Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

I C++ basics

- I.5 Fundamental data types
- I.6 Scopes and namespaces
- I.7 Data at the memory level
- I.8 Toward object orientation in C/C++

 14th September 2022

Lecture recordings

Information from Studieavdelingen – Learningcenter

"I have now set up **automatic recording and live streaming of INF205** in TF1-102 every Wednesday […] the recording will start at 14:15. It will also be stopped in the break 15:00-15:15 so that everyone can move around in the room without being recorded. The streamings and recordings will automatically be available for the students in the Canvas room under the Panopto Video-button. […] the lecturer needs to remember […] to use the microphone so that the audio is captured. […]

If there are other people than yourself appearing in the recording, you must **convey the information in the list below** to them. […]

- That **NMBU will be recording**.
- The purpose of the recording ([…] teaching/lectures […] in question).
- Where the recording is stored and shared (**Panopto Video via Canvas**).
- For how long the recording is stored. [until the next iteration of INF205]
- Where the recording is published (Panopto Video via Canvas).
- Who has access to the recording (students and teachers in the course).
- Where the audience can sit to avoid being recorded ([…] areas […] not captured by camera).
- How to ask questions and get replies without being recorded (*e.g.* ask their questions in a break or send them in through alternative channels as e-mail or Canvas).
- The basis of treatment for the recording ([…] **consent** […] **may be withdrawn at any time**)"

Glossary terms

void

C/C++ data type name keyword designating "no data type"

compile(r)

translate(s) human-readable source code into a lower-level representation by which it becomes more machine-actionable

procedural programming

Programming paradigm based on procedures (in C/C++, **functions**) as its highest-level device for structuring code and the program control flow.

function

Block of code with parameters, (and parameter types) and possibly a return type.

Programming paradigms

Imperative programming

- It is stated, instruction by instruction, what the processor should do
- Control flow implemented by jumps (**goto**)

Structured programming

- Same, but with **higher-level control flow**
- Contains "instruction by instruction" code

Procedural programming

- **Functions** (procedures) as **highest-level structural unit** of code
- Still contains loops, *etc.*, for control flow within a function

Object-oriented programming

- **Classes** as **highest-level structural unit** of code; objects instantiate classes
- Still contains functions, *e.g.*, as methods

Programming paradigms based on **describing the solution** rather than computational steps:

Functional programming
(also: "declarative programming")

Logic programming

Constraint programming

Functions / procedural programming

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.

Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.5 Fundamental data types

Tutorial example

```python
# output Fibonacci numbers smaller/equal to x
# return True if x is a Fibonacci number,
# False if it is not
#
def print_fibo_until(x):
    n = 1
    fibo_n = 1
    fibo_previous = 0

    while x >= fibo_n:

        print(n, fibo_n, sep="\t", end="\n")

        fibo_next = fibo_n + fibo_previous

        n += 1
        fibo_previous = fibo_n
        fibo_n = fibo_next

    return x == fibo_previous


y = 17711
if print_fibo_until(y):
    print(y, "is a Fibonacci number")
else:
    print(y, "is not a Fibonacci number")
```

```cpp
#include <iostream>
using namespace std;

/* output Fibonacci numbers smaller or equal to x
 * return true if x is a Fibonacci number, false if it is not */
bool print_fibo_until(int x)
{
  int n = 1;
  int fibo_n = 1;
  int fibo_previous = 0;

  while(x >= fibo_n)  // could also become a for loop
  {
    cout << n << "\t" << fibo_n << "\n";

    int fibo_next = fibo_n + fibo_previous;

    n += 1;
    fibo_previous = fibo_n;
    fibo_n = fibo_next;
  }
  return x == fibo_previous;
}

int main()
{
  int y = 17711;
  if(print_fibo_until(y)) cout << y << " is a Fibonacci number.\n";
  else cout << y << " is not a Fibonacci number.\n";
}
```

Fundamental data types in C/C++

int

- the default signed integer type

float and **double** (also, **long double**)

- Single-precision and double-precision floating point numbers

short (int), long (int), long long (int)

- less/more memory and smaller/larger range of values

unsigned, unsigned short (int), unsigned long (int), …

- holds natural number (or zero); modulo-arithmetic applies: $-n = 2^k - n$

bool

- integer-like; *meant to* hold the value **false** (0) or **true** (1, or any value ≠ 0)

char, wchar_t

- integer-like; *meant to* hold a ASCII (char) or Unicode (wchar_t) character

declaration only (be careful)

```
int n;
```

declaration with initialization (recommended)

```
int n = 0;
```

Style advice: Prefer int over unsigned

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
I.6: Prefer **Expects()** for expressing preconditions
I.7: State postconditions [with **Ensures()**]

