Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

2 The C++ programming language

2.1 Features of C++
2.2 Pointers and arrays
2.3 Pass by value or reference
2.4 Memory allocation
2.5 Immutability and constants
2.6 Working with libraries

Norwegian University
of Life Sciences

Weekly glossary concepts

What are essential concepts from the previous lecture?

Let us include them in the **INF205 glossary**.[1]

namespace

???

???

reference

pointer

???

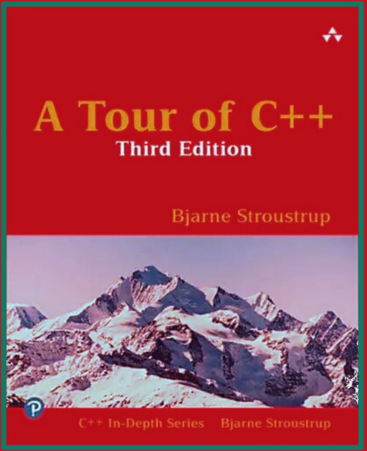
[1] https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

2 C++ basics



book Section 1.5

The three most typical memory bugs

1) Access a pointer that was **not initialized**, or that has the value **nullptr**, or that for any other reason points to an **invalid address** in memory. (“**Wild pointer**.”)

- **Question:** Why is this dangerous?

2) Memory is **allocated** using new, **but not deallocated again** using delete. This is called a **memory leak**.

- **Question:** Why is this dangerous?

3) Memory **has been deallocated**: Either it was on the stack in a stack frame that has been removed, or there has been a delete statement. But the address information was stored in **a pointer that still exists**: A **dangling pointer**!

- **Question:** Why is this dangerous?

Code that can produce wild pointers

```
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped) {
        for(int i = 0; i < num_strings; i++)
                strings[i] += characters_cropped;
}
```

1. What are argc and argv? Where do they come from?

Plan: Remove the first 2 characters from each command-line argument.

Implementation: Simply jump ahead by <characters_cropped> char's, that is, here, by 2 characters.

Example file: wildptr.cpp

Code that can produce wild pointers

2. What sort of a statement is this? Why do we need it?

```
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped) {

        for(int i = 0; i < num_strings; i++)
                strings[i] += characters_cropped;

}
```

3. Why this "-1" and "+1" on argc and argv?

4. How was this supposed to work? When will it inadvertently generate a wild pointer?

used for assert(condition), which checks that condition is true

used for strlen(char str), which returns the length of a C string*

Note that this is <u>one less</u> than the size of str as a char array:
For strlen, the terminal \0 character does not count.

```
#include <cassert>
#include <cstring>
#include <iostream>

void crop(int num_strings,  char** strings,  int characters_cropped);

int main(int argc,  char** argv) {
        int jump_to_index = 2;
        crop(argc-1,  argv+1,  jump_to_index);

        for(int i = 1;  i < argc;  i++)
                std::cout  <<  "Argument no. "  <<  i  <<  " was cropped to \""  <<  argv[i]  <<  "\".\n";
}

void crop(int num_strings,  char** strings,  int characters_cropped)  {
        assert(characters_cropped >= 0);
        for(int i = 0; i < num_strings; i++)
                if(strlen(strings[i]) >= characters_cropped)
                        strings[i] += characters_cropped;
                else strings[i] += strlen(strings[i]);
}
```

Example file: wildptr-fixed.cpp

Code that produces a memory leak

High-level languages operate with *automated garbage collection:* Memory is deallocated when there are no more variables referring to it.

C/C++ memory management on the heap must be done by hand. This causes two possible bugs:

Memory leak: Memory should have been deallocated, but was not.

