CONTENTS

1. Functions
2. std::string
3. std::vector<typename T>
4. Containers
5. Pointer and reference types
Notion of a function

- “A function declaration introduces the function name and its type. A function definition associates the function name and type with the function body.” [en.cppreference.com]

- Example: a function that computes the maximum of two integer values
  - Declaration
    ```
    int max(int, int);
    int max(int a, int b); // or with formal parameter names
    ```
  - Definition
    ```
    int max(int a, int b) {
      if (a >= b) { return a; }
    
      return b; // observe, that we do not need ‘else’ here
    }
    ```
  - Some languages allow function definition only (e.g. Java)
  - We will learn why function declarations are useful in the next lecture
What is a function?

- A function is a little machine
  - Gets some input
  - Manipulates input
  - Returns output
  - Think of it as a functional unit!
- Similar to a mathematical function
Mathematical functions and C++

- **Task**
  - Declare a function \( f \) that is able to sum two numbers \( x, y \in \mathbb{N} \)
  - Define this function \( f \) to actually sum two numbers \( x, y \in \mathbb{N} \)

- **Declaration in mathematics**
  - \( f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \)

- **Definition in mathematics**
  - \( f(x, y) \mapsto x + y \)

- **Declaration in C++**
  - `unsigned f(unsigned, unsigned);`

- **Definition in C++**
  - `unsigned f(unsigned x, unsigned y) { return x + y; }`

- **Note** `unsigned` is a shorthand for `unsigned int`
Functions in C++

- **Note**
  - A function may not return
  - A function may receive no parameters
    ```cpp
    void f() {} // void is a "special" type → no type
    void g(int a);
    void h(void);
    int returnOne() { return 1; }
    ```

- Functions should have a “meaningful” name (unlike mathematical functions)
  - General rule: name things according to their purpose, same holds for variables!

- Function’s input and output can be …
  - Built-in types
  - User-defined types (today and next time)
Functions in C++

- Lets define a function
- Why you should use meaningful names:

```cpp
int function(int x, int y) {
    int result = x;
    for (int i = 2; i <= y; ++i) {
        result *= x;
    }
    return result;
}
```

- What is the value of `result` after the function call?
  - `int result = function(2, 4);`
  - 16

- What does the function do?
  - Implements the power function

- What would be a better declaration?
  - `int pow(int base, int exponent);`

- Note this function “only works” for integers!
  - Don’t try `int result = pow(2.5, 4.8);`
    - Significant figures get cut off (type casting)
Use of functions

- Use a function to
  - perform a logical task
    - that has to be performed multiple times
      → don’t repeat yourself
  - build an abstraction / generalization
  - structure your source code

- The task described by a function can be reused!
  - Faster development
  - Less error prone
  - Improved readability
  - **Use libraries:** a collection of useful functions

```c
int pow(int base, int exponent) {
    int result = base;
    for (int i = 2; i <= exponent; ++i) {
        result *= base;
    }
    return result;
}
```
Use of functions

- Let’s consider the factorial function!
- Sequential

```c
int factorial(int n) {
    int f = n;
    while (n-- > 1) {
        f *= n;
    }
    return f;
}
```

- What is that?

```c
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n-1);
    }
    return 1;
}
```

- Computes the factorial function using recursion!
Conditional assignments and the ternary operator

- If an assignment depends on a condition you can use a shortcut

```c
int i = ... // some value
int variable;
if (i > 10) {
    variable = 100;
} else {
    variable = 0;
}
int variable = (i > 10) ? 100 : 0; // shorthand which does the same
```

- Note there are many of these short forms
  - `c++;`
  - `d += 10;`
  - `unsigned // shorthand for unsigned int`
  - You will get used to it
Recursion

- With functions one can make use of recursion!
- “Recursion occurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic. The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition.” [en.wikipedia.com]
- Another recursive definition of recursion: “Recursion, see recursion!”

