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

Structure of the course

1) Introduction (week 6)

- Getting started – the lecture last week.

2) The C/C++ programming language(s) (weeks 7 and 8)

- Essential features that make C/C++ different from Python; *e.g.*, dealing with memory allocation and deallocation explicitly, using pointers.

3) Data structures (weeks 9 to 11)

- Linked data structures, containers, C++ standard template library.
- Memory management for container data structures.

4) Concurrency (week 12 to 17)

- MPI and ROS2 for parallel programming and concurrent processes.

5) Production and optimization (week 18 and 19)

- Good practices and useful tools for programming projects.

Weekly glossary concepts

What are essential concepts from the previous lecture?

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

???

static array

compilation

???

procedural programming

???

[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

2.1 Features of C++

Functions / procedural programming

Norwegian University of Life Sciences

In many procedural programming languages, including C/C++ and Python, code blocks that can be called from other blocks are called **functions**. However, do not confuse **procedural programming** (as a programming paradigm) with **functional programming**, a name given to a very different approach (LISP, *etc.*).

- Functions are named
- Each function has a distinct task
- It may have its own variables
- It may call another function, including calls to itself (recursion),
- It may return a value; it must have a return type (which may be **void**)
- It may accept arguments
- Function **parameters** are the variables listed in the function's definition. Function **arguments** are the values passed to the function, which are assigned to the function's parameters at runtime.

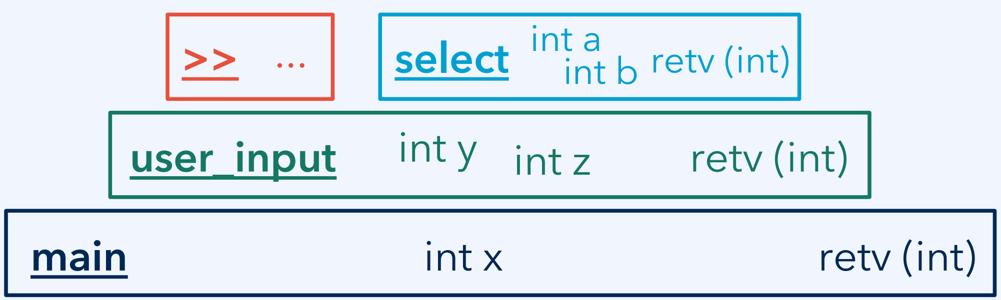
Functions and their stack frames

Stack-like memory management

When a function is called, a known amount of memory must be allocated for its variables (including parameters) "on top of the stack."

When the function returns, its memory can be released; the calling method and its variables become the top of the stack again.

The lifetime of local variables in a **stack frame** is limited to the function's runtime.




```
int select(int a, int b)
{
  if(a%2 == 0) return a;
  else return b;
}

int user_input()
{
  int y = 0, z = 0;
  std::cin >> y >> z;
  return select(y, z);
}

int main()
{
  int x = user_input();
}
```

Example file: three-functions.cpp; compile with "g++ -g3 -o ..." and run using gdb.

Observations: Stack

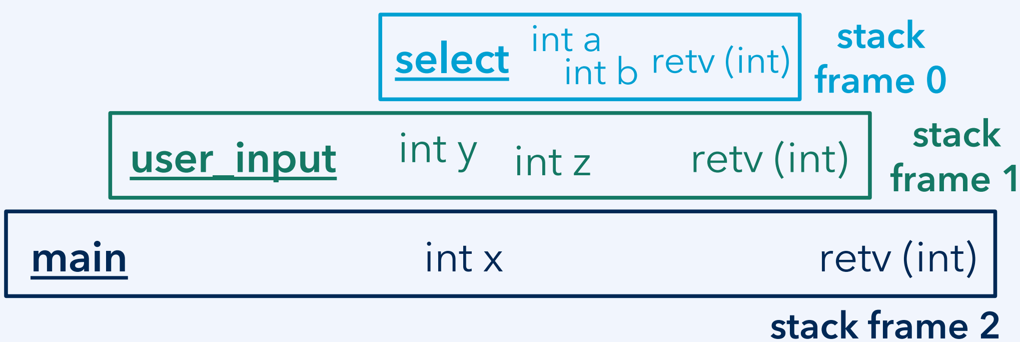
Backtrace and stack inspection using gdb

