C++14 Programming: Fundamentals and applications with the Qt class library

Peter Georg

October 1, 2019

Abstract

The main objective of this programming course is to teach modern C++14. It is split into three parts:

1. C++ basics, the standard library and objected-oriented programming
2. Present and work with the C++ Qt class library
3. Discuss selected advanced C++ and generic programming topics

The first part is derived from "C++ Primer (5th edition)" by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. However many details, potential pitfalls, and advanced topics have been omitted to fit in the timeframe of this course. For C++ beginners and for reference it is highly recommended to at least read parts of this book.

Disclaimer

The author provides no guarantee that information is up-to-date, accurate or complete and assumes no liability for the quality of this information. The author will in no way be liable for any direct or indirect damage resulting from use of or failure to use the information provided or from use of inaccurate or incomplete information.

Contents

1 Introduction

1.1 Literature and Reference

- C++ Primer (5th edition) by Stanley B. Lippman, Josée Lajoie, Barbara E. Moo
- The C++ Programming Language by Bjarne Stroustrup
- cppreference.com

1.2 Structure of a program

A C++ program consists of one or more source files. The file extension suffix for source files is not defined in the standard, most common naming conventions include `.cc`, `.cxx` and `.cpp`. Every C++ program must contain one function named `main` which will be called on execution. I.e. it is the starting point of the program. Here is the most minimalistic version of the `main` function that only returns a value to the operating system:

```
int main()
{
    return 0;
}
```

The value returned from `main` can be accessed by the operating system after execution.

1.3 Compiler

Having written our C++ program, we need to compile it. For this task we use a compiler to generate machine code from the C++ source code. There are several C++ compilers available, free and commercial. In this course we use the open source GCC (GNU Compiler Collection) as it is avaiable for various operating systems and is the de-facto standard C and C++ compiler for the GNU/Linux operating system. It is available for every Linux distribution and can be easily installed via the distribution specific package manager.

Actually it requires four steps to get from our source code to an executable: Preprocessing, Compilation, Assembly, and Linking. The GCC can be used for all four steps. Typically one simply does all steps at once, however one can also invoke GCC several times to do it step-by-step.

The GCC only provides command-line interfaces. The following commands have to be entered in a terminal.

To compile the source code in `source.cpp` to `app.out`:

```
g++ source.cpp -o app.out
```

To run the application:

```
./app.out
```

Print the return value of the last executed application in the current terminal:

```
echo $?
```

1.4 Preprocessor

Lines beginning with a # are directives read and interpreted by the preprocessor. The preprocessor is inherited from C, but not all of its features should be used in C++ . See e.g. the GCC documentation for more information about the C preprocessor.

1.4.1 include

The `include` directive is used to access code of a different file without having to copy anything. Indeed the preprocessor simply copies the content of included files.

Specifying the filename in angle brackets, the preprocessor searches for the file in system specified paths. Use this for system header files, e.g. headers of the standard library.

```
#include <iostream>
```

Filenames specified in double quotes are searched within the file's directory. A relative path can be used to access files in other directories. If the file can not be found, the preprocessor will search for the filename in the system specified paths as well. Use this for your own header files.

```
#include "mycode.h"
```

Note

`include` directives must appear outside any function.

Hint

Put all the `include` directives at the beginning of a file.

1.4.2 Macros

A macro is a kind of abbreviation which you can define once and use later. The `define` directive is used to define a macro. E.g.

```
#define SIMD_LENGTH_SP 16
```

defines a macro named `SIMD_LENGTH_SP` with value 16. Every occurance of `SIMD_LENGTH_SP` after the definition will be replaced by 16. Hence we can use this macro to define constants we set at compile time.

Hint

- Put all `define` directives at the beginning of a file.
- Use ALL CAPS names for macros to easily distinguish from variables.
- In C++ we prefer constant variables to macros.

1.4.3 Conditionals

Conditionals can be used to test macros or alter code depending on macros at compile time. We can define our own conditionals or test predefined conditionals.

Enable/Disable time consuming code paths, i.e. for debugging:

```
#define DEBUG

#ifdef DEBUG
    //Some code to help debugging.
#endif //End of DEBUG
```

Enable/Disable code depending on operating system or architecture it is compiled for:

```
#ifdef __linux__
    //Linux specific code
#elif _WIN32
    //Windows specific code
#elif _APPLE_
    //Mac OS X specific code
#else
    //Other systems
#endif // OS
```

Hint

- Put all `define` directives at the beginning of a file.
- Use ALL CAPS names for conditionals to easily distinguish from variables.

1.5 Comments

Comments help the human reader to understand a program. These typically summarize an algorithm, describe a function or identify the purpose of a variable. Especially for otherwise obscure segments of code it is crucial to add comments. This not only helps other people to understand your code, but also yourself as it might be necessary to re-read and understand your own code after few months/years. A comment is marked by starting a line with a double slash (//) and are ignored by the compiler and preprocessor.

For code that is intended to be used by other developers it is highly recommend to not only add comments, but also documentation. To ease this process tools like Doxygen can be used to generate documentation from comments. This requires Doxygen styled comments, hence decide wether to use it or not before you write any code.

Hint

- Any code that migt not be totally clear requires comments!
- There is no rule of thumb, e.g. 50% have to be comments.
- Write code and comments at the same time. Do not add comments later!

Warning

Whenever you change any code, be sure to update the comments!

1.6 Coding conventions

Recommended literature: “Clean Code” by Robert C. Martin

Why adhere to coding conventions?

- Improve readability
- Ease of maintenance

What is covered by coding conventions?

- Comment conventions
- Indent style conventions
- Line length conventions
- Naming conventions
- Programming practices
- Programming principles
- Programming rules of thumb
- Programming style conventions

Note

- There is no single correct coding convention.
- Especially for projects with many developers coding conventions are crucial.

Warning

Once you have set your coding conventions, use it consistently.

Hint

Adhere to coding conventions!

1.7 Basic Standard Input and Output

The C++ language itself does not define any statements for IO. The C++ standard library provides this functionality. For IO, only the `iostream` library is required. This library defines two types named `istream` and `ostream`, which represent input and output streams, repsectively. A stream is a sequence of characters read from or written to an IO device. In addition the library defines four IO objects for input and output. For standard input we use the object `cin` of type `istream`. For standard output we use the object `cout` of type `ostream`. The other two IO objects are of type `ostream` and are for standard error (`cerr`) and logging (`clog`).

```
#include <iostream>
int main()
{
    std::cout << "Please enter two numbers:" << std::endl;
    int a = 0, b = 0;
    std::cin >> a >> b;
    std::cout << "The sum of " << a << " and " << b << " is " << a + b << std::endl;

    return 0;
}
```

The output (`<<`) operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes the given value on the given ostream. The result of the ouput operator is its left-hand operand.

`std::endl` is a special value called a manipulator. It ends the current line and flushes the buffer. Flushing the buffer ensures that all ouput of the program is actually written to the output stream.

The input (`>>`) operator behaves analogously to the output operator. It reads data from the given `istream`, stores it in the given object and returns its left-hand operand as its result.

1.8 Using Names from the Standard Library - The Scope Operator

All objects, types, and functions provided by the Standard Library are defined inside the namespace `std`. Namespaces are used to avoid inadvertent collisions between the names we define and uses of those same names inside a library. Thus whenever we want to access a name defined within a namespace, we must explicitely specify the namespace as well. We use the scope operator (`::`) for this purpose.

Part I

The Basics

2 Variables and Basic Types

The type of a data defines its meaning and possible operations. C++ supports several primitive types, but also lets us define our own data type. The standard library extensively uses this possibility to define complicated data types.

2.1 Primitive Built-in Types

2.1.1 Arithmetic Types

Type	Meaning	Minimum Size	
bool	boolean	NA	
char	character	8 bit	
wchar_t	wide character	16 bit	
char16_t	Unicode character	16 bit	
char32_t	Unicode character	32 bit	integral types
short	short integer	16 bit	
int	integer	16 bit	
long	long integer	32 bit	
long long	long integer	64 bit	
float	single-precision floating-point	6 significant digits	
double	double-precision floating-point	10 significant digits	floating-point types
long double	extended-precision floating-point	10 significant digits	

Signed and Unsigned Types

All integral types, except bool and extended characters, can be signed or unsigned. A signed type represents zero, negative or positive numbers. A unsined type represents only values greater than or equal to zero. All types representing integers are signed by default. We obtain the corresponding unsigned type by adding `unsigned` to the type, e.g. `unsigned int`.

For `char` there exist three different types: `char`, `signed char`, `unsigned char`. In particular `char` is not the same as `signed char`. However there are only two representations for these three character types. The plain `char` uses one of the other two, but it is not defined which.

Which data type to use?

- Use `unsigned` only when you are sure the values cannot be negative.
- Do not mix `signed` and `unsigned` values.
- Use `int` for integer arithmetic. If `int` is to small to hold your values, then use `long long`.
- Use `bool` and plain `char` only to hold truth values and characters. If you really need a tiny integer use `signed char` or `unsigned char` explicitly.
- Use `double` for floating-point computations. Only use `float` if you are sure single-precision is sufficient and it is beneficial for run-time and memory requirements. The added precision of `long double` is typically not necessary and does not justify the added run-time cost.

2.1.2 void

The void data type has no associated values and can only be used in a few circumstances. Typically it used as return type for functions that do not return a value.

2.1.3 Type Conversions

Type conversions happen automatically when we use an object of one type where an object of another type is expected. The behavior depends on the involved types and might lead to loss in precision:

- Assigning a nonbool type to a bool object, it is `false` if the value is 0 and `true` otherwise.
- Assigning a bool to an arithmetic type, it is 1 if the bool is `true` and 0 otherwise.
- Assigning a floating-point value to an integer, the value is truncated.
- Assigning an integer to a floating-point object, the fracitional part is zero.
- Assigning an out-of range value to an object of unsinged type, the result is the remainder of the value modulo the number of values the target type can hold.
- Assigning an out-of-range value to an object of singed type, the resul is undefined.

Avoid undefined and implementation-defined behavior

Undefined behavior is not required to be detected by the compiler, it depends on the used compiler and its settings. Thus a program with undefined behavior might compile and even run in some circumstances, but is not guaranted to run or produce correct result in other circumstances. These problems are hard to track and hence should be avoided from the beginning.
A typical error for implementation-defined behavior is assuming that the size of `int` is a fixed and known value.

2.1.4 Literals

Every literal, e.g. a value, has a type that is determined by its form and value.