- A recursive function uses itself to solve a task
- A function exhibits recursive behavior if
  1. it defines one (or more) base case(s) that do not use recursion
  2. a set of rules that reduce all other cases towards the base case
Factorial function revisited

```c
int factorial(int n) {
    if (n > 1) { return n * factorial(n-1); }
    return 1;
}
```

- What happens if `factorial` gets called?
  ```c
  int result = factorial(5);
  ```
  - Let's see what happens:
    ```c
    factorial(5)
    if (5 > 1) return 5 * factorial(4);
    factorial(4)
    if (4 > 1) return 4 * factorial(3);
    factorial(3)
    if (3 > 1) return 3 * factorial(2);
    ```
    - If you are still not convinced have a look at:
      - What on Earth is Recursion? – Computerphile
    - Recursion often allows for elegant solutions
    - Requires some time to get used to
  ```c
  factorial(2) if (2 > 1) return 2 * factorial(1);
  ```
  ```c
  factorial(1)
  if (1 > 1) NO!
  ```
  ```c
  return 1;
  ```
  We have reached the base case!
  The call to `factorial(5)` can now evaluate
  ```c
  5 * 4 * 3 * 2 * 1 = 120
  ```
Functions

- You can now divide your computations into logical pieces (functions)
- The OS calls the `main` function for you
- In `main` you can call whatever you like

```c
int main() {
    int i = factorial(5);
    int j = factorial(6);
    return 0;
}
```

```c
int factorial(int n) {
    return (n > 1) ? n * factorial(n-1) : 1;
}
```
A note on functions

- With `constexpr` we effectively have to versions:
  - a `constexpr` version
  - a non-`constexpr`-version

  // can be evaluated at compile time
  `constexpr` int i = factorial(8);

  int x = ... // non-constant x
  // can only be evaluated at run time
  int j = factorial(x);

- Actual parameters passed to a function are copied by default!
- Inside a function you work on copies by default!

  ```
  int increment(int x) { return ++x; }
  int x = 10;
  int y = increment(x); // y is now 11
  // x is still 10
  ```

- Remember `constexpr`

  // C++11 allows one return statement
  `constexpr` int addNumbers(int a, int b) {
    return a + b;
  }
  // C++14 allows more than one statement
  `constexpr` int factorial(int n) {
    int result = 1;
    while (n-- > 0) {
      result *= n;
    }
    return result;
  }

  ```
  © Heinz Nixdorf Institut / Fraunhofer IEM
A note on functions

- Function calls come with some costs in terms of performance
  - Safe registers’ contents, put function arguments on the stack, increment stack pointer, …, restore registers, perform jump back
  - But usually that is not why your code is slow!
- If high performance really matters, compiler can inline small functions
  - A function call is replaced by copying the functions body to the call site
  - Use the keyword `inline` to give the compiler some hints
    ```c
    inline int add(int a, int b) { return a + b; }
    // a call to add()
    int c = add(10, 20);
    // may be replaced with
    int c = 10 + 20;
    ``
- Inlining is only necessary in rare cases (sometimes you make it worse)
  - Compiler inlines on its own if compiler optimizations are turned on (-Ox flag, where x is 1, 2 or 3)
Local and global variables

- Local variables are only accessible within a certain function / scope (e.g. main)
- A variable is local if it is defined inside a function
- Example
  ```cpp
  int main() {
      int i = 42;
      int j = 13;
      std::cout << i << 'n';
      std::cout << j << 'n';
      return 0;
  }
  ```
  - So far we only used local variables

- Global variables are accessible across functions (and modules)
- A variable is global if it is not defined within a function
- Example
  ```cpp
  int i = 10;
  double d = 1.234;
  void printGobals() {
      std::cout << i << 'n';
      std::cout << d << 'n';
  }
  ```
  ```cpp
  double addGobals() {
      return i + d;
  }
  ```
A note on global variables

- Try to avoid global variables as much as possible
  - You rarely need them
  - They break local reasoning
    - It becomes pretty hard to understand the code
  - It is hard to parallelize code that heavily makes use of globals
User-defined types / non-built-in data types

- Two very important user-defined types
  - `std::string`
  - `std::vector<typename T>`
- Implemented in the standard template library (STL)
- Vector is perhaps the most used non-built-in data type

- You can define your own data types
  - Use `class` or `struct` keyword
  - Next lecture!
Why should you use `std::string` in C++?

C has no built-in string datatype

- In C a string is stored in an `array` of characters
  ```cpp
cchar str[] = "Hello, World!";  
std::cout << str << '\n';  
int i = 0;  
while (str[i] != '\0') {  
  std::cout << str[i] << '\n';  
  ++i;  
}