- Compile with "-g" or "-g3" option
- gdb three-functions
  - break three-functions.cpp:6
  - run

```
Breakpoint 1, select (a=4, b=3) at three-functions.cpp:6
6        if(a%2 == 0) return a;
```

  - bt ["backtrace"]

```
#0  select (a=4, b=3) at three-functions.cpp:6
#1  [...] user_input () at three-functions.cpp:14
#2  [...] main () at three-functions.cpp:19
```




```
1
2
3
4  int select(int a, int b)
5  {
6     if(a%2 == 0) return a;
7     else return b;
8  }
9
10 int user_input()
11 {
12    int y = 0, z = 0;
13    std::cin >> y >> z;
14    return select(y, z);
15 }
16
17 int main()
18 {
19    int x = user_input();
20 }
```

Overloading and namespaces

Function **overloading** (identical name within the **same namespace**, if any) and the use of **multiple namespaces** are technically different mechanisms. However, they become similar if equal names occur in multiple namespaces.

```
namespace task_a
{
  void run(double x, double y);
}
namespace
{
  void run(int x, int y);
}
-------------------------------
int main()
{
  using namespace task_a;
  run(1.0, 1.0);
}
```

```
namespace task_b
{
  void run(int x, int y);
  void run(double x, double y);
}
-------------------------------
int main()
{
  using namespace task_b;
  run(1.0, 1.0);
}
```

```
namespace task_c
{
  void run(double x, double y);
}
namespace
{
  void run(double x, double y);
}
-------------------------------
int main()
{
  run(1.0, 1.0);
  task_c::run(1.0, 1.0);
}
```

In what case are we strictly overloading "run" (within a single namespace)?

In each of the cases, which version of "run" will be executed?

Example file: namespaces-overloading.zip

C++ Core Guidelines

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance
- CP: Concurrency and parallelism
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CPL: C-style programming
- SF: Source files
- SL: The Standard Library

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

Selected guidelines on namespaces

SF.20: Use namespaces to express logical structure

Use of the "unnamed namespace" construction: **namespace{ … }**

- **SF.21:** Don't use an unnamed namespace in a header
- **SF.22:** Use an unnamed namespace for all internal/non-exported entities

(This makes it easy to distinguish "helper" code from that needed outside.)

```
void do_task_a(int x);
void do_task_b(int x);
void do_task_c(int x);
…
```

header file, *.h

```
namespace
{
   int transform(int x) { … }
}

void do_task_a(int x)
{
   int y = transform(x);
   …
}
```

code file, *.cpp

was declared in the header

Selected guidelines on functions

Core Guidelines on functions:

- F.1: “Package” meaningful operations as carefully named functions
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple

  …
- F.46: int is the return type for main()

I.6: Prefer **Expects()** for expressing preconditions

I.7: State postconditions [with **Ensures()**]

example based on Grimm’s book, p.443:

```
int area(int height, int width)
{
  Expects(height > 0);
  int retv = height*width;
  Ensures(retv > 0);
  return retv;
}
```

More traditional style uses **assert(…)**.

Example files: conditions-gsl.cpp (modern) and conditions-assert.cpp (traditional).

Selected guidelines on signed/unsigned integers

Core Guidelines style rules against "**unsigned**".
These rules use elements taken from the **Guidelines Support Library (GSL)**.

ES.102: Use signed types for arithmetic

ES.106: Don't try to avoid negative values by using "unsigned"

ES.107: Don't use unsigned for subscripts [*e.g.*, array indices], prefer **gsl::index**

The reasoning against a normal (signed) integer is that "**int** might not be big enough."

Except in the very rare occurrence where that could be the case, we can use int.

Remember the pitfall: For arithmetics over "unsigned" variables, the result of the subtraction "**2 – 3**" is the value **4 294 967 295**.

