Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

1 C++ basics

Static arrays

Arrays in C/C++ are static: When declaring the array, the array size is specified and the exact amount of memory required for these data items is allocated. The array size does not change over time.

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

Accessing elements of an array is highly efficient: When x[i] is accessed, the compiler transforms this into accessing the memory address x + sizeof(int) * i.

How do we declare a array?

- Give the size as constant expression in square brackets; *e.g.*, `int values[6];`

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] = { };`

C strings: Character arrays

In C++, there is an explicit std:string datatype. But since C++ is backwards compatible to C, there is also the more traditional string type: The char array.

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

Note that while the string length above is six, one more is allocated in memory. The array has seven elements: It ends with the **null character '\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';**

C/C++ arrays are pointers

See example create-simple-segfault.cpp

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;**

What is a pointer?

See example create-simple-segfault.cpp

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;
```

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;
```

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

1 C++ basics

1.4 Design by contract

1.5 Formal program analysis

1.6 On procedural programming

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

Design by contract

- Specify
  - Function specification – what it should do
  - Non-functional specification – how well it should do it
- Design
  - Select appropriate algorithms and data structures
    - Consider effectiveness/correctness – *does it do what it is supposed to?*
    - Consider efficiency
      - Size
      - Speed
- Implement
  - Create solution at low level
- Evaluate
  - Debug, assess for syntactic & semantic correctness
  - Check performance (i.e., resource requirements)

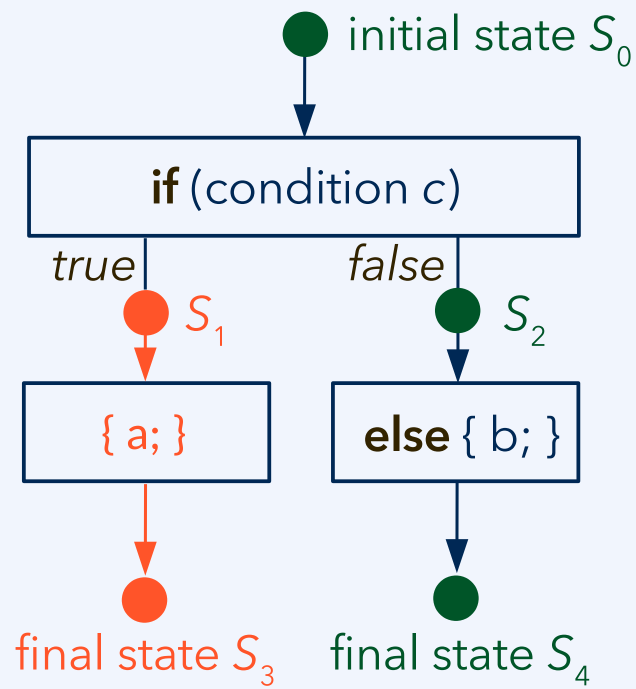
"contracts" between specifying and implementing person

(these often are the same person)

Preconditions and postconditions

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.



Note

Consider the statement "a" from transition $S_1 \rightarrow S_3$:

- Execution state S_1 fulfils the **precondition** of statement a.
- Execution state S_3 fulfils the **postcondition** of statement a.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

1 C++ basics

1.4 Design by contract

1.5 Formal program analysis

1.6 On procedural programming

Program flow graphs

Formal analysis goes the opposite way, constructing conditions for states.

