C++ Programming

Lecture 2
Software Engineering Group

Philipp D. Schubert
Contents

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

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

- Example: maximum function
  - Declaration
    
    ```cpp
    int max(int, int);
    int max(int a, int b); // or with formal parameter names
    ```
  
  - Definition
    
    ```cpp
    int max(int a, int b) {
        if (a >= b) return a;
        return b; // observe, that we do not need ‘else’ here
    }
    ```

- Some languages allow definition only (e.g. Java)

- We will learn about the use of declarations in the next lecture
What is a function?

- A function is a little machine
  - Gets input
  - Manipulates input
  - Returns output
  - Think of it as a functional unit!
- Very similar to a mathematical function

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 $x, y \in \mathbb{N}$
**Functions in maths & 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; }

- **Pretty much the same!**

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

- A function declaration in C++
  - `<return type> <function name> (<parameter list>);`
    - `<parameter list>` contains zero or more formal parameters
    - A function declaration is sometimes called a signature or function head

- A function definition in C++
  - `<return type> <function name> (<parameter list>) {
    <function body>
    <return type>
  }

- A function definition has to have a body
- Function body has to be enclosed by braces
Functions in C++

- Note
  - A function may return nothing!
  - A function may receive no parameters!
    ```cpp
    void f(); // void is a “special” type
    void g(int a);
    void h(void);
    int returnAlwaysOne() { return 1; }
    ```

- Unlike in mathematics a function should have a “meaningful” name
  - General rule: name things after their purposes, same holds for variables!

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

- Let's 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
Use of functions

- Use a function to
  - perform a logically task
    - that has to be performed multiple times
  - build an abstraction / generalization
  - structure your source code

- The task described by a function can be reused!
  - Programming becomes faster
  - Less opportunities for errors
  - More readable
  - 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 * fac(n-1);
    return 1;
}
```

- Compute the factorial function using recursion!
Conditional assignments

- If an assignment depends on a condition, you can use a shorthand:

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

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

- With the notion of a function one can use 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!”

Now for real

- 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 toward 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?
  ```
  int result = factorial(5);
  ```

- Let’s see what happens:
  ```
  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 elegant solutions
- Requires some time to get used to
Functions

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

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

int factorial(int n) {
    return (n > 1) ? n * factorial(n - 1) : 1;
}

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

void printInt(int i) {
    cout << i << endl;
}
```
A note on functions

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

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

- Remember constexpr

```cpp
// C++11 allows one return statements
cconstexpr 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;
}
```

- With constexpr we effectively have two versions:
  - a constexpr version
  - a non-constexpr-version

```cpp
// 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);
```
A note on functions

- Function calls have some costs in terms of performance
  - Safe register contents, put function arguments on stack, increment stackpointer, …, restore registers, perform jump back
    - But usually that’s not why your code is slow!

- If performance really matters compiler can inline small functions
  - A function call is replaced with copying the functions body to the call-site
  - Use keyword `inline` to give the compiler some hints
    ```cpp
    inline int add(int a, int b) { return a + b; }
    // such a call to add()
    int c = add(10, 20);
    // is replaced with
    int c = 10 + 20;
    ```

- Inlining 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 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;
    cout << i << endl;
    cout << j << endl;
    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 inside a function
- Example

```cpp
int i = 10;
double d = 1.234;
void printGobals() {
    cout << i << endl;
    cout << d << endl;
}

double addGlobals() {
    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 is harder to understand the code
  - It is hard to parallelize code that makes heavy use of globals
User-defined types/ non-build-in data types

- Learn about two very important user-defined types
  - `std::string`
  - `std::vector<typename T>`
- Both contained in the standard template library (STL)
- Vector is perhaps the most used non-build-in data type

- Define your own datatypes
  - Use `class` or `struct`
- Next lecture!
**std::string**

- **Why to use std::string in C++?**
- **C has no build-in string datatype**
  - In C a string is stored in an array of characters `char`
  - Example
    ```cpp
    char str[] = "Hello World";
    cout << str << '\n';
    int i = 0;
    while (str[i] != '\0') {
        cout << str[i] << '\n';
        i++;
    }
    char *ptr2str = "Hello World";
    ```
  - Such character arrays are terminated with `\0`
    - Which you can’t see

