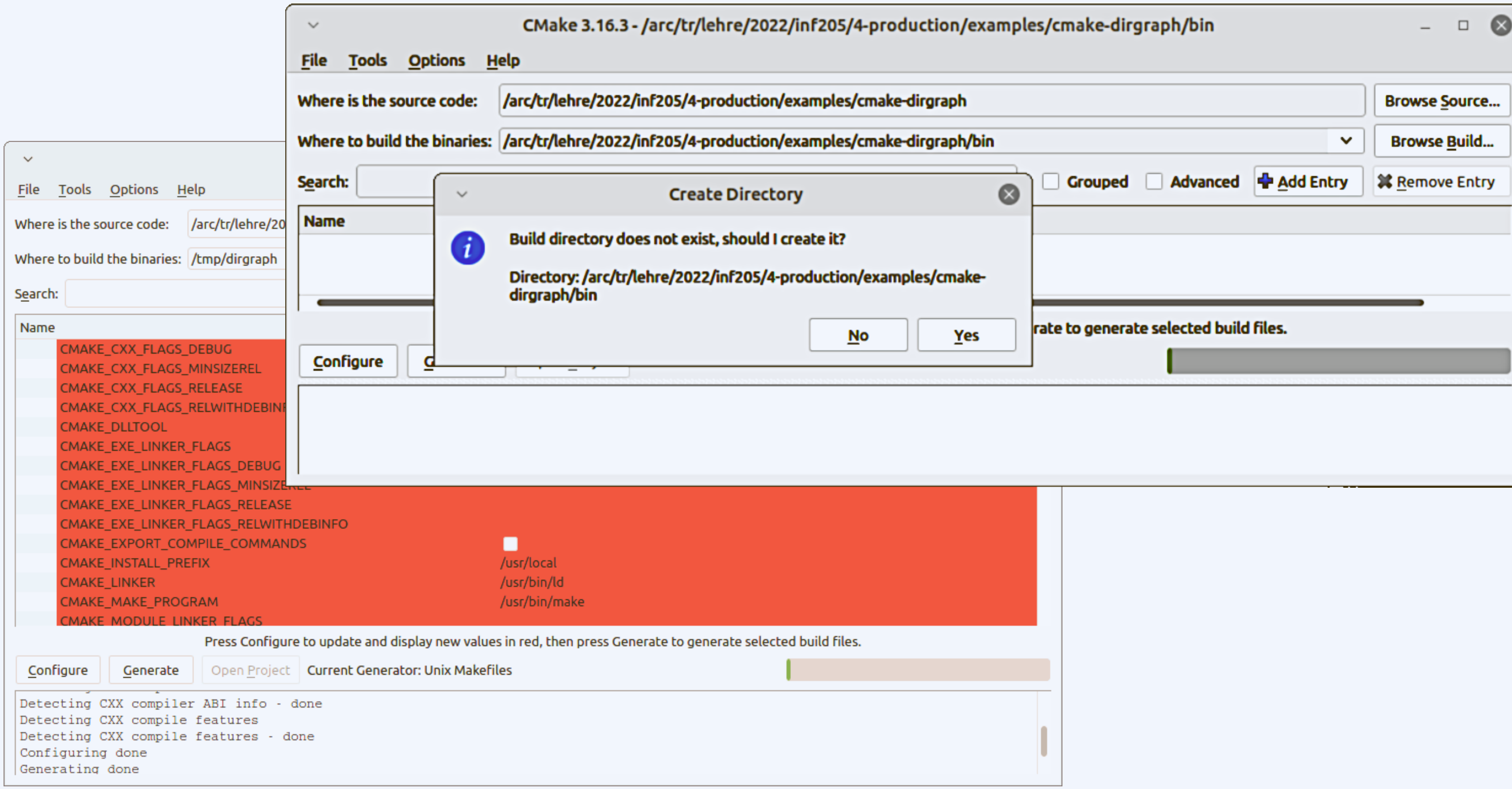
```
enable_testing()
add_test(
  NAME example_graph
  COMMAND dirgraph kb.dat query.dat
  WORKING_DIRECTORY data
)
set_tests_properties(
  example_graph
  PROPERTIES
  PASS_REGULAR_EXPRESSION "<INF200 2022H>[ \t\r\n]*<Rune Grønnevik>"
  PASS_REGULAR_EXPRESSION "<INF205 2023H>[ \t\r\n]*<Trine Næss Henriksen>"
  PASS_REGULAR_EXPRESSION "<KJM230 2023V>[ \t\r\n]*<Heidi Rudi>"
)
```