Dangling pointers: Memory has been deallocated, but it should not.

```
float* crw::step(long size, float previous[])
{
    // allocate the next configuration
    float* config = new float[size]();

    // first, let the chain contract:
    // each element is attracted
    // by its neighbours
    for(long i = 0; i < size; i++)
        config[i] = 0.5*previous[i]
            + 0.25*previous[(i-1) % size]
            + 0.25*previous[(i+1) % size];

    // actual random walk step
    stochastic_unit_step(size, config);

    // shift such that the average is zero
    shift_centre_to_origin(size, config);

    return config;
}
```

Example file: memleak.zip

Code that produces a memory leak

```
int main(int argc, char** argv) {
        [...]
        // configuration: an array of float numbers
        float* present_configuration
                = new float[size]();
        [...]

        for(long i = 0; i < steps; i++) {
                [...]
                // do the random walk step
                present_configuration
                        = crw::step(size,
                                present_configuration);
                float present_elongation
                        = crw::elongation(size,
                                present_configuration);
                [...]
        }
        delete[] present_configuration;
}
```

```
float* crw::step(long size, float previous[])
{
        // allocate the next configuration
        float* config = new float[size]();

        // first, let the chain contract:
        // each element is attracted
        // by its neighbours
        for(long i = 0; i < size; i++)
                config[i] = 0.5*previous[i]
                        + 0.25*previous[(i-1) % size]
                        + 0.25*previous[(i+1) % size];

        // actual random walk step
        stochastic_unit_step(size, config);

        // shift such that the average is zero
        shift_centre_to_origin(size, config);

        return config;
}
```

Example file: memleak.zip

How should the “memleak” code be fixed?

We know that there is a memory leak.
Lucky situation: We also know where in the code it comes from.

Discussion:

- What approaches can we try in general, when we have detected a memory leak? (Brainstorm a list of ideas.)
- Do they look promising as solutions for the present case?

There are a few techniques in C++ that help us write safer code with explicit memory management, still done on the heap but less prone to pitfalls.

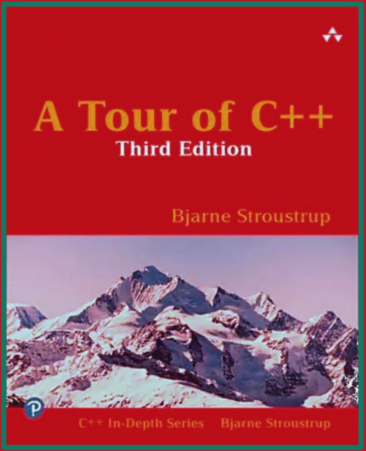
The key concept for safe manual memory management is **ownership** of a data item, *i.e.*, deciding what entity/part of the code has *responsibility for managing its allocation and deallocation* safely. The entity holding ownership is typically an object – so we will first need to discuss how OOP is done in C++.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

2 C++ basics



book Section 1.6

The keywords const, constexpr, and auto

auto: Leave it to the compiler to determine the type

This requires an initialization.

Remark:

typeid(x).name() can be used to output the type assigned to x.

```
for (auto i = 0; i < 26; i++)
{
   auto c = 'a';
   c += i;
   cout << c;
}
```

should become **int**

should become **char**

const: Used to declare an <u>immutable</u> variable

constexpr: Immutable and, additionally, can be <u>evaluated at compile time</u>

`constexpr int space_dimension = 3;  int n = 0;  cin >> n;  const int num_coords = n*space_dimension;`

Con.1: By default, make objects immutable

"make objects non-const only when there is a need to change their value"

Con.4: Use **const** to define objects with values that do not change

Con.5: Use **constexpr** for values that can be computed at compile time

“const” parameters of a function

If we pass an argument by reference but do not intend to modify it, the parameter should be declared as **const**. Such as:

```
void do_something(const int N);
void do_something(const int& N);
void do_something(const int* const x);
void do_something(const int x[]);
```

Const variables may only be passed by reference if the parameter is also const.