Integer literals

Integer literals can be in decimal, octal or hexadecimal notation. The notation is determined by the beginning of the literal. Integer literals that begin with 0 (zero) are octal, those with 0x, or 0X, are hexadecimal. The type of an integer literal is determined by the smallest type in which its value fits. A decimal literal has the smallest of `int`, `long`, or `long long`. For octal and hexadecimal it's the smallest of `int`, `unsigned int`, `long`, `unsinged long`, `long long`, or `unsigned long long`. There is no literal corresponding to `short`. The smallest possible type can be overwritten by adding suffixes to the literal. `l` or `L` for `long` and `ll` or `LL` for `long long`. These can be combined with `u` or `U` for `unsigned`.

Hint

Avoid `l` as it is easily mistaken for the digit 1.

Floating-point literals

Floating-point literals include either a decimal or an exponent specified using scientific notation. By default all floating-point literals are double-precision. To overwrite the default add the suffix `f` or `F` for single-precision, `l` or `L` for extended-precision.

Character and character string literals

A character literal is enclosed in single quotes. A string literal is enclosed in double quotation marks. The type can be set by adding one of the following prefixes: `u` (Unicode 16), `U` (Unicode 32) or `L` (wide) or `u8` (utf-8, string only).

Boolean and pointer literals

The literals for type bool are the words `true` and `false`. The word `nullptr` is a special pointer literal.

2.2 Variables/Objects

A variable/object is a named storage of a specified type that can be manipulated. The type determines the size and layout of the object's memory and the set of operations that can be applied to the variable.

2.2.1 Definition

A simple variable definition consists of a type specifier, followed by a list of one or more variable names separated by commas, and ends with a semicolon. A variable might (optionally) be set to an initial value.

```
int a, b, c, d, e, f;
```

2.2.2 Initialization

A value that is initialized gets the specified value at the moment it is created. The value for initialization can be an arbitrarily complicated expression. For initialization the = symbol or list initilization is used.

```
int a = 5, b = 2, c, d = 0x0, e(b), f{b};
```

Warning

Initialization is not assignment.

Default initialization

Variables without an initializer are default initialized, i.e., the variable is set to a default value. The default value depends on the type of the variable and may also depend on where the variable is defined. E.g. for built-in types the default value for variables outside any function body are initialized to zero. Most variables of built-in type defined inside a function are uninitialized, i.e., their value is undefined.

Warning

It is not allowed to access a variable with undefined value.

Hint

Initialize all variables of built-in type.

2.2.3 Declaration

C++ supports splitting programs in logical parts (multiple files) and compiling each part independently. However we might need to access code written in a different file, e.g. a variable. Hence C++ distiguishes between declaration and definition. A declaration makes a name known to the compiler, a definition in addition creates (and initializes) the associated entity. To obtain a declaration that is not also a definition, we use the `extern` keyword.

```
extern int a;       //Declaration
int b = 2;          //Declaration and Definition
extern int c = 2;   //Declaration and Definition
```

Note

A variable must be defined exactly once, but can be declared many times.

Information

Having to declare variables defined in other files is a consequence of statically typed languages. This means types are checked at compile time. This is useful to avoid runtime bugs in complicated programs, but requires that the type of every entity we use is known to the compiler.

2.2.4 Identifiers

The variable identifier is its name. An identifier can be composed of letter, digits, and the undescore character. Identifiers are case sensitive and must begin with either a letter or an underscore. The name length is not limited. The C++ language reserves a set of names for its own use which might not be used as identifiers. In addition the standard reserves a set of names for use in the standard library. Thus identifiers we define may not contain two consecutive underscores, nor begin with an underscore followed immediately by an uppercase letter. Identifiers defined outside a function may not begin with an underscore.

Hint

- An identifier should self-explain its meaning.
- Use one naming convention consistently.

C++ Keywords

alignas	constexpr	extern	noexcept	static_assert	union
alignof	const_cast	false	nullptr	static_cast	unsigned
asm	continue	float	operator	struct	using
auto	decltype	for	private	switch	virtual
bool	default	friend	protected	template	void
break	delete	goto	public	this	volatile
case	do	if	register	thread_local	wchar_t
catch	double	inline	reinterpret_cast	throw	while
char	dynamic_cast	int	return	true	
char16_t	else	long	short	try	
char32_t	enum	mutable	signed	typedef	
class	explicit	namespace	sizeof	typeid	
const	export	new	static	typename	

Alternative operator names

and	bitand	compl	not_eq	or_eq	xor_eq	and_eq	bitor	not	or	xor

2.2.5 Scope of a name

At any particular point in a program, each name that is in use refers to a specific entity. A name can be reused to refer to other entities at different points in the program. A scope is a part of the program in which a name has a particular meaning. Most scopes in C++ are delimited by curly braces. Names are visible from the point where they are declared until the end of the scope.

```
#include <iostream>
int sum = 0;
int main()
{
    sum += 1;
    int sum = 0;
    sum += 2;
    std::cout << "Sum is " << sum << " or " << ::sum << std::endl;
    {
        int sum = 0;
        sum += 3;
        std::cout << "Sum is " << sum << " or " << ::sum << std::endl;
    }
    sum += 4;
    std::cout << "Sum is " << sum << " or " << ::sum << std::endl;
    return 0;
}
```

Global scope: Names defined outside any curly braces, e.g. the `main` function.

Block scope: Names defined within curly braces, e.g. both `sum` variables.

Nested scopes: Scopes can contain other scopes. The nested scope is referred to as an inner scope, the containing scope is the outer scope. Inner scopes can access all names declared in outer scopes. Names declared in outer scopes can be redefined in an inner scope.

Warning

It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

Note

Use the `::` operator to access global variables with te same name as local variables.

Variable definition

Define objects near the point at which it is first used. This improves readability and it is often easier to give the variable a useful initial value.

Global variables

Avoid using global variables.

2.3 Compound Types

A compound type is defined in terms of another type. There exist several compound tyes in C++ , but only two — references and pointers — will be covered here.

Defining variables of compound type is more complicated that what we have seen so far for primitive types. A declaration consists of a base type followed by a list of declarators. Each declarator names a variable and gives the variable a type related to the base type.

So far declarators have only been variable names and the type of the variable is set by the base type. More complex declarators specify variables with compound types based on the base type of the declaration.

2.3.1 (Lvalue) References

A reference defines an alternative name for an object. A reference is defined by writing a declarator of the form `&name`.

When we define a reference, instead of copying the initializer's value, we bind the reference to its initializer. A reference can not be rebound to a different object, hence references must be initialized.

```
int iVal = 1;
int &refVal = iVal;      // refVal refers to iVal
refVal = 2;              // same as iVal = 2;
int &refVal2 = refVal    // refVal2 refers to iVal
int &refVal3;            // not allowed
```

Note

- A refence is not an object, it is just another name for an already existing object.
- With two exception, the type of a reference and the object to bind to must match exactly.

Warning

- A reference must be initialized.
- A reference to a literal is not allowed.

Information

- There is no reference to a reference.
- Multiple reference definitions in a single definition are allowed. Each identifier that is a reference must be preceded by the `&` symbol.

2.3.2 Pointers

A pointer is an object that "points to" another type by holding its address. A pointer can be assigned and copied, hence it can point to different objects over its lifetime and does not need to be initialized. A pointer is defined by writing a declarator of the form `*name`.

To get the address of an object we use the address-of (`&`) operator.
To access the object a pointer points to we use the dereference (`*`) operator.

```
int iVal = 1;
int *pi = &iVal;      // pi points to iVal
*pi = 2;              // same as iVal = 2
int *pi2 = nullptr;   // pi2 points to null
pi2 = pi;             // pi2 points to iVal
int **ppi = &pi;      // ppi points to pi
*ppi = pi2;           // same as pi = pi2
**ppi = 3;            // same as iVal = 3
pi = nullptr;
*pi = 4;              // undefined behavior
ppi = pi;             // error: types differ
double dVal = 2.0;
pi = &dVal;           // error: types differ
double *pd = &iVal;   // error: types differ
```

Pointer Value

The value of a pointer can be in one of four possible states:

- Point to an object.
- Point to the location just immediately past the end of an object.
- Point to null, i.e., not pointing to any object.
- All other pointers are invalid.

Null pointers

A null pointer does not point to any object. You can check for null before attempting to use a pointer. Any nonzero (i.e. not pointing to null) pointer evaluates as `true`. Use the literal `nullptr` to set a pointer to null.

void* pointer

The `void*` is a special pointer type that can point to any object regardless of the type. As the type of the object it points to is unknown we can not operate on the object. `void*` is used to work with memory as memory, rather than using the pointer to access the object stored in that memory.

Warning

- A non initialized pointer in block scope has undefined behavior.
- The result of accessing the value of an invalid pointer is undefined.

Note

- With two exception, the type of a pointer and the object to point to must match exactly.

Information

- There is no pointer to a reference.
- Multiple pointer definitions in a single definition are allowed. Each identifier that is a reference must be preceded by the `*` symbol.

Hint

- Initialize all pointers!
- Only dereference a valid pointer that points to an object.
- Avoid `void*` pointers.
- Prefer references to pointers.
- Read pointer or reference declarations from right to left.
- Use smart pointers.

2.4 const Qualifier

2.4.1 const variables

Sometimes we want to have a variable with a value that cannot be changed. We can make a variable unchangeable by definind its type as `const`:

```
const int vectorLength = 512;
int const vectorLength = 512;
```

Note

- A const object cannot be changed, hence it must be initialized.
- An object of const type can be used in any expression as its nonconst version as long as only operations are used that cannot change the object.

By default a const variable is local to a file. Use the keyword `extern` on both definition and declaration(s) to define a single instance of a const variable.

```
// file.cc
extern const int vectorLength = 512;     // definition and initialization
// file.h
extern const int vectorLength;           // declaration
```

2.4.2 References to const

We can bind a reference to an object of a const type. To do so we use a reference to const, which is a reference that refers to a const type. A reference to const cannot be used to change the object it refers to.

```
const int constInt = 512;
const int &refIntConst = constInt;    // Reference to const to const object
int const &refConstInt = constInt;
refConstInt = 256;                    // error: Reference to const
int &refInt = constInt;               // error: Reference to const object
```

A reference to const may refer to an object that is not const, but apart from that has a matching type. A reference to const only restricts what we can do through that reference. If the underlying object is nonconst it might be changed by other means:

```
double dVal = 3.14;
double const &refConstDouble = dVal; // Reference to const to nonconst object
refConstDouble = 3.142;              // error: Reference to const
dVal = 3.141;                        // refConstDouble is now equal to 3.141
```

A reference to const can refer to temporary objects, e.g., literals. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression.

```
float const &refConstFloat = 3.142;  // Reference to const to temporary object
int const &refConstInt = dVal;       // Reference to const to temporary object
```

Note

A reference to const can be initialized from any expression that can be converted to the type of the reference. However the reference to const then refers to a temporary object.