cchar *ptr2str = "Hello, World!";  
char data[10] = "Hi!";  
```

- Such character arrays are (hopefully) terminated with `\0`
  - Which you can’t see directly

Remember built-in arrays are dangerous

- What if you forget the size of that array?
- What if you lose `\0` or have multiple `\0` in your character array through incorrect string processing?
- You risk reads and writes outside your array
  - Undefined behavior / buffer overflows
  - Please watch this video
  - [Buffer overflow attack](#)

C++ has no built-in strings either

- But it offers a safe wrapper: `std::string`
std::string

- Use the `#include <string>` header file
- `std::string` allows you to store strings
- `std::string` offers a lot of useful functionalities as well
  - Functionalities are offered as member functions (member functions: next lecture)
- `std::string` can grow and shrink dynamically (dynamic memory allocation: next lectures)
- `std::string` knows its size as well, unlike simple built-in arrays!
- `std::string` automatically adds the terminal character `\0`
- No buffer overflows!
- For the complete list of functionalities see
  - [http://en.cppreference.com/w/cpp/string/basic_string](http://en.cppreference.com/w/cpp/string/basic_string)
std::string

- The design is so good, it can be used like an ordinary built-in type (C++ is powerful)

- Example

```cpp
// create a string from string literal
std::string str = "Hello World!";
// copy str to other
std::string other = str;
// get str’s size
std::cout << str.size() << 'n';
// replace a single character
str[4] = 'O';

// append some more characters
str += "some more characters";
// extract a substring
std::string hello = str.substr(0,5);
std::string yetanother = "Hello";
// check for equality
std::cout << (hello == yetanother) << 'n';
```
std::vector<typename T>

- Again built-in arrays are dangerous for several reasons
- std::vector<typename T> is a safe wrapper for built-in arrays (similar to std::string)
- std::vector<typename T> can store multiple elements of the same type in sequence
- It is mutable and can grow and shrink dynamically (dynamic memory allocation: next lectures)
- Ok fine, but what is this <typename T>?
  - This is called a template parameter
  - Templates and template metaprogramming? (in the next lectures)
  - What are templates used for?
    - Allow for writing code that is independent of the type! (Cannot be done in the C language)
    - A vector can store any type!

```cpp
vector<int> ivec = {1, 2, 3};
vector<double> dvec;
vector<std::string> svec = {"Hello", "World", "!"};
```
std::vector<
typename  T>

- How to initialize (or construct) a vector?
- Example

```cpp
std::vector<int> ivec;          // call to default constructor
std::vector<int> ivec(10);      // call to constructor
std::vector<int> ivec(10, 42);  // another constructor
std::vector<int> ivec{1, 2, 3, 4, 5};  // yet another constructor
std::vector<int> ivec = {1, 2, 3, 4, 5};  // even more
```

- A vector can be constructed using one of its constructors
- All user-defined data types have constructors
  - A constructor’s job is to construct a variable / an object
    - Acquires resources and initializes correctly
  - Constructors are special member functions (next lecture)
**std::vector**<typename T>

- **std::vector** is designed such that it can be used like a built-in type
- **Example**