1. What is the code supposed to do?

2. Why does it not compile?

3. What should be changed?

4. What more const/-expr can we add?

```
int second_of(int N, int* x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; i < N; i++)
                if(x[i] > largest) {
                        second_largest = largest;
                        largest = x[i];
                }
                else if(x[i] > second_largest)
                        second_largest = x[i];
        return second_largest;
}

int main() {
  int fixed_array_size = 5;
  const int x[fixed_array_size] = {4, 0, 6, 5, 2};
  int t = second_of(fixed_array_size, x);
}
```

Making proper use of const and constexpr

1. Variables are declared as const if we do not plan to modify them after initialization. If their value can be determined at compile time, we can even use "constexpr".

2. We declare pointers to const (of type T) as const T*. (References as const T&.)

3. Pointers that are constant (*i.e.*, have as value an address that cannot be changed) are of the type T* const. This can can be combined with the above: const T* const.

Solution: If x[] in main() is a const int array, or even "constexpr" (which is even stronger than "const"), we must make the x parameter in second_of() const int*.

All other changes are nice, but optional.

const-array.cpp – mistake fixed as follows:

```
int second_of(const int N, const int* const x) {
        int largest = x[0];
        int second_largest
                = std::numeric_limits<int>::min();

        for(int i = 1; i < N; i++)
                if(x[i] > largest) {
                        second_largest = largest;
                        largest = x[i];
                }
                else if(x[i] > second_largest)
                        second_largest = x[i];
        return second_largest;
}

int main() {
   constexpr int fixed_array_size = 5;
   constexpr int x[fixed_array_size] = {4, 0, 6, 5, 2};
   const int t = second_of(fixed_array_size, x);
}
```

"const", pass by reference, and const pointers

1) If you can pass by value, that is always to be preferred!
2) If you pass an argument by reference, the compiler assumes that the function will modify it. Write "const" whenever that's not the case.

An array is a pointer. Therefore **it is impossible to pass an array by value**. If you don't intend the function to write to the array, it should be a const parameter.

Pay attention to C++ syntax for combining pointers with "const". Illustration:

```
int v = 3;
const int x[3] = {1, v, v*v};       // x is an array of constant integers
const int* y = &x[1];               // y is a pointer to a constant integer
int* const pv = &v;                 // pv will forever point to address of v
const int* const z = &x[2];         // z will forever point to address of x[2]

(*pv)++;                            // this is legal, we may change *pv, just not pv
y++;                                // this is legal, we may change y, just not *y
```

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Second worksheet

INF205

19th February 2024

Second worksheet & tutorial statistics

On Fagpersonweb, there are now 37 registered course participants.
Out of these, 28 submitted the first worksheet.
Out of these, 6 presented solutions at the first tutorial session.

Based on these values, we now need eight problems per remaining worksheet.

Second worksheet schedule:

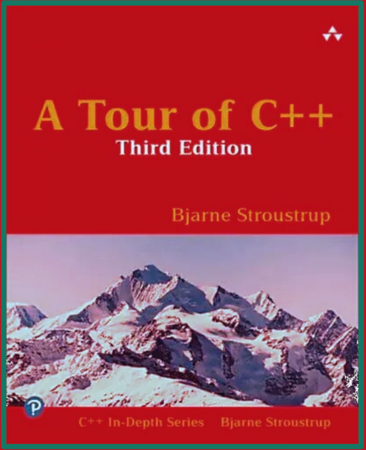
- Monday, 19th February: The worksheet is introduced at today's lecture.
- Wednesday, 21st February: Tutorial session for working on the problems.
- Monday, 26th February: Assigning presentation slots.
    - It looks like the booking system works, we can continue to use it.
- Tuesday, 27th February: Submission deadline.
- Wednesday, 28th February: Presentation of solutions at the tutorial.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

2 C++ basics



book Chapter 8

Standard libraries (and other common libraries)

In programming with C/C++ libraries, we have already seen:

- How to work with the C standard library, *e.g.*, …
- … with the C++ standard library, *e.g.*, …

1. What library includes were we using in today's examples? What for?

   … (discuss)

2. What other C/C++ libraries have you been using? Do you recommend them?

   … (discuss)

Standard libraries (and other common libraries)

In programming with C/C++ libraries, we have already seen:

- How to work with the C standard library, *e.g.*, <cassert>, <cstring>, …
- … with the C++ standard library, *e.g.*, <iostream>, <string>, …

Technically, libraries are pre-compiled object code that can be reused.