2.4.3 Pointers to const

Similar to reference to const we can define a pointer to const. Again the pointer cannot be used to change the object it points to. We may store the address of a const object only in a pointer to const.

```
const int constInt = 512;
const int *ptrIntConst = &constInt;  // Pointer to const to const object
int const *ptrConstInt = &constInt;
ptrIntConst = ptrConstInt;
*ptrConstInt = 256;                  // error: Pointer to const
int *ptrInt = &constInt;             // error: Pointer to const object
double dVal = 3.14;
double const *ptrConstDouble = &dVal;// Pointer to const to nonconst object
dVal = 3.141;
*ptrConstDouble = 3.142;             // error: Pointer to const to nonconst object
float const *ptrConstFloat = &3.142; // error: A literal has no address
```

Note

A pointer to const can point to a nonconst object.

Hint

Pointers and references to const think they point or refer to a const object.

2.4.4 const Pointers

Pointers are objects, hence pointers can be defined to be const just like any other object. We indicate that the pointer is const by putting the `const` after the `*`. This placement indicates that it is the pointer, not the pointed-to type, that is const:

```
int val = 512;
int const *ptrConstInt = &val;                  // Pointer to const to int
int *const constPrtInt = &val;                  // Const pointer to int
int const *const constPtrConstInt = &val;       // Const pointer to const to int
ptrConstInt = constPtrConstInt;
constPtrConstInt = ptrConstInt;                 // error: Const object can not be changed
```

Note

A `const` pointer must be initialized as its value, the address it holds, cannot be changed.

2.4.5 Top- and Low-level const

To better distinguish to what the `const` qualifier applies we use the terms top- and low-level const. Top-level const indicates that the object itself is const. Low-level const indicates that the base type of a compound type is const.

Hint

Use `const` for everything you do not modify!

2.4.6 Constant expressions

A constant expression is an expression whose value cannot be changed and can be evaluated at compile time. A literal is a constant expression. A const object that is initialized from a constant expression is also a constant expression.

```
const int constInt = 512;                // Is a constant expression
const long constLong = constInt + 1;     // Is a constant expression
int iVal = 512;                          // Is not a constant expression
const short constShort = getLength();    // Is not a constant expression
```

We can ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Objects declared as `constexpr` are imlicitly `const` and must be initialized by constant expressions:

```
constexpr int constInt = 512;                    // Is a constant expression
constexpr long constLong = constInt + 1;         // Is a constant expression
constexpr short constShort = vectorLength();     // Depends on vectorLength()
```

Note

Only `constexpr` functions can be used as an initializer for a `constexpr` function.

Hint

Use `constexpr` for variables you intend to use as a constant expression.

2.5 Dealing with types

2.5.1 Type aliases

A type alias is a name that is a synonym for another type. We can use type aliases to simplify complicated type definitions or emphasize the purpose for which a type is used. There are two ways to define a type alias:

The keyword `typedef` can be used to define an alias:

```
typedef float single;           // single is a synonym for float
typedef single price, *pprice;  // price is a synonym for float, p for float*
price apple;
const pprice cucumber;          // error: Uninitialized const pointer
float const* cucumber;
```

An alias declaration can also be used to define a type alias:

```
using single = float;
const single pi = 3.14;
```

Warning

Declarations that use type aliases for compound types and `const` can yield surprising results.

Hint

Prefer `using` to `typedef`.

2.5.2 The auto type specifier

Often we want to store the value of an expression in a variable. To declare the variable we must know the type of the expression. This might be quite difficult or even impossible in some cases. For that reason we can simply use the `auto` type specifier and let the compiler figure out the type for us. The compiler will deduce the type from the initializer, hence a variable using the `auto` type specifier must be initialized.

```
auto size = 12, *pSize = &size, &rSize = size;
auto length = 16, pi = 3.14;    // error: Different types for length and pi
auto height = getHeight();
auto iVal = rSize;
```

Note

- A variable with `auto` as type specifier must be initialized.
- Multiple variables can be defined using `auto`, but all have to be of the same type.
- For references the compiler uses the object's type for `auto`'s type deduction.
- `auto` ignores top-level `const`s.
- `auto` keeps low-level `const`s.

Hint

- Be lazy, use the `auto` type specifier!
- Explicitely define the deduced type to be top-level `const`.
- Prefer `auto const` to `const auto`
- Use references to `auto` to avoid unnecessary copies.

2.5.3 The decltype type specifier

Sometimes we might want to deduce the type from an expression, but not use the expression for initialization. For such cases, we can use the `decltype` type specifier, which returns the type of its operand. The expression is analyzed by the compiler, but not evaluated.

```
decltype(getLength()) length = 0;
```

Note

- For variables `decltype` returns the type, including top-level `const` and references.
- For expressions that yield objects that can stand on the left-hand side of an assignment, `decltype` returns a reference type.
- `decltype((variable))` always returns a reference type.

2.6 Defining our own data structures

A data structure is used to group together related data elements and operations to work with the data. In C++ we define our own data types by defining a class. For now we will limit ourselves to a data structure used to group together related data elements without any operations.

```
struct Coordinates
{
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
};

Coordinates mensa, cipPool;
mensa.x = 1.0;
mensa.y = 2.0;
cipPool.x = -5.0;
```

We defined a new class using the `struct` keyword with three data members defined in the class body. For each data member a in-class initializer has been supplied. We use the dot operator `.` to access data members of an object.

Note

- The class definition ends with a semicolon.
- For in-class initializers we can not use parantheses.

2.7 Writing our own header files

Typically classes are defined outside any function, hence there may be only one definition of a class in any source file. In addition, if we use a class in several different files, the class' definition must be the same in each. Thus we usually define classes in header files once and include the header in all files it is required. The header's file name is typically derived from the class' name.

We also use headers to divide our program code to many source files. To, e.g., use a function defined in one source file, it is enough to include a declaration of the function. Typically the header containing the function declaration has the same name as the source file containing the definition.

Whenever a header is updated, all source files that use that header must be recompiled.

Headers contain entities (such as class definitions and `constexpr` variables) that can be defined only once in any given file. Often headers rely on facilities from other headers, hence it can often happen that a header is included more than once. Thus we need to write our header in a way that is safe even if it is included multiple times. This is achieved using the preprocessor to define header guards.

```
#ifndef COORDINATES_H
#define COORDINATES_H
//include required headers

struct Coordinates
{
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
};
#endif // COORDINATES_H
```

3 Strings, vectors, and arrays

3.1 String

A string is a variable-length sequence of character. The string type is part of the standard library, hence we have to include its header, `string`, before we can use it.

3.1.1 Definition and Initialization

How to initialize objects of a class' type is defined by the class. It may define many different ways to initialize objects of its type. Each of these must be distinguished from the others by the number of initializers, or by the types of those initializers.

```
std::string s1;              //Default initialization; s1 is an empty string
std::string s2(s1);          //s2 is a copy of s1
std::string s2 = s1;
std::string s3("value");     //s3 is a copy of the string literal "value"
std::string s3 = "value";
std::string s4(n,'c');       //Initialize s4 with n copies of the character 'c'
```

As already seen before there are different forms of initialization. Using the = operator we copy initialize the object by copying the initializer on the right-hand side into the object being created. Otherwise we use direct initialization. When we have a single initializer, we can use either of the two initialization forms.Having more than one value we want to initialize from, we have to use direct initialization.

3.1.2 Operations

A class also defines operations that objects of the class type can perform. These operations are either used by defining what operator symbols mean when applied to objects or functions that are called by name (using the `.` operator). The string class lets us convert both character literals and character string literals to string. Thus we can use these literals where a string is expected. For each operator, such as `+`, at least one operand must be of string type.

Selected operations:

`os << s`	Write `s` onto the output stream `os`.
`is >> s`	Reads string from input stream `is` into `s` up to first whitespace.
`std::getline(is,s)`	Reads a line of input from `is` into `s`.
`s.empty()`	Returns `true` if `s` is empty.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `n+1`th char in `s`.
`s1 + s2`	Returns concatenation of `s1` and `s2`.
`s1 = s2`	Replaces `s1` by a copy of `s2`.
`s1 += s2`	Appends `s2` to `s1`.
`s1 == s2`	Returns true if both strings are equal (case-sensitive).
`s1 != s2`	Returns false if both strings are equal (case-sensitive).
`<,<=,>,>=`	Comparisions (case-sensitive).

3.1.3 Dealing with characters in a string

To deal with a character in a string we need a way to access it. We can access characters using a range-based for loop, iterators or subscript. The C library `ctype.h` includes many functions that can help you to deal with characters. Instead of including the C library we use the C++ version named `cctype`. This is more convenient as names from the standard library are cosistently found in the std namespace. Using the C library you have to remember which names are inherited from C and which are unique to C++ .

Note

The C++ library incorporates the C library. Always use the C++ version adding the prefix `c`.

Warning

The value of a subscript must be >= 0 and < the `size()` of the string.

Hint

Use a variable of type `std::string::size_type` as the subscript. Use `decltype(s.size())`.

3.2 Vector

A vector is a container of objects that are all of the same type. For access each object has an associated index. There is no limitation concerning the type of contained objects. Obviously there is not an implementation of the vector class for every object — vector is a class template. C++ support class and function templates, we will deal with both later. Templates are not functions or classes. Templates are instructions for the compiler how to generate classes of functions. The process to create a function or class from a template is called instantiation. For now it is enough to know that we have to specify which class to instantiate by specifiying additional information. This information is supplied inside a pair of angle bracket following the template's name.

```
#include <vector>
std::vector<int> iVec;
std::vector<Coordinates> coordsVec;
std::vector<std::vector<float>> matA;
```

Note

A vector is a template, not a type.

Warning

In pre C++11 standard the syntax to define a vector of a template was slightly different:

```
std::vector<std::vector<float> > matA;
```

Some compilers may still require the space between closing brackets.

3.2.1 Definition and Initialization

```
std::vector<T> v1;                  // empty vector containing objects of type T
std::vector<T> v2(v1);              // v2 is a copy of v1
std::vector<T> v2 = v1;
std::vector<T> v3(n, val);          // vector containing n elements with value val
std::vector<T> v4(n);               // vector contains n value-initialized objects
std::vector<T> v5{a, b, c...};      // vector with elements a, b, c...
std::vector<T> v5 = {a, b, c...};
```

3.2.2 Adding, removing and accessing elements

The number of elements in a vector is dynamic. We can add an element at the end of a vector using the `push_back()` member function. But we can also remove elements at any time, e.g. using the `pop_back()` member function. This possibility imposes some new obligation on our program. You can access elements in a vector using range for, subscripting or iterators.