```cpp
std::vector<int> ivec = {1, 2, 3};
std::cout << "size: " << ivec.size() << '\n';
ivec.push_back(42);
ivec.push_back(120);
std::cout << "size: " << ivec.size() << '\n';
for (int i : ivec) {
    std::cout << i << ' ';
}
std::cout << '\n';
```

- **Note:** we are using members functions (next lecture)
  - Members can be data (variables) or functions → data members / function members
  - Members can be accessed with the . (point) operator
Type aliasing

- Introduce type aliases
  - using the typedef or using keyword
  - Prefer using (modern version)
  - as types get more complicated
  - to stride towards more flexible programs

- typedef double real_t;
- using ivec = vector<int>;

- Dealing with types decltype(*) (this is a C++11 feature)
  - * can be a variable / expression / function
    - const int i = 13;
    - decltype(i) x = 10;

- x has now i’s declared type (which is const int)

- A “real world example”

  // oh dear
  std::vector<std::pair<std::string,int>> v;
  // better use an alias for that
  using vpsi_t = std::vector<std::pair<std::string,int>>;

  // you can declare variables of that type
  vpsi_t x; // easier to read and write
What are containers?

- `std::vector<typename T>` is a container
- A container can store a bunch of data
- Containers are generic
  - Use one or more template parameters
  - Can hold values of any type
- Use different containers for different purposes
- Choose the right container depending on your problem
- Note that you can nest containers!
  - `std::vector<std::vector<double>>` matrix = {{1, 2}, {4, 5}};
STL containers?

- Sequence containers
  - array  // fixed size array
  - vector  // flexible size array
  - deque  // double-ended queue
  - forward_list  // singly linked list
  - list  // doubly linked list

- Associative containers
  - set  // unique element set
  - map  // unique element associative storage
  - multiset  // non-unique element set
  - multimap  // non-unique element associative storage

- Unordered associative containers
  - unordered_set  // hash set
  - unordered_map  // hash map
  - unordered_multiset  // ...
  - unordered_multimap  // ...

- Container adaptors
  - stack  // stack adaptor
  - queue  // queue adaptor
  - priority_queue  // priority queue adaptor

- STL containers ...
  - are quite useful
  - are implemented very efficiently
  - are accessible by including their header file
When to use what?

- Sequence containers

```cpp
// fixed size array
std::array<int, 4> a = {1, 2, 3, 4};
std::cout << a.size() << '\n';
for (int i : a) {
    std::cout << i << ' ';
}
// flexible size array
std::vector<int> b = {1, 2, 3, 4};
std::cout << b.size() << '\n';
for (int i : b) {
    std::cout << i << ' ';
}
b.push_back(5);
b.push_back(6);
```

- Rarely used:
  - forward_list // singly linked list
  - list // doubly linked list

- Associative containers

```cpp
// unique element set
std::set<int> c = {1, 2, 3};
c.insert(5);
c.insert(6);
if (c.count(5)) {
    std::cout << "set contains '5'.\n";
}
// unique element associative storage
std::map<int, std::string> d;
d.insert(std::make_pair(1, "A"));
d.insert(std::make_pair(2, "B"));
d[3] = "C";
std::cout << d[2] << '\n';
```

- You may wish to use their unordered counterparts
Containers in action

- Use STL vector to represent mathematical vectors \( \in \mathbb{R}^n \)
- `std::vector<typename T>`  // use `#include <vector>`
- Task: create two vectors to represent vectors from maths and write a function that calculates the scalar product!
  - \( x, y \in \mathbb{R}^3 \)
  - The scalar product \( \langle \cdot , \cdot \rangle \) is defined as
    - \( \langle a, b \rangle = \sum_{i=0}^{n} a_i \cdot b_i \)
  - Solution in C++
    ```cpp
    std::vector<double> x{1, 2, 3}; // call the initializer_list constructor
    std::vector<double> y{4, 5, 6}; // call the initializer_list constructor
    ```
  - We now have two vectors \( x \) and \( y \) filled with some floating-point numbers
Containers in action

- \(<a, b> = \sum_{i=0}^{n} a_i \cdot b_i\)
- A function that computes the scalar product