The library needs to be accessed at three stages:

- At **compile time**, we need to **include** the **library headers**.
  - The complete source code for the libraries is unnecessary.
  - It is even possible for the library to be coded in another language.
- During **linking**, the object code is **dynamically linked** against the library.
  - At this stage, the library "**static object**" (*.so file) is needed.
  - The executable does not contain the library's object code!
- At **execution time**, the executable and the library are **loaded jointly**.
  - If the library's **static object** code is gone now, the code will not run!

C++ standard template library

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

```
// declare a list of int values
std::list<int> my_list();
```

```
// declare a list of std::string objects
std::list<std::string> my_list();
```

- **vector**<T> is a **dynamic array** for type **T** elements, similar to Python lists.
- **deque**<T> ("double ended queue"): **Dynamic array** with capacity both ends.

- **forward_list**<T> is a **singly linked list** data structure for type **T**.
- **list**<T> is a **doubly linked list** data structure for type **T**.

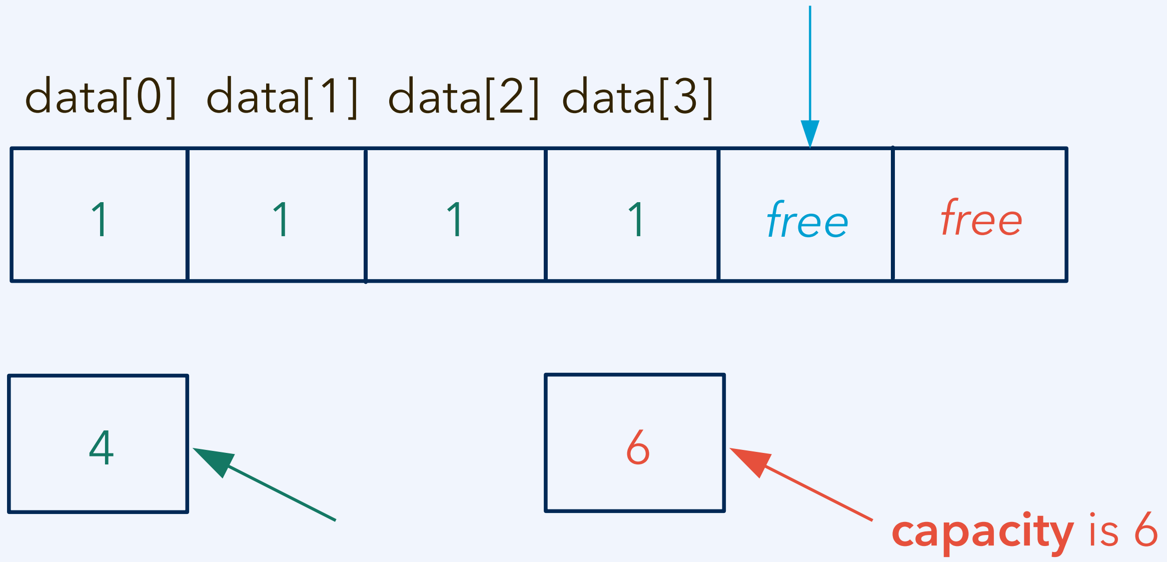
- **set**<T> is a container where each **key** (element) occurs only (at most) once.
- **map**<T, V> contains **key**-**value** pairs, which each key occurring at most once.
- **multimap**<T, V> contains **key**-**value** pairs; keys may occur multiple times.