Warning

- Loops need to be correct even if the loop changes the size of the vector.
- A range for must not change the size of the sequence over which it is iterating.
- The subscript operator does not add elements.
- Subscripting an element that does not exist is a buffer overflow error. These errors are extremely common and malicious errors. Indeed these bugs are the most common cause of security issues. In addition these bugs are very hard to detect and locate.

Note

The type of a subscript is the corresponding `std::vector<T>::size_type`.

Hint

Avoid using subscripting to ensure that subscripts are in range.

3.3 Iterators

We use iterators to access elements of a container or characters of a string. For this iterators, like pointers, give indirect access to an object. In addition iterators have operations to move from one element to another. Every container defined in the standard library supports iterators.

3.3.1 Using iterators

Every type that supports iterators has a `begin()` and `end()` member functions. The `begin()` function returns an iterator that denotes the first element, `end` returns an iterator positioned one element past the last element. As with pointers an iterator might be valid or invalid. A valid iterator either denotes an element or a position one past the element in a container.

```
auto b = iVec.begin(), e = iVec.end();
auto b = std::begin(iVec), e = std::end(iVec);
```

Note

- For empty containers the iterators returned by `begin()` and `end()` are equal.
- Instead of using the member functions, you can also use the corresponding functions in `std`.
- For a `const` container these functions return an iterator to `const`.
- To get an iterator to `const` elements of a non `const` container use the corresponding functions. These function names' start with the prefix `c`.
- There also exist functions to get iterators for reverse iteration.

3.3.2 Iterator operations

`*iter`	Returns a reference to the element denoted by `iter`.
`iter->mem`	Same as `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element.
`--iter`	Decrements `iter` to refer to the previous element.
`iter1 == iter2`	Iterators are equal if both itertors denote the same element or
`iter1 != iter2`	a position one past the element of the same container.

Note

- The iterator returned by `end()` may not be incremented or dereferenced.
- The arrow `->` operator combines dereference and member access in a single operation.

```
for(auto it = iVec.cbegin(); it != iVec.cend(); ++it)
{
    std::cout << *it << std::endl;
}
```

Warning

Changing the size of a container potentially invalidates all iterators.

Hint

Loops that use iterators should not change the size of the container to which the iterators refer.

3.3.3 Iterator Arithmetics

In addition to the just presented operations that all iterators support, iterators for string and vector support additional operations. These operations include arithmetic and relational operators. For all these operators all iterators used as operands or returned as a result must denote elements in, or one past the end of, the same container.

3.4 Arrays

An array is a data structure similar to the vector type with a different trade-off between performance and flexibility. Both are a container of unnamed objects of a single type that can be accessed by position. In contrast to vector, arrays have fixed size, which might offer better run-time performance at the cost of lost flexibility.

Hint

If in doubt how many elements you need, use a vector.

3.4.1 Definition and initialization of built-in arrays

Array is a compound type with the declarator of the form `name[size]`. The size of an array is part of the array's type, hence must be known at compile time.

```
float fArr[10];
int iArr[10] = {};
float vector[vectorLength()];
```

Note

- The type of `size` must be a constant expression.
- An array cannot be initialized as a copy of another array.
- We cannot assign one array to another array.

Warning

By default elements of an array are default initialized, thus might have undefined values.

3.4.2 Accessing elements

We can access elements using the range for or the subscript operator. The subscript type is `size_t`, a machine-specific unsigned type defined in `cstddef`.

Hint

Check subscript values!

3.4.3 Pointers and arrays

```
int iArr[10] = {};
int *p = &iArr[0];
int *p = iArr;
```

Note

In most expressions, when we use an object of array type, we are really using a pointer to the first elements;

Pointers are iterators

Pointers that address elements in an array can be used just like iterators. We can reuse the `begin` and `end` functions in `std` to get a pointer to the first element or one past the last element of an array.

3.4.4 Multidimensional arrays

To get multidemensional arrays we can simply define arrays of arrays.

```
float matA[10][10];
matA[0][0] = 1.0;
```

Hint

Prefer `std::vector` to `std::array` to built-in array.

4 Expressions

4.1 Fundamentals

4.1.1 Basic Concepts

There exist four different types of operators:

- Unary operators — act on one operand
- Binary operators — act on two operands
- Ternary operator — acts on three operands
- Function call — takes an unlimited number of operands

We can group several operators and operands in one expression. Understanding such expressions requires understanding precedence and associativity of the operators and may depend on the order of evaluation of the operands.

To be able to evaluate an expression, operands are often implicitly converted from one type to another. For example this is necessary for most binary operators as these expect both operands to be of the same type. In most cases the conversion is obvious, but in some cases the result is quite surprising.

Overloaded operators

The languange defines what the operators mean to built-in and compound types. We can define what most operators mean when to applied to class types. We refer to such a defined operator as overloaded

operator. The meaning of overloaded operators and the type of its operand(s) depend on the definition. However the number of operands and the precedence and associativity of the operator cannot be changed.

4.1.2 Precedence and Associativity

Compound expressions are evaluated by grouping the operands to the operators. This is determined by presendence and associativity. Parantheses can be used to override precedence ans associativity by manually grouping operands to operators.

4.1.3 Order of evaluation

Precedence specifies how the operands are grouped, but says nothing about the order in which the operands are evaluated. In most cases the order is unspecified.

```
auto iVal = vectorLength() * elementSize();
```

We know that both functions need to be evaluated before the multiplication can be done, but there is no way to tell the order the functions are evaluated.

Warning

For operators that do not specify evaluation order, it is an error to refer to and change the same object.

```
std::cout << iVal << " " << ++i << std::endl;
```

Hint

- When in doubt paranthesize compound expressions.
- If you change the value of an operand, don't use that operand elsewhere in the same expression.

4.2 Arithmetic operators

4.3 Logical and relational operators

4.4 Assignment operator

4.5 Increment and decrement operators

Hint

Use postfix operators only when necessary.

4.6 Member access operators

4.7 Conditional operator

The conditional `?:` operator is used to embed simple if-else logic inside expressions of the form:

```
cond ? expr1 : expr2;
```

where `cond` is an expression that is used as a condition and both `expr` are expressions of the same type. The operator executes by evaluating `cond`. If the condition is true, then `expr1` is evaluated, otherwise `expr2`.

```
std::string sign = (iVal < 0) ? "negative" : "positive";
```

Hint

- Use conditional operations only for simple if-else logic.
- Do not nest conditional operations.

4.8 Bitwise operator

4.9 sizeof operator

4.10 Comma operator

4.11 Type conversions

4.11.1 Implicit

Note

In most cases works as expected.

4.11.2 Explicit

We can use a cast to request an explicit conversion.

Warning

Although necessary at times, casts are inherently dangerous constructs.

```
cast-name<type>(expression);
```

where `type` is the target type of the conversion of `expression`. The `cast-name` may be one of `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. Here we only explain `static_cast`. The `dynamic_cast` supports run-time type identification. The other two are highly dangerous and thus should not be used anyway.

static cast

Any well-defined type conversion, other than whise involving low-level `const`„ can be requested using `static_cast`.

```
auto division = static_cast<double>(iVal)/iVal2;
```

Hint

- Use `static_cast` when a larger arithmetic type is assigned to a smaller type. The cast informs both the reader and the compiler that we are aware of and are not concerned about the potential loss of precision.
- Use `static_cast` to convert a `void*` to any other pointer.

4.12 Available operators and their precendence and associativity

See http://en.cppreference.com/w/cpp/language/expressions

5 Statements

5.1 Simple Statements

Expression statement

```
iVal + 5;            // rather useless
std::cout << iVal;
```

Null statement

```
while (!signal())
    ;
```

Hint

Comment null statements.

Warning

Extraneous null statements are not always harmless.

Compound statement

A compound statement, usually refered to as a block, is a (possible empty) sequence of statements and declarations surrounded by a pair of curly braces. Compound statements are used to group several statements to a single statement.

Note

A block is **not** terminated by a semicolon.

5.2 Statement Scope

Variables defined inside the control structure of the `if`, `switch`, `while` and `for` statements are only visible within that statement and are out of scope after the statement ends.

5.3 Conditional statements

5.3.1 The if statement

```
if (condition)
    statement

if(condition)
    statement1
else
    statement2
```

Hint

Use correct indentation to indicate the correct depth for nested `if` statements.

Note

In nested `if` statements each `else` is matched with the closest preceding unmatched if.

Hint

Use braces to control the execution path.

5.3.2 The switch statement

```
switch(command)
{
    case 'c':
        copy();
        break;
    case 'd':
        delete();
        break;
    case 'a':
        append();
        break;
    default:
        help();
        break;
}
```

Hint

Always add a default label even it is empty.

5.4 Iterative Statements

5.4.1 The while statement

```
while(condition)
    statement
```

Note

Variables defined in a while condition or statement are created and destroyed on each iteration.

5.4.2 The traditional for statement

```
for (init-statement; condition; expression)
    statement
```

Note

Omitting any of `init-statement`, `condition` or `expression` is allowed.

5.4.3 The range for statement

```
for(declaration : expression)
    statement
```

Note

- `Expression` must represent a sequence, e.g. an array or an object that has `begin` and `end` members that return iterators.
- It must be possible to convert each element of the sequence to the type of the variable defined by `declaration`.
- To modify to elements in the sequence, the variable must be a reference type.

Hint

Use the `auto` type specifies for `declarator`.

5.4.4 The do while statement

```
do
    statement
while(condition);
```

Note

Note the semicolon after the paranthesized condition.

5.5 Jumping statements

5.5.1 The break statement

Terminates the nearest enclosing iterative statement or switch statement.

5.5.2 The continue statement

Terminates the current iteration of the nearest enclosing iterative statement and immediately begins the next iteration.

5.5.3 The goto statement

Hint

Do **not** use `goto`!

5.6 try blocks and exception handling

```
try
{
    std::cout << "Throwing an error in 3..2..1" << std::endl;
    throw "Error!";
}
catch(char const *e)
{
    std::cout << e << std::endl;
}

try
{
    std::vector v;
    std::cout << v.at(5) << std::endl;
}
catch(std::exception const &e)
{
    std::cout << "Exception occured: " << e.what() << std::endl;
}
```

Note

- Exception handling is important.
- Writing exception safe code is **hard**.

6 Functions

6.1 Basics

A function definition consists of a return type, a name, a comma-seperated list of zero or more parameters, and a body.

```
return-type name(parameter-list)
{
    function-body
}
```

A function is called using the call `(argument-list)` operator, which takes an expression that is a function or points to a function. Argument list is a comma-seperated list of zero or more arguments. The arguments are used to initialize the function's parameters. Thus the number of parameters and arguments must match. In addition the type of each argument must match the corresponding parameter. The type of a call expression is the return type of the function. Execution of a function is ended by a return statement.