```cpp
double scalar_product(std::vector<double> x, std::vector<double> y) {
  double scalar_prod = 0;  // create a variable holding the result
  if (x.size() != y.size()) {  // handle that error */ }  // check dimensions
  for (size_t i = 0; i < x.size(); ++i) {
    scalar_prod += x[i] * y[i];  // multiply the entries and sum up to result
  }
  return scalar_prod;  // return the result
}
```

- More on error handling later on
Containers in action

- Data
  ```cpp
  std::vector<double> x{1, 2, 3};
  std::vector<double> y{4, 5, 6};
  ```

- Function to manipulate data (computes scalar product)
  ```cpp
double scalar_product(std::vector<double> x, std::vector<double> y) {
  double scalar_prod = 0; // create a variable holding the result
  if (x.size() != y.size()) { /* handle that error */ } // check dimensions
  for (size_t i = 0; i < x.size(); ++i) { // iterate over vectors’ entries
    scalar_prod += x[i] * y[i]; // multiply the entries and sum up to result
  }
  return scalar_prod; // return the result
}
```

- `double s = scalar_product(x, y);`
  - `s` is 32
More on types: pointer, reference, and value types

- Take a deep breath!

- What makes C++ so powerful?
  - Full control over resources (e.g. memory)

- Three “kinds / versions” of types exist in C++
  - “Normal”/value integer type
    \[
    \text{int } i = 42;
    \]
  - Pointer to an integer type
    \[
    \text{int } *j = \&i;
    \]
  - Reference to an integer type
    \[
    \text{int } \&k = i;
    \]
  - Makes C++ very powerful
  - Pointers and references are types that store addresses
    - Think of them as “pointers” (points-to graphs)
More on types: pointers

- Pointers, references, addresses?
- Every variable has a memory address
  - Think of houses (= variables)
  - People live in houses (= values)
  - Every house has a house number (= address)

```c
int *i_ptr; // i_ptr can store an address to an int
double *d_ptr; // d_ptr can store an address to a double
float *f_ptr = nullptr; // f_ptr is initialized with a null-pointer: f_ptr points to nothing!
```

```c
int i = 42; // integer initialized with 42
int *j = &i; // j holds the address of i (or points to i), & is the address of operator here
int *k; // uninitialized pointer to an integer
k = &i; // let k point to i
int **l = &j; // l holds the address of j
```
More on types: pointers

- Pointers, references, addresses?
- Every variable has a memory address
  - A mail man can deliver letters and parcels
    - You can also find a person using his address

```
int i = 42;
int *j = &i;  // get i’s address, this is called referencing (we create a pointer / reference)
*j = 100;    // modify i’s value through its address, this is called dereferencing
int k = *j;  // obtain i’s value through its address, this is called dereferencing
```
More on types: pointers

- Pointers, references, addresses?
- Every variable has a memory address