Noregs miljø- og biovitskaplege universitet

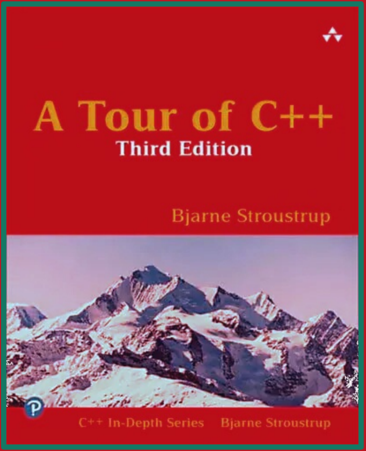
Institutt for datavitskap

Digitalisering på Ås

2 C++ basics

2.1 Features of C++

<u>2.2 Pointers and arrays</u>



Section 1.7

What is a pointer?

Compare:

- An **int** is a variable that contains an integer number, such as **7**.
- A **std::string** is a variable that contains a string, such as **"INF205"**.
- A pointer to X, of type **X***, is a variable that contains a memory address, such as **0x7ffeaea5174c**. It is meant for an address of a value of type X.
- It is good practice to set pointers to **nullptr** (“null pointer”) whenever it is impossible to assign them a valid memory address.
- We can **allocate** memory for an X object by hand, with **X* pt = new X**.
- We can **deallocate** (release) the memory again by hand, with **delete pt**.

A **pointer** is a variable that has a **memory address** as its value.

- double* b is a pointer to an address for storing a double value.
- The address of an object is obtained by **referencing**, e.g., pt = &var;
- While pt is the address, we can **dereference** it (*pt) to access the content.

Operators for referencing (&) and dereferencing (*)

Referencing operator &:

- Used to obtain the address of a variable: &x is the address of x.
- If x has type X, the address has the type X*, *i.e.*, "pointer to X."

```
int x = 5;  int* y = &x;
```

- A second, independent use of this operator is "passing a reference" as a function argument, *e.g.*, as in `void increment(int& x);`

Dereferencing operator *:

- If y is a pointer of type X* (pointer to X), the value of y is an address.
- To access the value stored at the address y, we dereference it as *y.
- The value stored at y, and accessed by *y, is then of type X.
- & and * are inverse operators, therefore, *(&x) is the same as x:

```
int x = 5;  int* y = &x;  cout << x << " is the same as " << *y;
```

Allocate with new, deallocate with delete

Allocation: Reserve memory to store data.
Deallocation: Release the memory.

On the stack

The stack is already handled completely and safely by the compiler. **Memory on the stack** (local variables of functions) is **allocated** as part of a **stack frame** **when the function is called**. It is **deallocated** again **when the function returns**.

On the heap

Memory on the heap is managed independent of the stack, at runtime, subject to **explicit allocation and deallocation** instructions that must come from the programmer. There is no garbage collection in C++!

- **Allocation** is done with **new**. Example: **int* i = new int(42);** (initialization to *i = 42)
- **Deallocation** is done with **delete**. Example: **delete i;**

C/C++ arrays are pointers

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. This supports fast access using **pointer arithmetics**. Once created, the size of a C/C++ array is fixed; we cannot append elements.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

x = &(x[0])   x + 3 = &(x[3])   x + 6 = &(x[6])

In C/C++, the type of an array such as **int[] is the same as the corresponding pointer type int***, *i.e.*, **the array actually is a pointer**. Its value is an address at which an integer is stored, namely, the memory **address of the first element**.

When x[i] is accessed, the compiler transforms this into x + sizeof(int) * i.

- **Allocation** is done with **new**. Example: **int* i = new int[8]();**
- **Deallocation** is done with **delete[]**. Example: **delete[] i;**

Summary: Allocation and deallocation of pointers

How do we declare a pointer?

- Like any other variable. Its type is a pointer type; *e.g.*, int* my_int_pointer;

How do we initialize a pointer?

- Initialize to **nullptr** (pointer version of 0): int* my_int_pointer = **nullptr**;
- Initialize to **another variable's address**: int* my_int_pointer = &my_index;
- **Allocate memory** on the heap: int* my_int_pointer = **new** int(0);

How do we deallocate a variable if it is stored on the heap?

- Delete the pointer to it. Example: b = **new** BookIndex; …; **delete** b;

How to release the memory if it is a local variable that is stored on the stack?

- Don't do that! You can only call "delete" on memory allocated with "new".

What if we call **new**, but there is not enough free memory left on the system?

- **new** VeryBigObject may throw an exception (a high-level construct).
- **new(std::nothrow)** VeryBigObject may return **nullptr** (low-level construct).