- **^** Matches at beginning of input
- **.** Matches any single character
- **[Xy2]** Any of the characters X, y, or 2
- **[^vV]** Any character other than v or V
- **[C-F]** Any of the characters C, D, E, or F
- ***** Preceding pattern occurs >= 0 times
- **+** Preceding pattern occurs >= 1 time
- **?** Optional (occurs 0 or 1 times)
- **|** Disjunction ("or")

https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html#test-properties

CMake GUI

Graphical interface to CMake: **cmake-gui**



Noregs miljø- og
biovitskaplege
universitet

Institutt for datavitskap

Digitalisering på Ås

3 Data structures

3.3 CMake

3.4 Containers

Rule of three

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

"**Rule of three:**" For a container, implement at least
(1) destructor,
(2) copy constructor,
(3) copy assignment operator.

Most often you will then also need to implement **(0)** a constructor.

At least implement **(1) the destructor**!
If **(2)** and **(3)** are not there, **forbid copying**.

Rule of three: (1) Destructor

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

Example: Let us assume that **class T** has one property for which it has ownership, a pointer p to class S that points to an array of 1000 S elements.

It is typical for the owned content, if manual memory management needs to be done, to be allocated in the constructor, **T::T()** and/or **T::T(...)**.

```
T tobject;
```

```
T* tpointer = new T;
```

```
class T
{
public:
   T() { this->p = new S[1000](); }
   ~T() { delete[] this->p; }
   ...
private:
   S* p = nullptr;
   ...
}
```

Rule of three: (2) Copy constructor

The **copy constructor T::T(const T& orig)** is called when the following two are done at the same time: **(1) allocation** of an object, so that a constructor needs to be called, and its **(2) initialization** to the value of a pre-existing object that continues to exist.

Examples for when the **copy constructor** is called:

```
// default constructor
T tfirst;
…
// copy constructor
T tsecond = tfirst;
```

```
void func(T param) { … }

int main() {
   T tobject;
   …
   // copy constructor
   func(tobject);
}
```

after running the copy constructor, the same content must exist in memory twice!

```
class T
{
public:
   T() { this->p = new S[1000](); }
   T(const T& original) {
      this->p = new S[1000]();
      std::copy(
         original.p, original.p+1000,
         this->p
      );
   }
   …
}
```

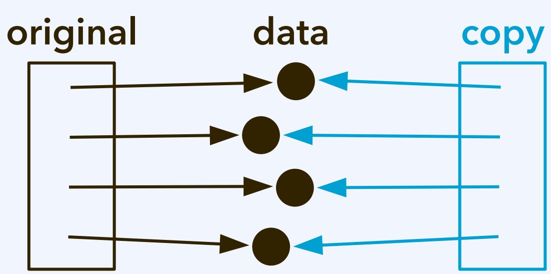
std::copy can be used for data that are contiguous in memory

1. Create space for the duplicate.
2. Now write the duplicate into it.

Copying an object

Shallow copy:

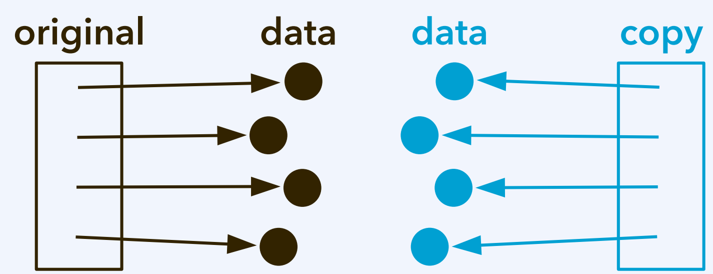
Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After shallow copying, the content will exist **once in memory**. This can be appropriate when the content is **not owned** but just pointed at.

Deep copy:

Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After deep copying, content exists **twice in memory**. Design following the concept of a "container" that **uniquely "owns"** its content requires deep copying.

Fast copying (to implement deep copying)

Example: **std-copy.cpp**

Element-wise copying

```
for(int i = 0;  i < num_copy;  i++)  target[i] = source[idx_start + i];
```

This is slow! Don't do this for large numbers of elements adjacent in memory. Also, note that Core Guidelines recommend "size_t" instead of int.