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

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;
}
```

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.

auto and const(expr) keywords

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** (auto i)

should become **char** (auto c)

const: Used to declare an immutable variable

constexpr: Immutable and, additionally, can be evaluated at compile time

```
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

Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.5 Fundamental data types

1.6 Scopes and namespaces

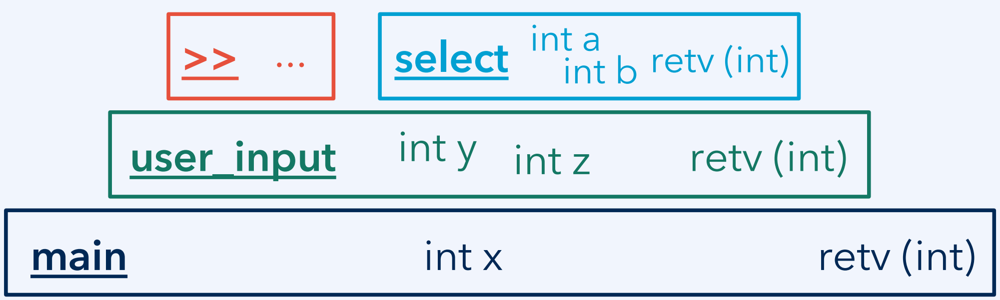
Functions and their stack frames

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();
}
```

Namespaces and overloading

Norwegian University of Life Sciences

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);
}
```

```
namespace task_b
{
  void run(int x, int y);
  void run(double x, double y);
}
```

```
namespace task_c
{
  void run(double x, double y);
}
namespace
{
  void run(double x, double y);
}
```

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

Namespaces and overloading

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?

Core 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.)

cannot have been declared 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);
  …
}
```

was declared in the header

code file, *.cpp

Example problem: Use of "auto"

What data types will the compiler use below where it says auto?

1) float y = 2.5; const auto x1 = y;
2) auto x2 = 2;
3) const auto x3 = x1*x2;
4) auto x4 = 'C';
5) auto x5 = x3 + x4;
6) auto x6 = x4++;
7) auto x7 = ++x1;
8) auto x8; std::cin >> x8;

This may depend on the compiler!

Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.5 Fundamental data types

1.6 Scopes and namespaces

1.7 Data at the memory level

Observations: Data types

The realization of C/C++ fundamental data types in memory, including their size, can be machine-dependent and even compiler-dependent.

The keyword **sizeof** is used to obtain the size of a variable in memory, in bytes.

Example (check **datatype-size.cpp**)

size of **int**:	4	(int)-1	= -1
size of **short**:	2	(short)-1	= -1
size of **long**:	8	(long)-1	= -1
size of **long long**:	8	(long long)-1	= -1
size of **unsigned short**:	2	(unsigned short)**-1**	= **65535**
size of **unsign. long long**:	8	(unsigned long long)**-1**	= **18446744073709551615**

Observations: Data types

The realization of C/C++ fundamental data types in memory, including their size, can be machine-dependent and even compiler-dependent.

The keyword **sizeof** is used to obtain the size of a variable in memory, in bytes.

Example (check **datatype-size.cpp**)

size of **int**:	4	(int)-1	= -1
size of **short**:	2	(short)-1	= -1
size of **long**:	8	(long)-1	= -1
size of **long long**:	8	(long long)-1	= -1
size of **unsigned short**:	2	(unsigned short)-1	= 65535
size of **unsign. long long**:	8	(unsigned long long)-1	= 18446744073709551615
size of **bool**:	1	(bool)**-1**	= **1**
size of **char**:	1	(char)-1	= �
size of **wchar_t**:	4	(wchar_t)-1	= -1
size of **float**:	4	(float)-1	= -1
size of **double**:	8	(double)-1	= -1
size of **long double**:	16	(long double)-1	= -1

Observations: Stack

Stack-based memory allocation is simple and safe:

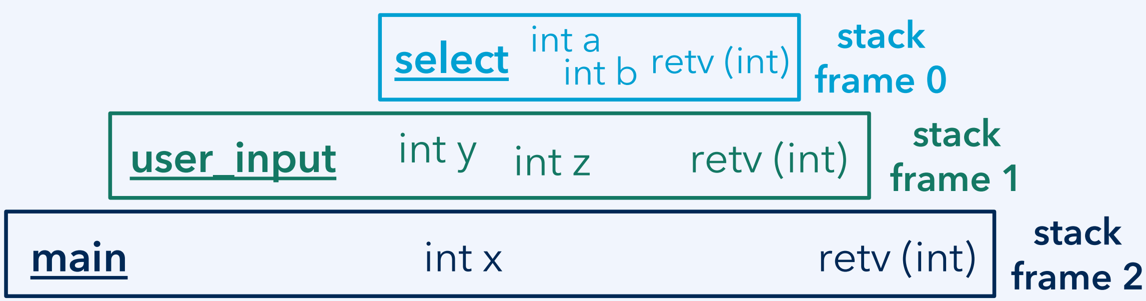
- Memory handling and optimization can be done at compile time, by the compiler
- Variable lifetime ends and memory is released automatically by removing the top element (frame 0) from the stack
- No need for an explicit deallocation of memory by the programmer
- No need for an automatic garbage collection running in the background

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

addresses of a, b, and return value (as offset, e.g., from the top of the stack) are known at compile time

compiler knows where y and z need to be written to and where the return value of select can be read from