6.1.1 Local static objects

Objects defined in a function body go out of scope when the function ends. In some cases it might be usefull to have a variable that is not destroyed when the function ends, instead its lifetime coninues across calls to the function. We obtain such object by defining a local variable as `static`. Each local static object is initialized before the first time a function is executed and are not destroyed when te program terminates. I.e. can be used to count how often a function has been executed or to store a runtime dependent constant.

```
unsigned int count_calls()
{
    static unsigned int count = 0;
    return ++count;
}
```

Note

A local static variable with no explicit initializer is value initialized. This means initialized to zero for built-in types.

6.1.2 Function declarations

As with variables a function can be declared several times, but only be defined once. A function must be declared before we can use it. A function declaration is just like a function definition except that a declaration has no function body. Parameter names can be omitted, but migth be helpful for users to understand the function. A function declaration ends with a semicolon.

```
void swap(int, int);
```

Hint

- Functions are declared once in headers files and defined in source files.
- The header that declares a function has to be included in each source file using the function.
- The header that declares a function should be included in the source file that defines that function.

6.1.3 Seperate Compilation

More complex applications should be seperated not only in a header and source file, but in several header and source files. An application with several source files can be compiled by simply passing all source files as an argument to g++.

```
g++ main.cpp functions.cpp
```

6.2 Argument Passing

6.2.1 General

Each time we call a function its parameters are create and initialized by the arguments passes in. If the parameter is a reference, the paramter is bound to its argument ("passed by reference"). Otherwise the argument's value is copied ("passed by value"), hence the parameter and argument are independent.

Hint

- Prefer passing arguments by reference to avoid copies.
- Pass by value if you need a local copy.
- Reference parameters that are not changed inside a function should be references to const.

Note

Top-level consts are ignored for function parameters.

6.2.2 Passing arrays

We can not pass an array by value as it cannot be copied. When we use an array it is usually converted to a pointer, hence we typically pass an array to a function by a pointer to the array's first element. Still we can write a paramter that looks like an array:

```
void print(int const*);
void print(int const[]);
void print(int const[2]);
```

All three declarations are equivalent, the paramter is always of type `const int*`.

Hint

Use pointer to const if the array is not to be changed.

As with any code that uses arrays, functions need to ensure that all access to the array are within the array bounds. Passing arrays by pointers, the size of the array is not known. Thus additional information is required to avoid out of bound errors:

Using a marker to mark the end of an array

Classic C-style character strings use this approach. These arrays end with a null character marking the end of the string. Function have to check each character and stop processing when they see a null character.

Using Standard Library conventions

Passing a pointer to the first element of an array and one past the last element in the array, we can iterate through the array incrementing the pointer to the first element until it points to one past the last element.

Explicitely passing a size parameter

A common approach in C programs is to pass a pointer to the first element and a second parameter that indicates the size of the array.

Array reference parameters

An array can be passed by reference, however the type looks slightly odd and might be misunderstood.

```
void print(int (&arr)[10])
{
    for (auto i : arr)
    {
        std::cout << i << std::endl;
    }
    return;
}
```

This allows use to use `std::begin()` and `std::end()`, and hence also to use a range for. However the function is now limited to arrays of size 10. We will later learn how to remove this limit.

6.2.3 Handling command line options

The main function itself passes an array. So far we have always used:

```
int main() { ... }
```

But we might want to pass arguments to our application, hence we use

```
int main(int argc, char *argv[]) { ... }
```

where `argv` is an array of pointers to C-style character strings. The first element of the array either points to the name of the program or to an empty string. Subsequent `(argc-1)` elements pass the arguments provided on the command line. The element just past the last pointer is guaranteed to be 0.

6.2.4 Functions with varying parameters

We can pass a varying number of parameters to a function if all the arguments have the same type. Therefor we use a parameter of `initializer_list` type. The `initializer_list` is template type, hence we must specify the type of the elements.

```
void print(std::initializer_list<std::string> msg)
{
    for(auto s : msg)
    {
        std::cout << s << " ";
    }
    std::cout << std::endl;
}

int main()
{
```

```
    print({"Hello,", "who", "are", "you?"});
}
```

6.3 Return statement and types

There are two forms of `return` statements:

```
return;
return expression;
```

Note

- A return with no value may only be used in a function that has a return type of void.
- In a void function an implicit return takes place after the function's last statement.
- The value returned must have the same type as the function return type, or a type that can implicitely converted to it.

Hint

Every function should end with a return statement.

How values are returned

Values are returned exactly the same way as variables and parameters are initialized.

Hint

- Never return a reference to a local object.
- Only return a reference to a preexisting object.
- We can assign to the result of a function that returns a reference to non`const`.

6.4 Overloaded functions

Functions that have the same name but different parameter list and appear in the same scope are overloaded.

Note

Top-level `const` is ignored. A parameter with and without top-level `const` is indistinguishable.

Hint

All functions with the same name should perform the same general action, but apply to different paramter types.

Information

Function names do not overload accross scopes.

6.5 Special features

6.5.1 Default Arguments

In some cases we might want to have default values for some of the function's parameters. This can be achieved by specifying an initializer for a parameter in the parameter list.

```
int func(int a, int b, int c = 15, int d = 20);

func(1, 2);
func(1, 2, 3);
func(1, 2, 3, 4);
```

Warning

Default arguments can only be used for the trailing (right-most) arguments.

Hint

Default arguments ordinarily should be specified with the function declaration in an appropriate header.

6.5.2 Inline functions

An inline function avoids function call overhead.

```
inline int square(int);
```

Note

The `inline` specification is only a request to the compiler.

6.5.3 constexpr functions

A `constexpr` function can be used in a constant expression. It is defined like any other function, but restricted to: The return type and the type of each parameter in it must be a literal type, and the function body must contain exactly one return statement.

```
constexpr int solution()
{
    return 42;
}

constexpr int scale(int s)
{
    return solution() * s;
}
```

Note

Inline and constexpr functions can be defined multiple times in a program.

Hint

Define constexpr and inline functions in header files.

6.5.4 The assert

The assert is a preprocessor macro. The assert macro takes a single expression which it uses as a condition:

```
#include <cassert>
assert(expr);
```

If the evaluated expression is false, assert writes a message and terminates the program. If the expression is true, assert does nothing.

Note

- The assert can be disabled by defining the preprocessor variable NDEBUG.
- See `static_assert` for compile-time assertion checking.

6.6 Pointers to functions

```
bool isValid(string const&);
bool (*pf)(string const&) = isValid;
```

Note

The paranteses around `*pf` are necessary.

7 Direct Dynamic Memory

Warning

Although necessary at times, dynamic memory is notoriously tricky to manage correctly.

So far we have mainly used static or stack memory. Objects allocated in static or stack memory are automatically created and destroyed by the compiler. In addition to static or stack memory, every program has a pool of memory that it can use — the heap. Programs use the heap for objects that they dynamically allocate. Dynamic allocation happens at runtime and the program controls the lifetime of dynamic objects.

7.1 Dynamically allocate objects

To allocate (and initialize) an object in dynamic memory we use `new`, which returns a pointer to the new dynamic object. To destroy an object when it is not needed anymore we use `delete` which takes a pointer to a dynamically allocated object.

```
int *iVal = new int();
delete iVal;
```

Note

- Dynamically allocated objects that are never destroyed exist until the program is terminated.
- `delete` can only destroy dynamically allocated objects.
- `delete nullptr` is allowed.

Hint

- Initialize dynamically allocated objects.
- Destroy a dynamically allocated object when not required anymore.
- Pointers to freed dynamically allocated object are dangling pointers.
- Reset the value of a dangling pointer to `nullptr` if it does not go out of scope immediately after `delete`.

Warning

- You are in charge of managing the lifetime of a dynamically allocated object.
- `new` may fail. In this case it returns a `nullptr`.
- It is an error to access a pointer to a destroyed object (dangling pointer).

Hint

Use smart pointers! Smart pointers manage dynamic objects.

7.2 Dynamically allocate arrays

As we can dynamically allocate any object, we can also dynamically allocate an array of objects.

```
int * iArr = new int[arraySize()]();
delete [] iArr;
```

Note

Dynamic allocation of an array returns a pointer to the first element, hence it is not of type array.

Information

It is allowed to dynamically allocate zero sized arrays.

Hint

Prefer containers to dynamically allocated arrays.

8 Classes

The fundamental idea behind classes are data abstraction and encapsulation. Data abstraction is a programming technique that focuses on the interface to a type. It lets programmers ignore the details of how a type is represented and instead only think about the operations a type can perform. Encapsulation enforces the seperation of a class' interface and implementation. A class that is encapsulated hides its implementation — users of the class can use the interface but have no access to the implementation.

A class that uses data abstraction and encapsulation defines an abstract data type. In an abstract data type, the class designer worries about how the class is implemented. Class users need not know how the type works. They can instead think abstractly about what the type does.

8.1 Defining abstract data types

We have already seen how to define our own data type using the `struct` keyword. It lets users access its data members and forces users to write their own operations. An abstract data type defines operations for users, hence we have to add member functions to the data type. Such abstract data types are named class. Each class defines its own new scope with the same name as the class.

```
struct Coordinates
{
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
    float getX()
    {
        return x;
    }
    float getY();
};
```

8.2 Defining member functions

Every member function must be declared inside its class, but can be defined either inside or outside of the class body. To define a function outside of the class body we have to specify the class scope, i.e.

```
float Coordinates::getY()
{
    return y;
}
```

Introducing this pointer

We call a member function by using member (dot) operator to fetch the member of a an object, which we then call.

```
Coordinates mensa;
mensa.getX();
```

Thus member functions are always called on behalf of an object. The member function `getX()` is implicitly returning `mensa.x`.

Member functions access the object on which they where an extra, implicit parameter `this`. When we call a member function, `this` is initialized with the address of the object on which the function was invoked. We can use `this` inside the body of a member function, hence we could rewrite `getY()`:

```
float Coordinates:getY()
{
    return this->y;
}
```

Note

- Any direct use of a member of the class is assumed to be an implicit reference through `this`.
- `this` is intended to always refer to "this" object, hence it is a `const` pointer.

Warning

It is illegal to define a parameter or variable named `this`

const member functions

We have seen before that any parameter of a function that does not modify the value should be `const`. The same applies to member functions. In case a member function does not modify member data we want `this` to be a pointer to `const`. However as the `this` pointer is an implicit parameter we can not change it to be `const` like any other parameter. Thus to get a `this` pointer to const we declare the member function as `const` by putting a `const` after the parameter list.

```
class Coordinates
{
    ...
    float getX() const;
};
```

Defining nonmember class-related functions

Functions that are conceptually part of a class, but are not member functions.

Hint

Declare class-related functions in the same header as the class.

Member function or class-related function?