Summary: Allocation and deallocation of arrays

How do we declare a array?

- Give the size as constant expression in square brackets; *e.g.*, int values[6];
- Also possible: Just declare a pointer; *e.g.*, int* values;

How do we initialize an array?

- Explicitly give all the values: int values[ ] = {4, 2, 3, -7, 2, 3};
- Initialize to **all zeroes**, indicating the array size: int values[6] = { };
- **Allocate memory** with **default initialization**: int* values = **new** int[6]**()**;

How do we deallocate an array if it is stored on the heap?

- Use **delete[]**. Example: b = **new** BookIndex[100]**()**; …; **delete[]** b;
- **Pitfall:** If you use **delete** instead of **delete[]**, only b[0] will be deallocated!

What if we call **new**, but there is not enough free memory left on the system?

- **new** BigObject[100000]**()** may throw an exception.
- **new(std::nothrow)** BigObject[100000]**()** may return **nullptr**.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Tutorial scheduling

INF205

12th February 2024

Registration to present at the tutorial session

terminplaner DFN

ADD ▾ LOGIN ▾ LANGUAGES ▾

INF205 tutorial problem presentations, 14.2.2024

INF205 problems 1 to 7.

Problem 1. Basic tools Wednesday, 14. February 2024 - 14:15	BOOK
Problem 2. From C++ to Python Wednesday, 14. February 2024 - 14:20	BOOK
Problem 3. From Python to C++ Wednesday, 14. February 2024 - 14:25	BOOK
Problem 4. Size of the primary data types, in bytes Wednesday, 14. February 2024 - 14:30	BOOK
Problem 5. Using C/C++ arrays Wednesday, 14. February 2024 - 14:35	BOOK
6. Program analysis - termination of a recursive function Wednesday, 14. February 2024 - 14:40	BOOK
7. Program analysis - return value of a function Wednesday, 14. February 2024 - 14:50	BOOK

It is a mandatory activity to present once at the tutorial.

At present, we will have just enough problems to discuss so that everyone can present.

Therefore, all slots will be used – where nobody signs up, somebody will be chosen and announced in advance of the meeting.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