- **array**<T, **n**> is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

STL vector: Dynamic array in C++

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

A dynamic array can be declared (with "#include <vector>") as an object of the parameterized class **vector**<**T**>, *e.g.*, "**vector**<**int**> data = {1, 2, 3, 4};".



Functionalities of the **STL vector** include explicit addressing with "[index]" notation, and many more.

STL, object orientation, containers, and templates

It is **good style** to **use the STL containers** (vectors, lists, *etc.*).
They are implemented to deal with memory safely.

We will look more into them, and how to build such data structures ourselves, once all the required concepts have been introduced.

- **vector**<**T**> is a **dynamic array** for type **T** elements, similar to Python lists.
- **deque**<**T**> ("double ended queue"): **Dynamic array** with capacity both ends.

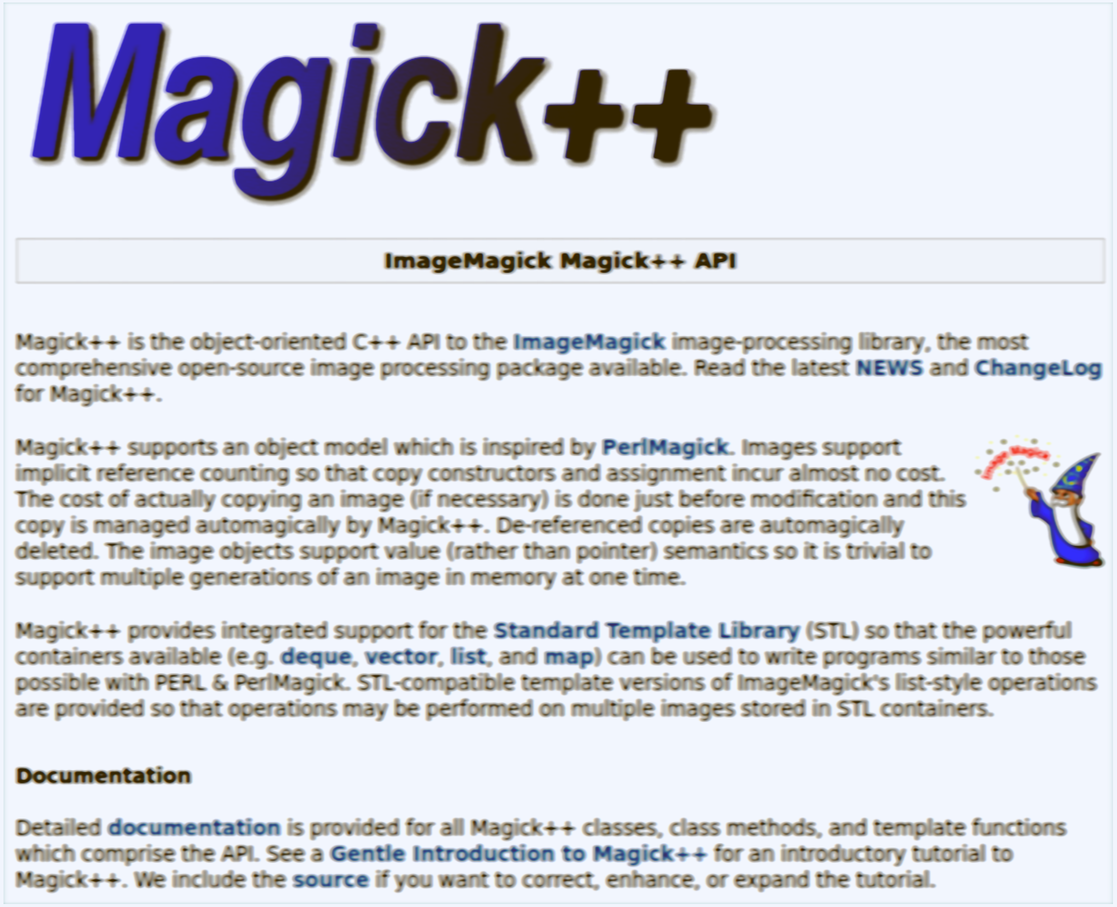
- **forward_list**<**T**> is a **singly linked list** data structure for type **T**.
- **list**<**T**> is a **doubly linked list** data structure for type **T**.