Only functions that modify member data and getter functions should be member functions.

8.3 Encapsulation and access control

We have learned how to add operations to a class, and hence how to define an interface. However nothing forces users to use that interface. Our class is not yet encapsulated — users can access everything and directly modify member data. We use access specifiers to enforce encapsulation:

- public: Members defined after a public specifier are accessible to all parts of the program. The public members define the interface to the class.
- private: Members defined after a private specifier are accessible to the member functions of the class, but are not accessible to code that uses the class. The private section encapsulate (hide) the implementation.

```
class Coordinates
{
    public:
        float getX() const;
        float getY() const;
        float getZ() const;
        float& setX(float);
        float& setY(float);
        float& setZ(float);

    private:
        float x = 0.0;
        float y = 0.0;
        float z = 0.0;
};
```

We now use the keyword `class` instead of `struct`. The only difference is the default acces level for members defined before the first access specifier. For `struct` these member are public, for `class` private.

Note

The only difference between using class and using struct to define a class is the default access level.

8.4 static class members

Static class members are associated with the class, not with inidividual objects of the class type. We add a member associated with the class adding the keyword static. Static members can be accessed directly using the scope operator or through objects using the member operator.

Note

The keyword static appears only with the declaration inside the class body.

8.5 Overloading operators

Operator overloading lets us define the meaning of an operator when applied to operand(s) of a class type. Operator overloading can make our programs easier to write and read.

Overloaded operators are functions with special names — the keyword `operator` followed by the symbol of the operator. An overloaded operator function has the same number of parameters as the operator has operands. See cppreference.com operators' description for operator function prototype examples.

Note

- Precedence and associativity of an operator can not be changed.
- An operator function must either be a member of a class or have at least one parameter of class type.
- For a binary operator the left-hand operand is passed to the first parameter, the right-hand operand is passed to the second paramter.
- If an operator function is a member function, the first (left-hand) operand is bound to the implicit `this` pointer. Thus member operator function have one less (explicit) parameter than the number of operands.
- Almost all operators can be overloaded. `::`, `.*`, `.`, and `?:` can not be overloaded.

Member or nonmember implementation?

- The assignment, subscript, call and member access arrow operators must be defined as members.
- The compound assignment operators ordiniarily outght to be members.
- Operators that change the state of their object or that are closely tied to their given type — increment, decrement, and dereference — should be members.
- Symmetric operators — those that might convert either operand, such as arithmetic, equality, relational, and bitwise operators — should be defined as ordinary nonmember functions.
- IO operators must be nonmember functions.

Hint

- The comma, address-of, logical AND, and logical OR operators should not be overloaded.
- Use definition that are consistent with the built-in meaning.
- Classes that define both an arithmetic operator and the related compound assignment ought to implement the arithmetic operator by using the compound assignment.
- Classes for which there is a logical meaning for equality should define `operator==`.
- One of the equality or inequality operators should delegate the work to the other.
- (Compound) assignment operators should return a reference to the left-hand operand.
- If a class has a subscript operator, it should define two versions — one that returns a plain reference and the other that is a const member and returns a reference to const.
- Classes that define increment or decrement operators should define both the prefix and postfix versions. The prefix operators should return a reference to the incremented or decremented object. The postfix operators should the old value as a value.

8.6 Constructors

Each class defines how objects of its type can be initialized. Object initialization is controlled by defining one or more special member functions — constructors. A constructor is intended to initialize the data members of a class object. A constructor is run whenever an object is created.

Constructors have the same name as the class. Constructors have no return type, a (possible empty) parameter list, and a (possible empty) function body. Thanks to function overloading a class can have several constructors different in number or types of their parameters.

Note

- Same name as the class.
- No return type.
- Can not be declared `const`.
- A `const` object is non`const` until after the constructor completes.

The default constructor

If no constructor is defined the compiler will implicitly define a synthesized default constructor. The default constructor is the constructor that takes no argument. The synthesized default constructor initializes each data member. If an in-class initializer is supplied it is used by the synthesized default constructor, otherwise the member is default-initialized.

Note

A compiler generates a default constructor only if a class declares no constructors.

Warning

Classes that have members of built-in or compound type usually should rely on the synthesized default constructor only if all such members have in-class initializers.

Hint

If you define any constructor you have to define the default constructor.

Example constructors

```
class Coordinates
{
    public:
        Coordinates() = default;
        Coordinates(float x, float y, float z): mX(x), mY(y), mZ(z) {}
        Coordinates(std::istream &);
        float getX() const;
        float getY() const;
        float getZ() const;
        float& setX(float);
        float& setY(float);
        float& setZ(float);

    private:
        float mX = 0.0;
        float mY = 0.0;
        float mZ = 0.0;
};
```

Constructor initializer list

In the example the second constructor uses a constructor initializer list to initialize member data. The constructor initializer list follows the parameter list seperated by a colon. It is a list of member names, eaach of which is followed by that member's initial value in paranteses (or inside curly braces), seperated by commas. When a member if ommited from the constructor initializer list, it is implicitly initialized using the same process as is used by the default constructor.

We can delegate a constructor to another constructor using constructor initializer list.

...

```
Coordinates(): Coordinates(0.0, 0.0, 0.0) {}
...
```

Note

- We must use the constructor initializer list (or in-class initializers) for members that are const, reference, or of a class type that does not have a default constructor.
- Members are initialized in the order in which they appear in the class definition.
- A constructor that supplies default arguments for all its parameters also defines the default constructor.

Hint

- If we have to define the default constructor only because we want to provide other constructors, we can ask the compiler to generator the synthsized default constructor by writing `= default` after the parameter list.
- Constructors should not override in-class initializers except to use a different initial value.
- Every member with no in-class initializer should be explicitly initialized by all constructors.
- Avoid using member to initialize other members.
- Write constructor initilaizers in the same order as the members are declared.

8.7 Copy control

Classes control how they are copied, assigned, or destroyed.

8.7.1 The copy constructor

The copy constructor defines what happens when an object is initialized from another object of the same type. A constructor is the copy constructor if its first paramter is a reference to the class type and any additional parameters have default values. If no copy constructor is defined, the compiler synthesizes one. The synthesized copy constructor memberwise copies the member of its arguments into the object being created.

```
Coordintes::Coordinates(const Coordinates &orig):
    mX(orig.mX), mY(orig.mY), mZ(orig.mZ) {}
```

8.7.2 The copy-assignment operator

The copy assignment operator defines what happens when we assign an object of a class type to another object of the same class type. If no copy-assignment operator exists, the compiler synthesizes one. The synthesized copy-assignment operator assigns each nonstatic member of the right-hand object to the corresponding member of the left-hand object using the copy-assignment operator or the type of that member.

```
Coordinates::Coordinates& operator=(const Coordinates& rhs)
{
    mX = rhs.mX;
    mY = rhs.mY;
    mZ = rhs.mZ;
```

```
    return *this;
}
```

Hint

Assignment operators should return a reference to their left-hand operand.

8.7.3 The destructor

The destructor operates inversely to the constructors — they do whatever work needed to free the resources used by an object and destroy the nonstatic data member of the function. If no destructor is defined the compiler synthesizes one. The synthesized destructor has an empty function body. After the function body is executed the members are destroyed. Members of class type are destroyed by running the member's own destructor. The built-in types do not have destructors, they are destroyed implicitly.

```
Coordinates::~Coordinates() {}
```

Warning

The implicit destruction of a member of built-in pointer type does not delete the object it points to.

Rule of Three/Five

Copy constructor, copy-assignment operator and destructor should be thought of as a unit.
If you need to define one, you typically need to define them all.

Note

- We can ask the compiler to generate the synthesized version for any member function that have a sythesized version using `= default`.
- We can prevent copying or assignment by deleting the corresponding member function. This is done by adding `= delete` after the member function's parameter list, to the member function declaration. Any member function can be deleted this way in general.

Warning

The destructor should not be deleted

9 Inheritance

Classes related by inheritance form a hierarchy. Typically there is a base class at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as derived classes. Derived classes inherit the members of their base class. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

9.1 Defining base and derived classes

In many, but not all, ways base and derived classes are defined like other classes.

9.1.1 Defining a base class

```
class Book
{
    public:
        Book() = default;
        Book(std::string title, std::string author): mTitle(title), mAuthor(author) {}
        void setTitle(std::string);
        std::string getTitle() const;

        void setAuthor(std::string);
        std::string getAuthor(std::string);

        virtual void setPrice(double);
        virtual double price() const
        { return mPrice; }

        virtual ~Book() = default;

    protected:
        double mPrice = 0.0;

    private:
        std::string mTitle;
        std::string mAuthor;
}
```

protected

For base classes we have a new access control specifier — protected. Derived classes can access any public member of the base class, just like anybody else, and in addition access all protected members.

virtual

A base class must distinguish functions it expects the derived classes to override from those functions it expects its derived classes to inherit without change. The base class defines functions it expects the derived classes to override as `virtual`.

Note

- Any nonstatic member function, other than a constructor may be virtual.
- The virtual keyword appears only on the declaration inside the class.
- A function that is declared virtual in the base class is implicitly virtual in the derived class.

Hint

The destructor of a base class should be virtual.

9.1.2 Defining a derived class

```
class PaperBook : public Book
{
    public:
        PaperBook() = default;
        PaperBook(std::string title, std::string author): Book(title, author) {}

        virtual double price() const final override
        { return mPrice + mShippingCost; }

        void setShippingCost(double);
        double getShippingCost();

        ~PaperBook() = default;
    private:
        double mShippingCost = 0.0;
}

class EBook final : public Book
    public:
        EBook() = default;
        EBook(std::string title, std::string author): Book(title, author) {}
        EBook& operator=(const EBook& orig)
        { Book::operator=(orig); mSizeMB = orig.mSizeMBM }

        void setSizeMB(double);
        double getSizeMB();

        ~EBook() = default;
    private:
        double mSizeMB = 0.0;
}
```

Respecting the Base-Class Interface

It is essential to understand that each class defines its own interface. Interactions with an object of a class-type should use the interface of that class, even if that object is the base-class part of a derived object.

Derivation list

A derived class must specify from which class(es) it inherits. This is done in the class derivation list that follows the class name seperated with a colon. The derivation list is a comma seperated list of base class names where each base class name is preceded by an access control specifier. The access control specifier may be used to control the access control to the base class members through the derived class.

Member inheritation

The derived class inherits all members of the base class, expect constructors, desctructor and copy-asignment operator.

Override and final

The `override` specifier can be used in the derived class to ensure that we properly override a virtual function of the base class. The `final` specifier can be used for member function and the class itself. Members functions declared as `final` can not be overwritten by derived classes. A class declared as `final` can not be used as a base class at all.