C-style fast copying (apply this only to a traditional C/C++ array)

```
#include <cstring>
...
std::memcpy(target,  source + idx_start,  num_copy * sizeof(element_type));
```

Modern C++ style fast copying (can also be used for STL containers)

```
#include <algorithm>
...
std::copy(source + idx_start,  source + idx_start + num_copy,  target);
```

Rule of three: (3) Copy assignment operator

The **copy assignment operator** technically is an overloaded "=" operator:

```
T& T::operator=(const T& rhs) { … }
```

Difference from the copy constructor:

- Object already exists, hence *no initial allocation* of memory for content.
- But *deallocate pre-existing content*.

```
// default constructor
T tfirst, tsecond;
…
// copy assignment
tsecond = tfirst;
```

A **copy assignment** is done whenever we copy the value of one variable to another, **both existed** before, and **both continue to exist**.

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T& operator=(const T& rhs) {
    if(&rhs == this) return *this;
    delete this->p;
    this->p = new S[1000];
    std::copy(
      rhs.p, rhs.p+1000, this->p
    );
    return *this;
  }
  …
}
```

Note that a reference to *this needs to be returned.

Copy assignment operator

The **copy assignment operator** technically is an overloaded "=" operator:

```
T& T::operator=(const T& rhs) { … }
```

Difference from the copy constructor:

- Object already exists, hence *no initial allocation* of memory for content.
- But *deallocate pre-existing content if necessary*.

```
// default constructor
T tfirst, tsecond;
…
// copy assignment
tsecond = tfirst;
```

A **copy assignment** is done whenever we copy the value of one variable to another, **both existed** before, and **both continue to exist**.

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T& operator=(const T& rhs) {
    if(&rhs == this) return *this;


    std::copy(
      rhs.p, rhs.p+1000, this->p
    );
    return *this;
  }
  …
}
```

after running the copy assignment, the same content must exist in memory twice!

Rule of three and rule of five

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

The programmer needs to take care of this whenever there are data subject to **manual memory management** (*new* and *delete*) in a **self-designed container**.

"**Rule of five:**" Implement

- **(1)** destructor,
- **(2)** copy constructor,
- **(3)** copy assignment operator,
- **(4)** move constructor,
- **(5)** move assignment operator.

Most often you will then also need to implement **(0)** a constructor.

"**Rule of three:**"

- **(1)** destructor,
- **(2)** copy constructor,
- **(3)** copy assignment operator.

At least implement **(1) the destructor**!
If **(2)** and **(3)** are not there, **forbid copying**.

Rule of five: (4) Move constructor

The **move constructor** is called when the content of an *old object* can be shifted to a *new object* that is *allocated and initialized* (*e.g.*, *before we deallocate the old object*).

```
T::T(T&& old) { … }
```

```
T func(…) {
   T tfirst;
   …
   return tfirst;
   // the destructor will be called
}

int main() {
   // but before, call the move constructor
   T tsecond = std::move( func(…) );
}
```

Typical use case: Efficient **handover of content** returned by a function.

```
class T
{
public:
   T() { this->p = new S[1000](); }

   T(T&& old) {
      this->p = old.p;
      old.p = nullptr;
   }
   …

private:
   S* p …
}
```

A **shallow copy** of the pointer to the content is good enough; after the action, the content *exists in memory only once*!

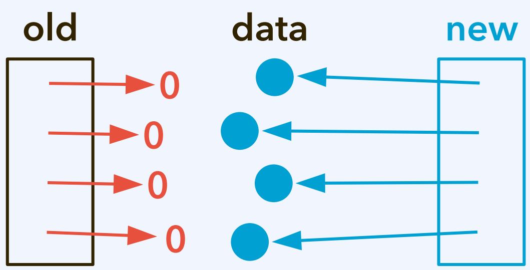
Attention: Right after the move constructor for "this", the destructor of "old" might be called.

Remove all pointers to the content from old, so that it does not get deallocated!

Move: Why can it be advantageous?

Move constructor + destructor:

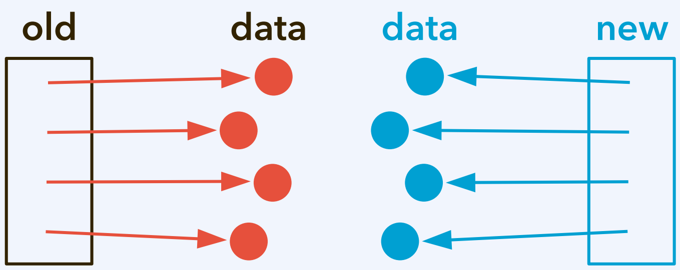
The move constructor is used to *make a new container own the data* without copying the data. A **shallow copy** is made, and the *data are detached* from the old container.