```cpp
int i = 42;
int *j = &i; // get i's address, this is called referencing (we create a pointer / reference)
int k = *j; // obtain i's value through its address, this is called dereferencing
std::cout << &i << '\n';
std::cout << i << '\n';
std::cout << &j << '\n';
std::cout << j << '\n';
std::cout << &k << '\n';
std::cout << k << '\n';
```

<table>
<thead>
<tr>
<th>Variable's name</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0x7ffab7c4770</td>
<td>42</td>
</tr>
<tr>
<td>j</td>
<td>0x7ffab7c4778</td>
<td>0x7ffab7c4770</td>
</tr>
<tr>
<td>k</td>
<td>0x7ffab7c4774</td>
<td>42</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
More on types: pointers

- Important
  - A pointer might be null
    - `int *i = nullptr;`
  - Meaning: the address does not exist / there is no address / `i` points to nothing
  - Don’t dereference a `nullptr`!
  - A pointer can be checked for `nullptr`
    ```
    if (i == nullptr) { cout << "i holds the null pointer\n"; }
    ```
  - Or if you wish to pretend to be cool
    ```
    if (!i) { cout << "i holds the null pointer\n"; }
    ```
More on types: pointers

- Things to remember
  - Declare a pointer type using *
  - Take an address of a variable with &
  - Dereference a pointer with *
  - A pointer variable may hold the null pointer `nullptr`
  - A pointer may dangle
    ```
    int *p;
    int q = *p; // please don't
    ```
- We will discuss techniques and tools to debug memory issues later on
More on types: references

- Example
  ```c
  int i = 42;
  int &j = i;
  ```
  - Declare a reference type by using &
  - “You can use j as if it was i”

- References behave much like pointers, but
  - Pointers can be re-assigned, references can not
  - Pointers can be null and are allowed to dangle
    - References always refer to a valid object
  - Pointer’s address can be taken, references addresses cannot be taken
  - Pointers allow for pointer arithmetic, references don’t (next lecture(s))
  - References are internally implemented as pointers
  - In general: references are much safer to use
References vs pointers

- When to use what and why do I need references and pointers?
  - References
    - Use references in functions’ parameter lists
    - See next slides
  - Pointers
    - Use pointers to implement algorithms and data structures (e.g. linked lists)
    - Use pointers for dynamic memory allocation
    - Next lecture(s)
Functions: parameter passing (and returning)

- How to pass and return huge amounts of data to and from a function?
- Consider a function that implements a matrix multiplication
  \[ \text{matrix \ matrixMult(matrix a, matrix b);} \]
  - Problem
    - If \text{matrixMult()} is called, actual parameters are \text{copied}!
    - Matrices can be huge, millions of elements \(\rightarrow\) copying may be very expensive
      - Processor is only copying data, rather than computing useful results
    - Can we avoid copying large data into functions?
    - \textbf{Pass data by reference, rather than by value!}
      \[ \text{matrix \ matrixMult(matrix\& a, matrix\& b);} \]
      - Matrices are not copied, we just pass a reference to a matrix (which is an address)
      - Matrix references can be used as if they were the matrices within the function’s body
Functions: parameter passing (and returning)

matrix matrixMult(matrix& a, matrix& b);

- Problem
  - Caution: If we modify the references \(a\) and \(b\) within the function we are changing the actual matrices
  - How can we avoid accidental changes made to the matrices \(a\) and \(b\)?
    - Use `const` references to avoid modifications
      
      ```cpp
      matrix matrixMult(const matrix& a, const matrix& b);
      ```

      - Changes made to `const` references result in compiler errors
  - How to return results if data to be returned is very large?
    - Return by reference?
      
      ```cpp
      matrix& matrixMult(const matrix& a, const matrix& b);
      ```

      - No! Return by value, compilers use return value optimization (RVO)!
    - Use:
      
      ```cpp
      matrix matrixMult(const matrix& a, const matrix& b);
      ```
Functions: parameter passing (and returning)

- If your data is small (e.g. built-in types such as `int`)
  - Pass and return by value (copy data)

- If you do not know the size upfront (e.g. in case of containers) or deal with huge data
  - Pass by reference (data itself stays where it is, no unnecessary copying)
  - Use `const` if you do not wish to modify the data within the function
  - Return by value (since all modern compilers support RVO)
Recap

- Functions
- Recursion
- Conditional assignments
- constexpr functions
- inline functions
- Local and global variables
- `std::string` and `std::vector<typename T>`
- STL containers
- Containers in action: scalar product
- Values, pointers, references
- Parameter passing
Thank you for your attention
Questions?