Hint

Virtual functions that have default arguments should use the same argument values in the base and derived classes.

Derived-to-Base Conversion

A derived object, apart of its own members, contains all members of the base class. Thus we can use an object of a derived type as if it where an object of its base type(s). In particular, we can bind a base-class reference or pointer to the base class part of a derived object.

```
Book item;                  // object of base type
EBook hobbit;               // object of derived type
Book *pHobbit = &hobbit;    // Pointer to the Book part of hobbit
Book &rHobbit = hobbit;     // Reference to the Book part of hobbit
```

Hint

A derived class should be useable in any place the base class is used.

9.2 Polymorphism

When we call a function defined in a base class through a reference or pointer to the base class, we do not know the type of the object on which that member is executed. The object can be a base-class object or an object of a derived class. If the function is virtual, then the decision as to which function to run is delayed until run time. The version of the virtual function that is run is the one defined by the type of the object to which the reference is bound or to which the pointer points. On the other hand, calls to nonvirtual functions are bound at compile time. Similarly, calls to any function (virtual or not) on an object are also bound at compile time. The type of an object is fixed and unvarying—there is nothing we can do to make the dynamic type of an object differ from its static type. Therefore, calls made on an object are bound at compile time to the version defined by the type of the object.

Note

Virtuals are resolved at run time only if the call is made through a reference or pointer. Only in these cases is it possible for an object's dynamic type to differ from its static type.

Warning

Classes used in polymorphism must have a virtual destructor.

9.3 Abstract base classes

Virtual function declared in the class header using = 0 just before the semicolon are pure virtual functions. A pure virtual function need not be (but may be) defined. Classes with pure virtuals are abstract classes. An abstract base class defines an interface for subsequent classes to override. We cannot (directly) create objects of a type that is an abstract If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well.

```
class Book
{
    public:
        Book() = default;
        Book(std::string title, std::string author): mTitle(title), mAuthor(author) {}
        void setTitle(std::string);
        std::string getTitle() const;

        void setAuthor(std::string);
        std::string getAuthor(std::string);

        virtual void setPrice(double);
        virtual double price() const = 0;

        virtual ~Book() = default;

    protected:
        double mPrice = 0.0;

    private:
        std::string mTitle;
        std::string mAuthor;
}
```

9.4 Inheritance vs. Containment or "When to use inheritance?"

Use inheritance when...or ...

- At least one function of the base class is overriden.
- You want to use polymorphism.
- A library requires you to inherit from of its classes to gain access to certain funtionality.

10 Generic Programming

In generic programming we write the code in a way it is independent of any particular type. When we use a generi program, we supply the type(s) or value(s) on which that instance of the program will operate.

Templates are the foundation of generic programming. We already have used templates without exaclty knowing and understanding how they are defined. A template is a blueprint or formula for creating fcuntion or casses. When we use a generic function or type, we supply the information needed to transform that blueprint into a specific function or class. That transformation happens during compilation.

10.1 Defining a Function Template

A template definition start with the keyword `template` followed by a template parameter list, which is a comma-seperated list of one or more template parameters bracketed by the less/greater-than tokens.

```
template <typename T>
void swap(T &a, T &b)
{
    T c;
    c = a;
    a = b;
    b = c;
    return;
}
```

Note

In a template definition, the template parameter list cannot be empty.

Instantiating a function template

When we all a function template, the compiler (ordinarily) uses the arguments of the call to deduce the template argument(s) for us. That is, when we call `swap`, the compiler uses the type of the arguments to determine what type to bind the template parameter `T`. The compiler uses the dedued template parameter(s) to instantiate a specific version of the function. When the compiler instantiates a template, it creates a new "instance" of the template using the actual template argument(s) in place of the corresponding template parameter(s). I.e. when calling the `swap` function with two arguments of type `int` the compiler creates a new funtion

```
void swap(int &a, int &b)
{
    int c;
    c = a;
    a = b;
    b = c;
    return;
}
```

The compiler instantiates as many different versions of the template function as required.

Template Type Parameters

A template type parameter can be used as a type specifier in the same way as a built-in or class type specifier. In particular, a type parameter can be used to name the return type or a function parameter type, and for variable declarations or casts inside the function body.

Nontype Template Parameters

We can define templates that take nontype parameters. A nontype parameter represents a value wather that a type. Nonetype parameters are specified by using a specific type name instead of the typename keyword. A template nontype parameter is a constant value inside the template definition.

```
template<typename T, int N>
void print (T (& arr)[N])
{
    for (auto i : arr)
    {
        std :: cout << i << std :: endl;
    }
    return ;
}
```

Note

- Template arguments used for nontype template parameters must be constant expressions.

Writing Type-independent Code

Hint

- Use reference to `const` instead of call by value. Call by value restricts the function to copyable types.
- Template functions should try to minimize the number of requirements plaed on the arguments.

Template Compilation

Ordinarily, when we call a function, the compiler needs to see only a declaration for the function. Similarly, when we use objects of class type, the class definition must be available, but the definitions of the member functions need not be present. As a result, we put class definitions and function declarations in header files and definitions of ordinary and class-member functions in source files.

Templates are different: To generate an instantiation, the compiler needs to have the code that defines a function template or class template member function. As a result, unlike nontemplate code, headers for templates typically include definitions as well as declarations.

Note

Definitions of function templates and member functions of class templates are ordinarily put into header files.

Compilation Errors Are Mostly Reported during Instantiation

Due to the fact that code is not generated until a template is instantiated affects when we learn about compilation errors in the code inside the template. In general there are thress stages during which the compiler might flag an error.

- Compilation of the template itself: The compiler can't detect many errors as the type is still unknown. Thus it can mainly only detect syntax errors.
- Call to a function template: The compiler checks that the number and type of the arguments is appropriate.
- Instantiation of a template: After instantion the type is known, hence the compiler can finally detect any error.

Template parameter assumptions

Although the code may not be overtly type specific when we write a template, the template code usually makes some assumption about the types that will be used.

Warning

It is up to the caller to guarantee that the arguments passed to the template support any operations that template uses, and that those operations behave correctly in the context in which the template uses them.

10.2 Defining a class template

Like function templates, class templates begin with the keyword `template` followed by a template parameter list.

```
template <typename T>
class Coordinates
{
    public:
        Coordinates() = default;
        Coordinates(T x, T y, T z): mX(x), mY(y), mZ(z) {}
        Coordinates(std::istream &);
        T getX() const;
        T getY() const;
        T getZ() const;
        T& setX(T);
        T& setY(T);
        T& setZ(T);

    private:
        T mX = 0;
        T mY = 0;
        T mZ = 0;
};
```

Instantiating a class template

Class templates differ from function templates in that the compiler cannot deduce the template parameter type(s). Thus, as we've already seen using `std::vector`, we must supply additional information inside angle brackets following the template's name. The extra information is the list of template arguments to use in place of the template parameters. The compiler uses these to instantiate a specific class from the template. The compiler generates a different class for each element type we specify.

```
Coordinates<float> mensa;
```

Note

Each instantiation of a class template constitutes an independent class.

Member Functions of Class Templates

As with any class, we can define the member functions of a class template either inside or outside of the class body.

A class template member function is itself an ordinary function. However, each instantiation of the class template has its own version of each member. As a result, a member function of a class template has the same template parameters as the class itself. Therefore, a member function defined outside the class template body starts with the keyword template followed by the class' template parameter list.

As usual, when we define a member outside its class, we must say to which class the member belongs. Also as usual, the name of a class generated from a template includes its template arguments. When we define a member, the template argument(s) are the same as the template parameter(s).

```
template <typename T>
T& Coordinates<T>::setX(T x)
{
    mX = x;
    return mX;
}
```

Instantiation of Class-Template Member Functions

By default, a member function of a class template is instantiated only if the program uses that member function. Thus we might instantiate a class with a type that may not meet the requirements for some of the template's operations.

Note

By default, a member of an instantiated class template is instantiated only if the member is used.

Simplifying Use of a Template Class Name inside Class Code

There is one exception to the rule that we must supply template arguments when we use a class template type. Inside the scope of the class template itself, we may use the name of the template without arguments.

However when we define members outside the body of a class, we must remember that we are not in the scope of the class until the class name is seen.

Note

Inside the scope of a class template, we may refer to the template without specifying template argument(s).

Template Type Aliases

We can define a type alias for a class template:

```
template<typename T> using twin = pair<T, T>;
template<typename T> using iD = pair<T, unsigned>;
```

Member Templates

A class — either an ordinary class or a class template — may have a member funtcion that is itself a template. Such member are referred to as member templates.

Warning

A member template may not be virtual.

10.3 Default Template Arguments

Just as we can supply default arguments to function parameters, we can also supply default template arguments to both function and class templates.

Whenever we use a class template, we must always follow the template's name with brackets. The brackets indicate that a class must be instantiated from a template. In particular, if a class template provides default arguments for all of its template parameters, and we want to use those defaults, we must put an empty bracket pair following the template's name.

```
template<typename T = float>
class Coordinates
{ ... };

template<typename T, typename F = std::less<T>>
int compare(T const &v1, T const &v2, F f = F())
{ ... }
```

Note

Angle brackets are always required for template class instantiation, even if empty.

10.4 Function-Template Explicit Arguments

In some situations, it is not possible for the compiler to deduce the types of the template arguments. In others, we want to allow the user to control the template instantiation. Both cases arise most often when a function return type differs from any of those used in the parameter list. In this case we have to supply an explicit template argument to a call the same way that we define an instance of a class template.

```
template<typename T1, typename T2, typename T3>
T1 sum(T2, T3);

iVal = sum<int>(lVal, iVal);
```

10.5 Trailing Return Types

In some cases we might have a return type depending on the template paramter types and do not want the user to have to specify the return type explicitely. In this case we can use trailing return types.

```
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    ...
    return *beg;
}
```

10.6 Template Specializations

It is not always possible to write a template that is best suited for every possible template argument. In some cases, the general template definition is simply wrong. In other cases we might take advantage of some specific knowedge to write more efficient code. In both cases, we can define a specialized version of the function or class template.

To indicate that we are specializing a template, we use the keyword `template` with an empty parameter list. The empty paramter list indicates that arguments for all the template parameters of the originla template will be supplied.

```
template<>
int compare(Coordinates const &a, Coordinates const &b)
{ ... }
```

Note

We can partially specialize a class template.

11 Namespaces

Large programs tend to use independently developed libraries. Such libraries also tend to define a large number of global names, such as classes, functions, and templates. When an application uses libraries from many different vendors, it is almost inevitable that some of these names will clash. Libraries that put names into the global namespace are said to cause namespace pollution

Namespaces provide mechanism for preventing name collisions. Namespaces partition the global namespace. A namespace is a scope. By defining a library's names inside a namespace, library authors (and users) can avoid the limitations inherent in global names.