- **set**<**T**> is a container where each **key** (element) occurs only (at most) once.
- **map**<T, **V**> contains **key**-**value** pairs, which each key occurring at most once.
- **multimap**<T, **V**> contains **key**-**value** pairs; keys may occur multiple times.

- **array**<T, **n**> is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

Dynamic (shared) library use: Example

As an example, let us compile and run a code using Magick++ (**ImageMagick**).



ImageMagick Magick++ API

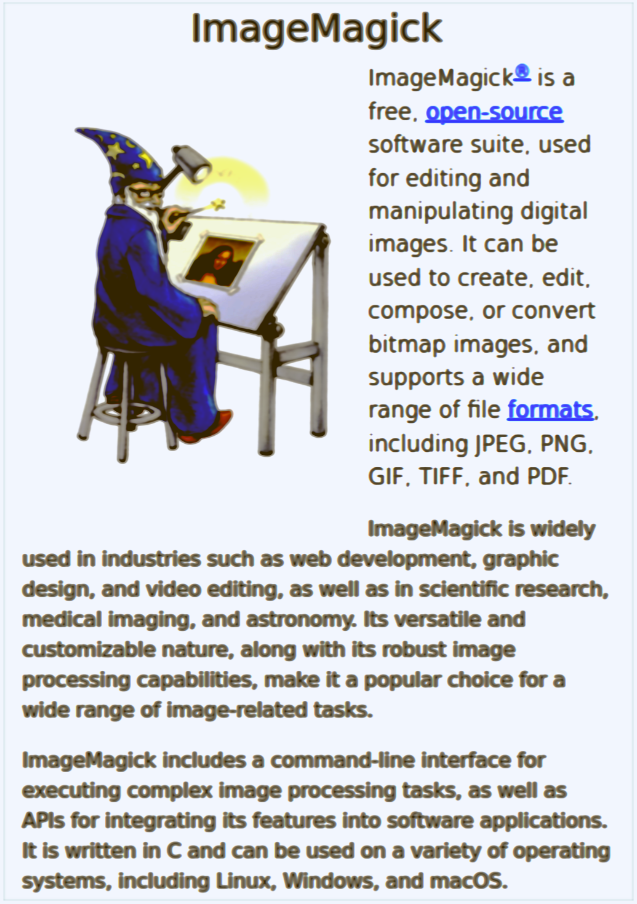
Magick++ is the object-oriented C++ API to the ImageMagick image-processing library, the most comprehensive open-source image processing package available. Read the latest NEWS and ChangeLog for Magick++.

Magick++ supports an object model which is inspired by PerlMagick. Images support implicit reference counting so that copy constructors and assignment incur almost no cost. The cost of actually copying an image (if necessary) is done just before modification and this copy is managed automagically by Magick++. De-referenced copies are automagically deleted. The image objects support value (rather than pointer) semantics so it is trivial to support multiple generations of an image in memory at one time.

Magick++ provides integrated support for the Standard Template Library (STL) so that the powerful containers available (e.g. deque, vector, list, and map) can be used to write programs similar to those possible with PERL & PerlMagick. STL-compatible template versions of ImageMagick's list-style operations are provided so that operations may be performed on multiple images stored in STL containers.

Documentation

Detailed documentation is provided for all Magick++ classes, class methods, and template functions which comprise the API. See a Gentle Introduction to Magick++ for an introductory tutorial to Magick++. We include the source if you want to correct, enhance, or expand the tutorial.



ImageMagick

ImageMagick is a free, open-source software suite, used for editing and manipulating digital images. It can be used to create, edit, compose, or convert bitmap images, and supports a wide range of file formats, including JPEG, PNG, GIF, TIFF, and PDF.