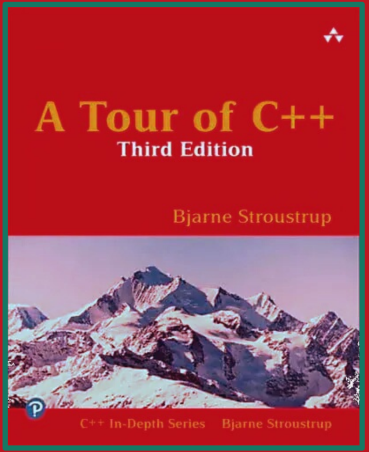
Digitalisering på Ås

2 C++ basics

2.1 Features of C++

2.2 Pointers and arrays

2.3 Pass by value/reference

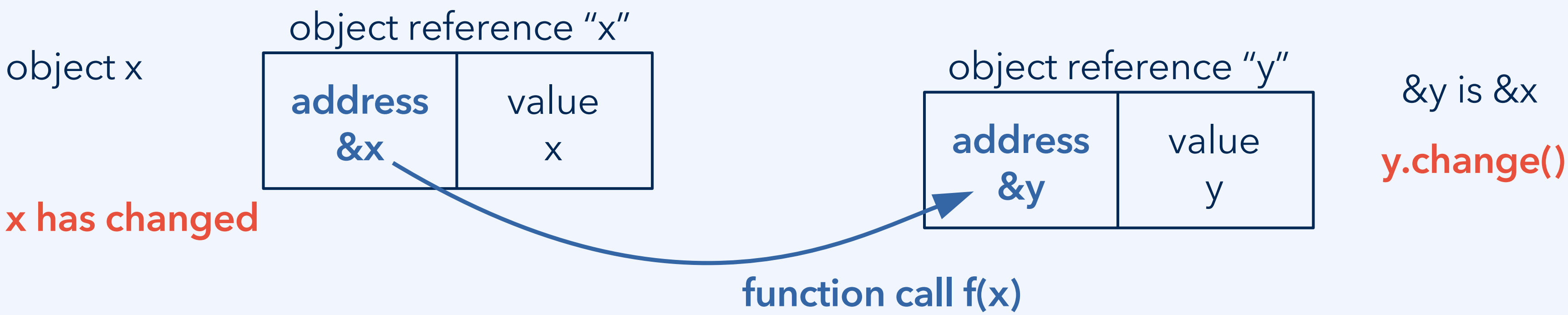


Sections 1.9, 3.6

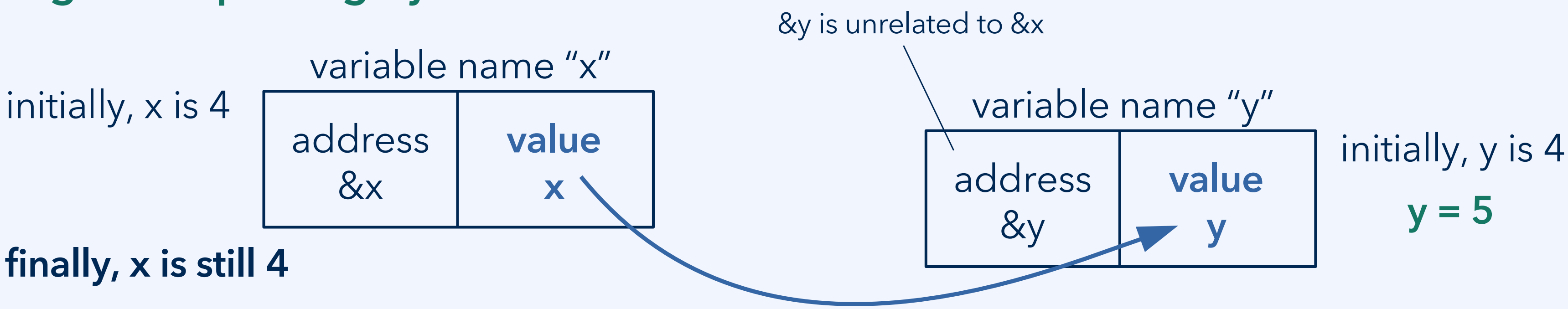
Pass by value in C++ (compared to Python)

In Python, object references are passed by value (*i.e.*, "pass by object reference"):

Argument passing by object reference in Python (similarly, in Java)