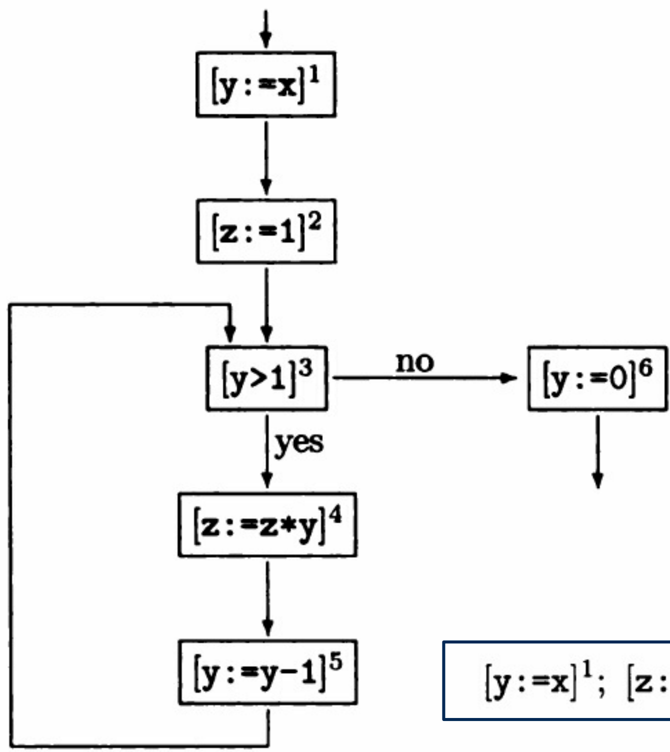


Figure 1.2: Flow graph for the factorial program.

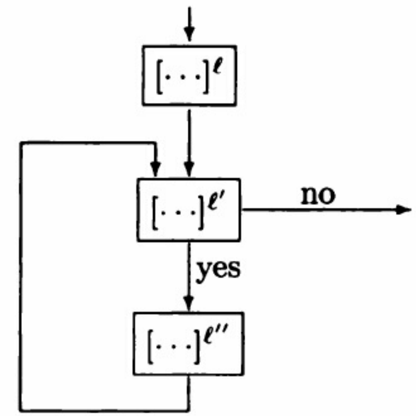


Figure 2.2: A schematic flow graph.

$[\texttt{y:=x}]^1$; $[\texttt{z:=1}]^2$; **while** $[\texttt{y>1}]^3$ **do** ($[\texttt{z:=z*y}]^4$; $[\texttt{y:=y-1}]^5$); $[\texttt{y:=0}]^6$

[1]F. Nielson, H. Riis Nielson, C. Hankin, *Principles of Program Analysis*, Heidelberg: Springer, **2005**.

Program flow graphs: Formal analysis

For purposes of formal analysis, the program flow is analysed step by step, e.g., at the instruction (statement) level, at the level of blocks of code that form a coherent unit, or at the level of functions or methods.

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

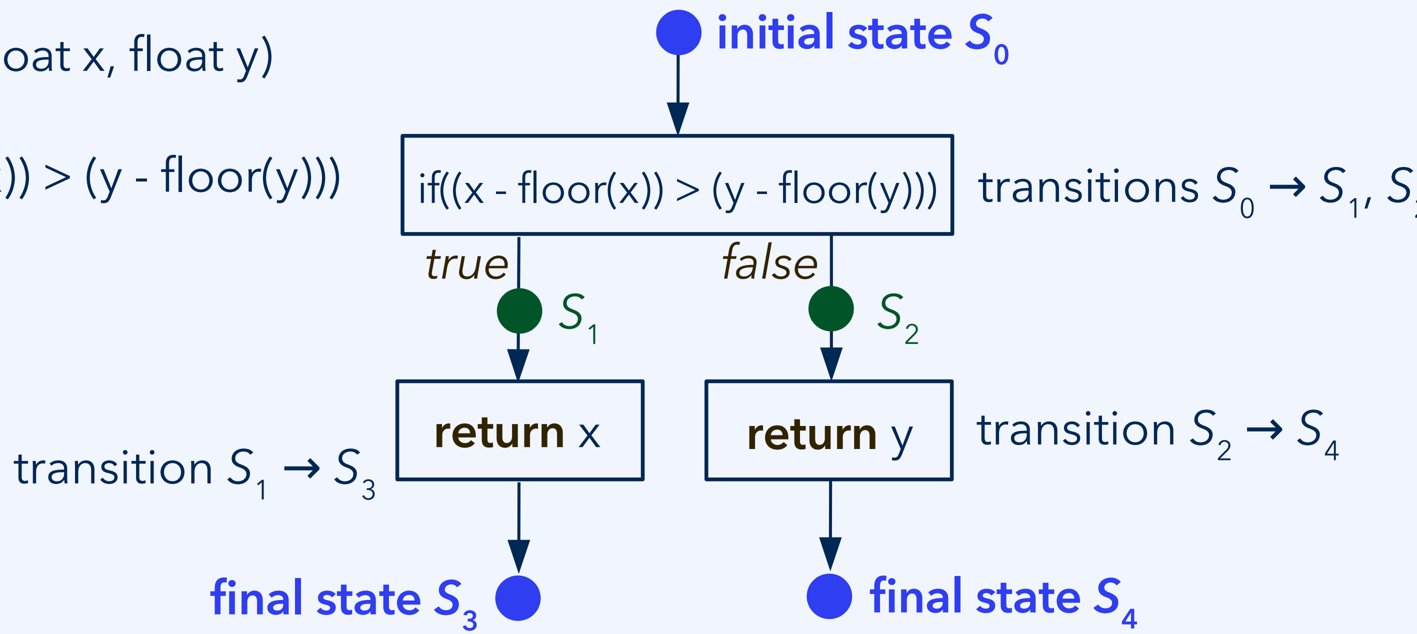
Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.