ImageMagick is widely used in industries such as web development, graphic design, and video editing, as well as in scientific research, medical imaging, and astronomy. Its versatile and customizable nature, along with its robust image processing capabilities, make it a popular choice for a wide range of image-related tasks.

ImageMagick includes a command-line interface for executing complex image processing tasks, as well as APIs for integrating its features into software applications. It is written in C and can be used on a variety of operating systems, including Linux, Windows, and macOS.

Dynamic (shared) library use: Example

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>

int main(int argc, char** argv)
{
  // charmap input
  std::ifstream pixistrm(argv[1]);
  diskgraphics::Charmap cm;
  pixistrm >> cm;
  pixistrm.close();

  // image object setup
  Magick::InitializeMagick(*argv);
  Magick::Image img(Magick::Geometry(cm.get_sizex(), cm.get_sizey()), "white");
  img.magick("BMP");
  img.monochrome();
  img.type(Magick::BilevelType);

  // pixel-by-pixel transfer of content
  for(int x = 0; x < cm.get_sizex(); x++)
    for(int y = 0; y < cm.get_sizey(); y++)
      img.pixelColor(x, y, Magick::Color(cm.get_pixel(x, y) == 0? "black": "white"));

  // output in BMP format using one bit per pixel
  img.quantize(2);
  img.write(argv[2]);
}
```

read a "Charmap" object from a pixel graphics file, using an ad-hoc format

ImageMagick can deal with many file formats; we need an uncompressed pixel graphics format such as BMP

copy colour value of pixels

with quantize(2) we get one bit per pixel

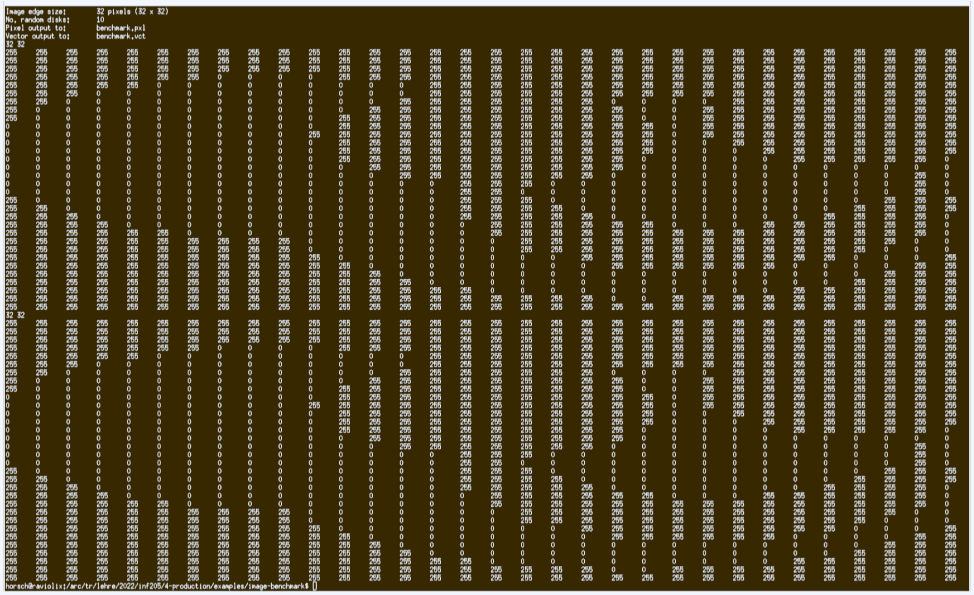
[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

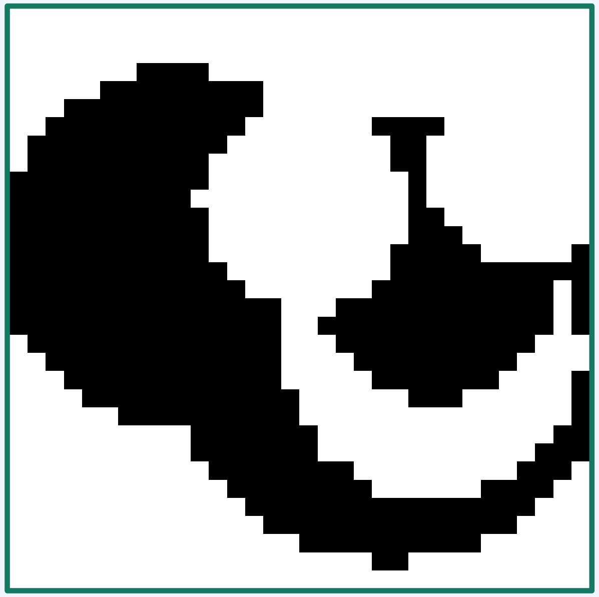
Dynamic (shared) library use: Example

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

#include <Magick++.h> ← How does the compiler know where to look for this file?