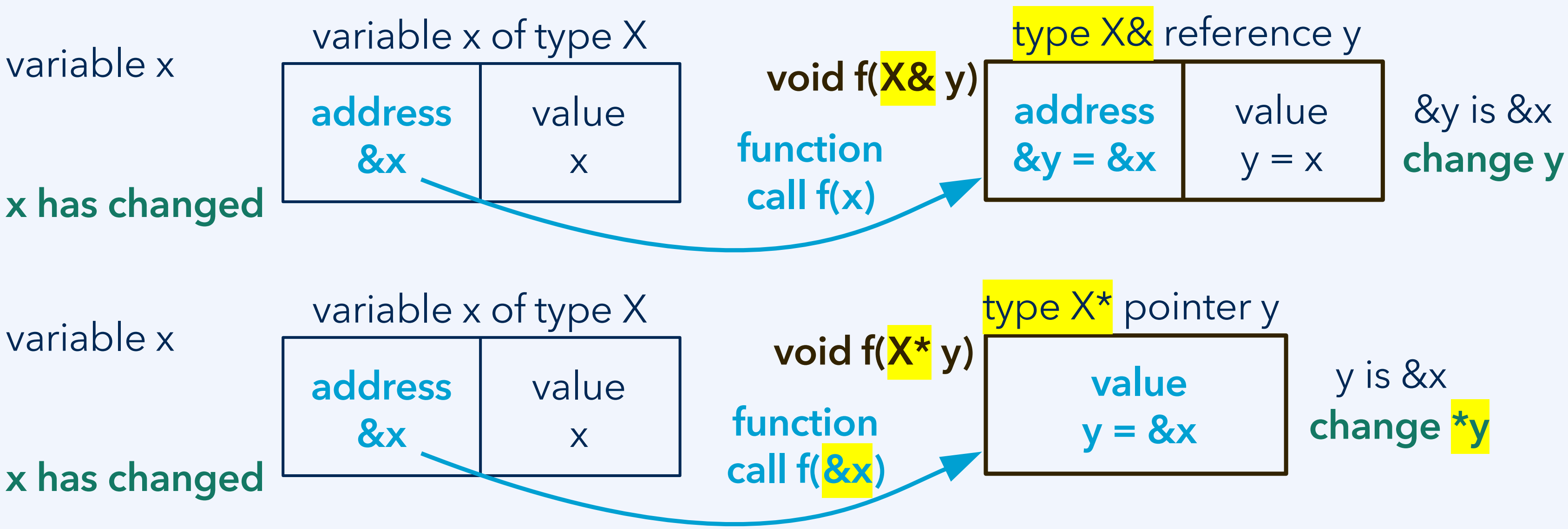
Argument passing by value



Pass by reference in C++ (compared to pass by value)

Pass by value: A new copy of the argument value(s) is created in memory. The function works with the copy. The function cannot access the original variable.

Pass by reference: The function is enabled to access the original variable at its address in memory. No copy is created. Changes affect the original variable. C++ has two mechanisms for this: **Passing a reference** and **passing a pointer**.*



*Unfortunately there is some terminology confusion about this. We will call both "**pass by reference**."

Pass by reference vs. pass by value

Advantages of **passing** a function argument **by value**:

- **Memory management** is done at the stack level, **by the compiler**. The programmer can relax and does not need to deal with this aspect.
- The stack can be optimized at compile time, and it is **faster to access** memory on the stack because there is no need to look up an address.
- **Variable lifetime** coincides with the runtime of functions that use them.
- The value of **the variable in the calling function is protected** from any intransparent changes by the called function.
- This makes the code **more modular**. It is easier to understand and even verify the function. (The point of using local instead of global variables.)

Advantages of **passing** a function argument **by reference**:

There must be a reason there is a second mechanism, pass-by-reference. Even Python uses it when dealing with objects. **Discussion:** What is the advantage?

Pass by reference using a pointer vs. a reference

Pointers and references are two equivalent notations for the same techniques.

```
void some_function(int& parameter) {
  …
  // convert the reference to a pointer
  int* y = &parameter;
  // now we can work with pointer y
  …
}
```

```
void some_function(int* parameter) {
  …
  // convert the pointer to a reference
  int& x = *parameter;
  // now we can work with reference x
  …
}
```

Advantages of pass-by-reference **using a reference**:

- Some memory-related errors become less likely if we only work with references; *e.g.*, errors from applying incorrect pointer arithmetics.
- Looks more like Java, Python, and other modern high-level languages.

Advantages of pass-by-reference **using a pointer**:

- It is visible to the programmer at all times that we deal with memory.
- Looks more like C, and it is closer to the object-code representation.

Norwegian University of Life Sciences

Remark: Strings in C and C++

The C++ language only prescribes what functionalities a **std::string** should provide, not how it is realized at the memory level, which is up to the compiler.

Most implementations remain close to that from the C language, where character arrays terminated by the null character '\0' are employed. (If you want to enforce this, you can also still use all the C style constructs explicitly.)

string s = "INF205"; or **char s[] = "INF205";** produce the following in memory:

'I'	'N'	'F'	'2'	'0'	'5'	'\0'
73	78	70	50	48	53	0

Also to ensure backwards compatibility with C, **string literals** between double quotation marks such as "INF205" are of the type **const char*** (not **std::string**). Between single quotation marks there is always a **char**, such as **char x = 'a';**

Remark: Strings in C and C++

C++ strings may be the same as arrays at the memory level, but they are not arrays to the language. Therefore, **it is possible to pass C++ strings by value**.

C strings, however, **can never be passed by value** because they are arrays.

```
void increment_at(int p, char* str)
{
   str[p]++;
}

int main()
{
   char c_style_str[] = "INF205";
   increment_at(5, c_style_str);
   cout << c_style_str << "\n";
}
```

```
void increment_at(int p, std::string str)
{
   str[p]++;
}

int main()
{
   std::string cpp_style_str = "INF205";
   increment_at(5, cpp_style_str);
   cout << cpp_style_str << "\n";
}
```

Example file: string-argument-passing.cpp

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Conclusion

INF205

12th February 2024

Weekly glossary concepts

Norwegian University of Life Sciences

What are essential concepts from this lecture?

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

namespace

???

???

pointer

reference

???

[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