The shallow copy is an inexpensive operation. If the data exist **once in memory** both before the operation and after, *why copy them* from one place to another?

Copy constructor + destructor:

If there is *no move* constructor, or the compiler does not enforce a move, first all the content is copied (**deep copy**); the old container is probably *deallocated right after*.



This is an expensive operation whenever there is a substantial amount of data. All data are *copied*, *unnecessarily*, since at the end they still exist only **once in memory**.

Rule of five: (5) Move assignment operator

The move assignment operator relates to the move constructor the same way as the copy assignment operator relates to the copy constructor.

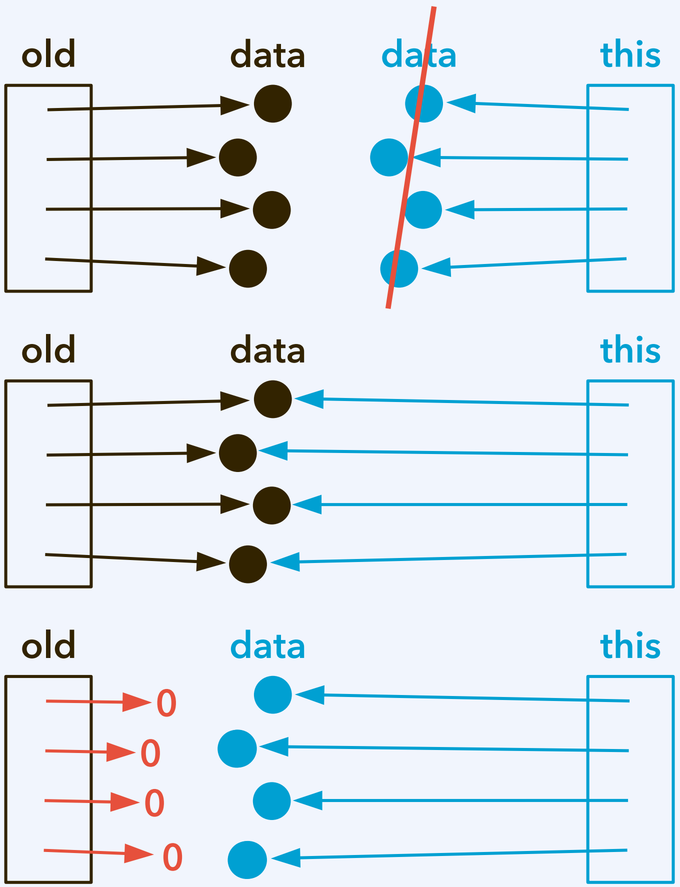
```
T& T::operator=(T&& old) { … }
```

```
T func(…) {
  T tfirst;
  …
  return tfirst;
  // the destructor will be called
}

int main() {
  T tsecond;
  …
  // but before, call the move assignment operator
  tsecond = std::move( func(…) );
}
```

constructor called

tsecond exists already



Example: Copying vs. moving

Norwegian University of Life Sciences

See **copying-and-moving.zip** for an implementation and performance comparison between the STL and self-implemented sequences with int elements.

Below: Copy and move assignment operators for the singly linked list.

```
// copy assignment: clear pre-existing content,
// then make a deep copy of original content

SinglyLinkedList& SinglyLinkedList::operator=(
  const SinglyLinkedList& right_hand_side
) {
  if(&right_hand_side == this) return *this;
  this->clear();    // remove pre-existing content

  for(
    auto n = right_hand_side.begin();
    n != nullptr;
    n = n->get_next()
  ) this->push_back(n->get_item());

  return *this;
}
```

```
// move assignment: clear pre-existing content,
// then shallow-copy pointers to moved content

SinglyLinkedList& SinglyLinkedList::operator=(
  SinglyLinkedList&& old
) {
  if(&right_hand_side == this) return *this;
  this->clear();  // remove pre-existing content

  // now proceed as for the move constructor
  this->head = old.head;
  this->tail = old.tail;
  old.head = nullptr;
  old.tail = nullptr;

  return *this;
}
```

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

3 Data structures and libraries