- **But remember build-in arrays are dangerous**
  - What if you lost the size of that array?
  - What if you lost `\0` or have multiple `\0` in your character array through incorrect string manipulation?
  - You risk reads and writes outside your array
    - Undefined behavior/ buffer overflows
    - Please watch this video
      - **Buffer overflow attack**

- **C++ has no build-in strings either**
- **But is offers a safe wrapper: std::string**
std::string

- Use `#include <string>` header file
- `std::string` is able to store a string
- `std::string` offers a lot of useful functionality as well
  - Defined 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 build-in arrays!
- No danger of buffer overflows!
- For a complete list of functionality 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 build-in variable
- Example

```cpp
// create a string from string literal
string str = "Hello World!";
// copy str into other
string other = str;
// get str's size
cout << str.size() << '\n';
// replace a single character
str[4] = 'O';
str += "some more characters";
// extracting a substring
string hello = str.substr(0,5);
string yetanother = "Hello";
// check for equality
cout << (hello == yetanother) << '\n';
```
std::vector<typename T>

- Again build-in arrays are dangerous for several reasons
- std::vector<typename T> is a safe wrapper for build-in arrays
- 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
  - Template metaprogramming: later in this course
- What are templates used for?
  - Allowing to write code that is independent of the type!
  - A vector can store any type!
    
    ```
    vector<int> ivec = {1, 2, 3};
    vector<double> dvec;
    vector<string> svec = { "Hello", "World", "!" };
    ```
std::vector<typename T>

- How to initialize or construct a vector?
- Example
  ```cpp
  vector<int> ivec; // call to default constructor
  vector<int> ivec(10); // call to constructor
  vector<int> ivec(10, 42); // another constructor
  vector<int> ivec{1, 2, 3, 4, 5}; // yet another constructor
  vector<int> ivec = {1, 2, 3, 4, 5}; // even more
  ```

- A vector can be constructed using a constructor
- All user-defined data types have constructors
  - A constructors job is to construct a variable / an object
    - Acquire resources and initialize it correctly
  - Constructors are special member functions (next lecture)
std::vector<

typename T>

- vector is designed in such a way that it can be used like a build-in type

- Example

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

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

- Introduce type aliases
  - with `typedef` or `using` keyword
  - Prefer `using`
  - as types get more complicated
  - May lead to a more flexible program
- `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
  vector<set<pair<string,int>>> v;
  // better use an alias for that
  using RD_t =
  vector<set<pair<string,int>>>;
  
  // now you can declare variables of that type
  RD_t inset; // looks nicer, easier to read and write
  ```
What are containers?

- `std::vector<typename T>` is a container
- A container in general can store a bunch of things
- Containers are generic
  - Using one or more template parameters
  - They can hold values of any type
- Use different containers for different purposes
- Choose the right container depending on your problem
- Remember you can nest containers!
  - `vector<vector<double>> matrix;`
STL containers?

- **Sequence containers**
  - array // fixed size array
  - vector // flexible size array
  - deque // double-ended queue
  - forward_list // single linked list
  - list // double 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 efficient
  - are available via same named header file
Containers in action

- Use STL vector to represent mathematical vectors $v \in \mathbb{R}^n$
- `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++
  - `vector<double> x{1, 2, 3};` // call the initializer_list constructor
  - `vector<double> y{4, 5, 6};` // call the initializer_list constructor
- We now have two vectors $x$ and $y$ filled with floating-point numbers
Containers in action

- \[ <a, b> = \sum_{i=0}^{n} a_i \cdot b_i \]

- A function calculating the scalar product