```
int user_input()
{
  … select(y, z); …
}
```



Observations: Stack

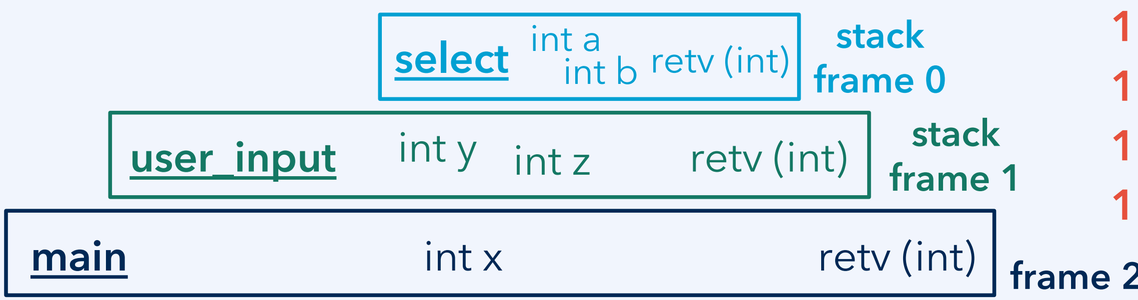
Backtrace and stack inspection using gdb

- 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();
   }
```

Noregs miljø- og biovitskaplege universitet

1 C++ basics

1.5 Fundamental data types

1.6 Scopes and namespaces

1.7 Data at the memory level

1.8 Toward OOP in C/C++

Structures: Object orientation from C

The structure (**struct**) was introduced as a device for OOP-like programming in the C language. It is retained in C++, where it is seen as somewhat archaic.

Structures group different *variables* (**properties**) together into one **composite data type**. In C, that is all they do. Each instance of a structure is called an **object**; each object has its own instance of the structure's properties. In C++, a structure can also have **methods**, *i.e.*, *functions* with the structure name as a prefix that are called and carried out for an individual object.

Structures are what comes closest in C/C++ to object orientation from Python: Structure elements are accessible from outside the structure. They're "**public**".

Example:

- A book indexation structure, giving a position in a book in terms of chapter number, section number, and page number.
- Methods can be used for going to the next chapter, section, page.
- But we can also directly set chapter and section numbers from outside.

Example: struct syntax and use

```
struct BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();
  int next_section();
  int next_page();

  void out();
};
```

header file contains
method declarations

```
int BookIndex::next_chapter()
{
  chapter++;
  section = 1;
  page++;
  return chapter;
}

...

void BookIndex::out()
{
  std::cout
    << "Section " << chapter
    << "." << section
    << ", p. "<< page << "\n";
}
```

code file contains the
method implementation

Each BookChapter object contains three int variables:

BookChapter i

- int i.chapter
- int i.section
- int i.page

BookChapter j

- int j.chapter
- int j.section
- int j.page

Norges miljø- og biovitenskapelige universitet

INF205

Resource-efficient programming

I C++ basics

I.5 Fundamental data types

I.6 Scopes and namespaces

I.7 Data at the memory level

I.8 Toward object orientation in C/C++