```
Image edge size:        32 pixels (32 x 32)
No. random disks:       10
Pixel output to:        benchmark.pxl
Vector output to:       benchmark.vct
32 32
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 0   0   0   0   255 255 255 255 255 255 255 255
255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255
255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   0   0   255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   255 255 255 255 255 0
0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   255 0
255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255
255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 0   0   0   0   0   0   0   0   0   255 255 255 255
255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 0
255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 0   0   0   255 255 255 255 255 255 0
255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0
255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 0   0
255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 0   0   0
255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   0   255
255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   255 255 255 255 255 255 0   0   0   0   255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
32 32
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 0   0   0   0   255 255 255 255 255 255 255 255
255 0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255
255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 0   255 255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 0   0   0   255 255 255 255 255 255 255
0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   255 255 255 255 255 0
0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   255 0
255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 0   0   0   0   0   0   0   0   0   0   0   255 255 255
255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 0   0   0   0   0   0   0   0   0   255 255 255 255
255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 0
255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 0   0   0   255 255 255 255 255 255 0
255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0
255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 255 0   0
255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 255 255 255 0   0   0
255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   255 255 255 255 255 255 255 255 255 0   0   0   255
255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   255 255 255 255 255 255 0   0   0   0   255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   0   0   0   0   255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   0   0   0   0   0   0   0   0   255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 0   0   255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
horsch@raviolix:/arc/tr/lehre/2022/inf205/4-production/examples/image-benchmark$ 
```



[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Compiling and linking with libraries

The **convert-to-bmp** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>
```

How does the compiler know where to look for this file?

For this particular library, there is a tool that helps call g++ with the right flags:

```
g++ -c -std=c++17 -o <name>.o <name>.cpp `Magick++-config --cppflags`
```

↓

```
-fopenmp -DMAGICKCORE_HDRI_ENABLE=1
-DMAGICKCORE_QUANTUM_DEPTH=16 -I/usr/local/include/ImageMagick-7
```

```
g++ -std=c++17 -o <name> *.o `Magick++-config --libs`
```

↓

```
-L/usr/local/lib -lMagick++-7.Q16HDRI -lMagickWand-7.Q16HDRI -lMagickCore-7.Q16HDRI
```

More typically, you need to provide this information to the compiler by hand.

[1] Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2] Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Norwegian University
of Life Sciences

Creating a dynamic (shared) library

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

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

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

Conclusion

INF205

19th February 2024

Norwegian University
of Life Sciences

Weekly glossary concepts

What are essential concepts from this lecture?

Let us include them in the **INF205 glossary**.[1]

???

wild pointer

dynamic library

???

constant expression

???

[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

2 The C++ programming language

2.1 Features of C++
2.2 Pointers and arrays
2.3 Pass by value or reference
2.4 Memory allocation
2.5 Immutability and constants
2.6 Working with libraries