```cpp
double scalar_product(vector<double> x, vector<double> y) {
    double scal_prod = 0;  // create a variable holding the result
    if (x.size() != y.size()) { /* handle that error */ }  // check for right dimensions
    for (size_t i = 0; i < x.size(); ++i)  // iterate over vector entries
        scal_prod += x[i] * y[i];  // multiply the entries and add up to result
    return scal_prod;  // return the result
}
```
Containers in action

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

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

- A call to `scalar_product(x, y);`
  - Returns 32
More on types: Pointers, References & Values

- Take a deep breath!

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

- Three “kinds” of types exist in C++
  - “Normal/Value” integer type: `int i = 42;`
  - Pointer to an integer type: `int *j = &i;`
  - Reference to an integer type: `int &k = i;`
  - Makes C++ very powerful
  - Pointers and references are just addresses
    - But think of them as “pointers” (points-to graphs)

[Figure taken from https://www.victorybeachvacations.com/uploads/slides/kure%20beach%20nc%20vacation%20rentals.jpg]
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 = NULL; // f_ptr is initialized with a null-pointer: means 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), & at rhs is the get address of operator
int *k; // uninitialized pointer to an integer k
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 mailman can deliver letters and parcels
  - You can find a person using his address

```c
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
```
More on types: Pointers

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

```c
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
```

```
cout << &i << endl;
cout << i << endl;
cout << &j << endl;
cout << j << endl;
cout << &k << endl;
cout << k << endl;
```

<table>
<thead>
<tr>
<th>Variable's name</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0x7fffab7c4770</td>
<td>42</td>
</tr>
<tr>
<td>j</td>
<td>0x7fffab7c4778</td>
<td>0x7fffab7c4770</td>
</tr>
<tr>
<td>k</td>
<td>0x7fffab7c4774</td>
<td>42</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>
More on types: Pointers

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

- Important
  - A pointer might be 0
    - `int *i = nullptr;`
    - Means: 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 is null pointer\n"; }`
More on types: Pointers

- Things to remember about pointers
  - Declare a pointer type using \* (lhs)
  - Take an address of a variable with \&
  - Dereference a pointer with \*
  - A pointer might be a null pointer \$nullptr\$
More on types: References

- Example
  ```
  int i = 42;
  int &j = i;
  ```

- Declare a reference type by using `&` (lhs)
- You can use `j` as if it were `i`

- References behave much like pointers, **but**
  - Pointers can be re-assigned, references can’t
  - Pointers can be null and are allowed to dangle
    - References always refer to an object
  - Pointer’s address can be taken, can’t take references addresses
  - Pointers allow pointer arithmetic, references don’t (next lecture)
  - References are internally implemented as pointers
  - “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 function parameters
  - On the next slides

- Pointers
  - Use pointers to implement algorithms and data structures (e.g. linked lists)
  - Use pointers for dynamic memory allocation
  - Next lecture
Functions: parameter passing (and returning)

- How to pass and return huge data to and from a function?
- Consider a function that implements a matrix multiplication

```c
matrix matrixMult(matrix a, matrix b);
```

- Problem
  - If `matrixMult()` is called, actual parameters are copied!
  - Matrices can be huge, millions of elements
  - Processor is only copying data, rather than compute meaningful results
  - Can we avoid the process of copying large data?
  - Pass data by reference, rather than value!

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

- Matrices are not copied, we just pass a reference to a matrix (which is an address)
- Matrices can be used as usual in the function body
**Function: parameter passing (and returning)**

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

- **Problem**
  - Caution: If we modify `a` and `b` within the function we are changing the matrices.
  - How can we avoid 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 an error.

- **How to return huge results?**
  - Return by value or by reference
    ```cpp
    matrix matrixMult(const matrix& a, const matrix& b);
    ```
    ```cpp
    matrix& matrixMult(const matrix& a, const matrix& b);
    ```
  - Does not matter most of the times: modern compilers use return value optimization (RVO)!
Function: parameter passing (and returning)

- If your data is small (e.g. build-in types)
  - Pass and return by value (copy data)

- If you don’t know the size or deal with huge data
  - Pass by reference (data stays where it is, no unnecessary copying)
  - Use `const` if you don’t wish to modify the data
  - Return by value (since most compilers have RVO)
Recap

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

Questions?