Example

As part of a development project, we need a function grtfrac(x, y) that takes **two floating-point arguments** and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6". **In design by contract, the caller, not the called method needs to guarantee the precondition.**

Program flow graphs: Formal analysis

```
float grtfrac(float x, float y)
{
  if((x - floor(x)) > (y - floor(y)))
    return x;
  else
    return y;
}
```



S_0: x and y are floating-point numbers (by specification).

S_1: x, y as above; the fractional part of x is greater than that of y.

S_2: x, y as above; the fractional part of y is greater than that of x, or equal.

S_3: The fractional part of x is the greater one, and x was returned.

S_4: The fractional part of y is greater (or they are equal); y was returned.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

1 C++ basics

1.4 Design by contract

1.5 Formal program analysis

1.6 On procedural 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.

Memory on the stack vs. memory on the heap

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;**

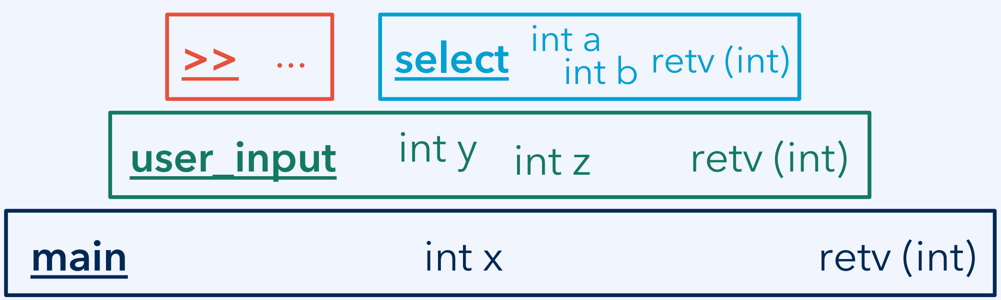
Functions and their stack frames

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

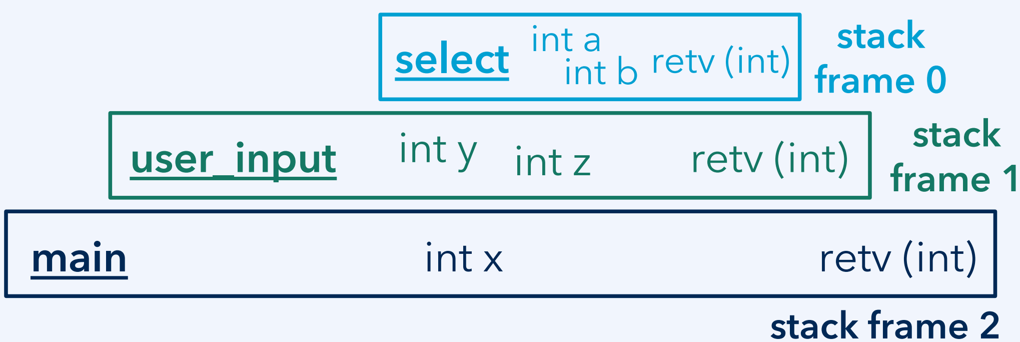
Backtrace and stack inspection using gdb

- 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, 2022, 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: Signed/unsigned

Core Guidelines style rules against "**unsigned**" (same as **unsigned int**).
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."

Then rather use **long** (instead of **unsigned int**) …

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

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

1 C++ basics

1.1 Why C++
1.2 From Python to C++
1.3 Static arrays

1.4 Design by contract
1.5 Formal analysis
1.6 Stack frames