11.1 Definition

A namespace definition begins with the keyword `namespace` followed by the namespace name. Following the namespace name is a sequence of declarations and definitions delimited by curly braces.

```
namespace Welt
{
    class Welt{ ... };
    void print(std::string);
}
```

Note

- Each namespace is a scope.
- A namespace scope does not end with a semicolon.
- Namespaces may be defined at global scope or inside another namespace, but not inside a function or class.
- Namespaces can be discontiguous, each definition adds to the previous.
- Template specialization must be declared in the same namespace that contains the original template.
- Unlike other namespaces, a unnamed namespace is local to a particular file and never spans multiple files.

Warning

The use of file static declarations is deprecated by the C++ standard. File statics should be avoided and unnamed namespaces used instead.

11.2 Using namespace members

Namespace alias can be used to associate a shorter synonym with a namespace.

```
namespace lnn = langerNamespaceName;
namespace nn = nested::nspace;
```

We can use `using` declarations to introduce only one namespace member at a time. It allows us to be very specific regarding which names are used in our programs.

```
using std::cout;
```

The `using` directive, like a `using` declaration alows us to use the unqualified form of a namespace name. Unlike a using declaration, we retain no control over which names are made visible — they all are.

```
using namespace std;
```

Warning

Providing a using directive for namespaces, such as std, that our application does not control reintroduces all the name collision problems inherent in using multiple libraries.

Hint

Avoid `using` directives.

Part II

Qt

12 Overview

Qt is a cross-platform application framework. It is mainly used for developing applications with graphical user interfaces (GUIs). Qt is available under a commercial license and GPL/LGPL that are suitable for free/open source software.

Modules

Qt supports many modules for various applications. Of our main interest is the QtWidget module to create GUIs.

Projects using Qt

- KDE, LXQt, Hawaii,…
- Texmaker/TeXStudio
- WebOS
- Autodesk Maya
- CryEngine
- Spotify (Linux)
- VLC media player

Documentation

doc.qt.io

13 Qt Creator

Typically Qt Creator, an integrated development environment (IDE), is used to develope applicatioans that use the Qt application framework. We use Qt Creater to create and manage our project(s). A project is defined using a `.pro` file. In this file we can set a few options:

- The application type (`TEMPLATE`)
- The application name (`TARGET`)
- Lists os used source and header files (`SOURCES` and `HEADERS`)
- Various configurations (`CONFIG += c++11`)
- Used Qt modules (`QT`)
- Used non-Qt libraries (`LIBS`)
- Preprocessor defines (`DEFINES`)

Note

Disable “Generate form” in the project creation dialog.

14 The main routine

Each Qt application has exactly one object of the class `QApplication`. The class takes care of input arguments, the event loop, and many other things. The event loop waits for user input. The main event loop is started using the member function `exec()` and can be stopped using `quit`. Typically the return value of `exec()` is used as the return value of `main()`. To quite the event loop the global pointer `qApp` to the only object of type `Qapplication` can be used.

Before we start the event loop we initialize one Qt object and use its member function `show()` to actually draw it, our application's main/first window. This Qt object typically either inherits from QWidget or QMainWindow. QWidget is the most general Widget that allows for full customization. QMainWindow (which itself inherits from QWidget) provides some functionaly that might be useful for an apllications main window, i.e., it defines a layout that contains a menu bar, a status bar, docks and a central widget (see http://doc.qt.io/qt-5/mainwindow.html for details). This process is automated by Qt Creator, we simply have to decide which one to use when creating a project and then implement its functionaliy. We do not have to take care of the main routine.

15 Properties

Each Qt class has plenty of member data. In Qt we refer to member data that is used to customize an object as "property". Setters and getters are defined for each property. The setter is named `set`popertyname, the getter simply by the property name. The setters are by default void functions, the getters return by value.

15.1 The Meta Object

Qt provides a meta-object system to achieve some programming paradigms that are otherwise impossible in C++:

- Introspection — Capability of examining a type at run-time
- Asynchronous function calls

Important macros

Macros are used to mark sections that should be interpreted and translated by the meta-object compiler. E.g. the signal-slot connections and their syntax cannot be interpreted by regular C++ compiler, hence the meta-object compiler is required to translate these first. This is done by specifying the macro `Q_OBJECT` in the header containgin class definitions that use such syntax.

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
}
```

Other marker macros for the meta-object compiler are:

- `signals` sections
- `public slots`, `protected slots`, and `private slots` sections.

16 Qt class hierarchy

Qt extensively uses inheritance with `QObject` being the most basic class in Qt. Most of the classes in Qt inherit from the `QObject` class that provides some very powerful capabilities:

- Parenting system
- Signals and Slots
- Event management

All widget classes inherit from `QObject`, hence can use all of these capabilities. The most basic widget is `QWidget` that contains the basic properties to descibe a window or a widget. Nearly all graphical elements inherit from QWidget. The inheritace is done in order to faciliate properties management. E.g. the `QAbstractButton` provides basic properties that are shared among all buttons.

16.1 Parenting system

Any Qt object that inherits from `QObject` can have a parent and children. Thus all Qt objects are typically part of an hierarchy tree, which makes things convenient:

- When an object is destroyed, all of its children are destroyed.
- Child widgets in a `QWidget` automatically appear inside the parent widget.

Child widgets are typically added to a parent in its constructor.

```
Widget::Widget(QWidget *parent): QWidget(parent)
{
    QPushButton *button = new QPushButton("Hello World", this);
    button->setGeometry(50,100,100,20);
}
```

Note

Due to the parenting system it is not necessary to write a destructor to take care of dynamically allocated child widgets. If a parent is destroyed, Qt will take care of all of its children.

Warning

For statically allocated child widgets the order of appearence is important.

Hint

Always use dynamic allocation for Widgets, except the main Widget.

Note

Parenting System ! = Class Inheritance

16.2 Layouts

We have just shown that we can add widgets to any parents widget by simply creating child widgets in the contructor of the parent and specifying the parent. This way widgets are added, still I have to manually set the size and position of each widget. An alternativ, and suggested, way is to use Layouts to place child widgets in a parent widget. This is done by first creating a layout, adding widgets or layouts to the layout, and adding the layout to the parent widget. Again all this is typically done in the parent widget's constructor.

```
QGridLayout *layout = new QGridLayout;

layout->addWidget(mButton,0,0);
layout->addLayout(mLayout,1,0);

this->setLayout(layout);
```

Examples

- QBoxLayout
- QHBoxLayout - Lines up widgets horizontally
- QVBoxLayout - Lines up widgets vertically
- QGridLayout - Lays out widgets in a grid

See http://doc.qt.io/qt-5/layout.html

16.3 Using Signals and Slots

UI toolkits typically have a mechnism to detect user action, and respond to this action. There are various approaches to implement this, but basically all of them are inspired by the observer patter. Observer pattern is used when an observable object wants to notify other observers objects about a state change, e.g. a button has been pressed. Observer pattern is the complicated part of GUI developement. Qt uses the signal and slots mechanism to easily implement observer pattern.

Qt provides two high level concepts: signals and slots.

- A signal is a message that an object can send, most of the time to inform of a status change.
- A slot is a function that is used to respond to a signal.

In order to respond to a signal, a slot must be connected to a signal.

```
QObject::connect(sender, &Sender::valueChanged, receiver, &Receiver::updateValue);
```

or the old syntax:

```
QObject::connect(sender, SIGNAL (valueChanged(parameters)),
    receiver, SLOT(updateValue(parameters)));
```

Example

```
connect(mButton, &QPushButton::clicked, QApplication::instance(), &QApplication::quit);
```

Note

- Basically, signals and slots are member functions, that might or might not have arguments, but that never return anything.
- The parameters of the signal are used to transfer information to the slot. The first parameter of the signal is passed to the first parameter of the slot, etc.
- With the new syntax signals can be connected to any member function, not only slots.

Information

Signals are generally only used as signales, whereas slots may be used just like any member function.

Warning

Connecting a signal and a slot with different parameters, will result in a warning at run-time.

Features of signals and slots

- A signal can be connected to several slots.
- Many signals can be connected to a slot.
- (Signal relaying) A signal can be connected to a signal.

16.4 Creating custom signals and slots

Creating signals and slots is a trivial task and described as by the following checklist:

- Add `Q_OBJECT` macro.
- Add `signals` section and write signals declarations.
- Add `public slots`, `protected slots`, or `private slots` sections and write slots declarations.
- Write a definition for a slot as normal member function.
- Establish connections.

Hint

Using the new `connect` syntax it is not necessary anymore to use the `slots` section.

Emitting signals

Signals can be emitted at any time using the `emit` keyword.

```
emit mySignal(arguments);
```

Example: Custom slot

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();
private slots:
    void slotButtonClicked(bool checked);
private:
    QPushButton *mButton;
};

Widget::Widget(QWidget *parent): QWidget(parent)
{
    mButton = new QPushButton("Hello World", this);
    mButton->setCheckable(true);
    connect(mButton, &QPushButton::clicked, this, &Widget::slotButtonClicked);
}

void Widget::slotButtonClicked(bool checked)
{
     // Get a pointer to the sender of the signal. Use with caution!
     QPushButton *clickedButton = qobject_cast<QPushButton *>(sender());

     if(checked) {mButton->setText("Checked");}
     else {mButton->setText("Hello World");}
}
```

Example: Custom signal

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();
signals:
    void goToHell();
private slots:
    void slotButtonClicked(bool checked);
private:
    QPushButton *mButton;
    unsigned mCounter = 0;
};

Widget::Widget(QWidget *parent): QWidget(parent)
{
    mButton = new QPushButton("Hello World", this);
    mButton->setCheckable(true);
    connect(mButton, &QPushButton::clicked, this, &Widget::slotButtonClicked);
    connect(this, &Widget::goToHell, QApplication::instance(), &QApplication::quit);
}

void Widget::slotButtonClicked(bool checked)
{
     if(checked) {mButton->setText("Checked");}
     else {mButton->setText("Hello World");}
     ++mCounter;
     if(mCounter == 10) {emit goToHell();}
}
```

16.5 A small selection of widgets

A small selection of some useful widgets, check the Qt documentation for detailed descriptions.

QPushButton

A push button is used to command (by clicking the button) the computer to perform some action by emitting the signal clicked(). Typically it displays a text label, but can also display an icon. Push buttons are, by default, not checkable, but can be set so.

```
// Create a new button, has to be added to a layout or parent widget later.
QPushButton *button = new QPushButton(tr("I'm a button"));

// Set button to be checkable
button->setCheckable(true);

// Set button size policy
button->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

// Set button to be checked
button->setChecked(true);

// Connect signal to a slot
connect(button, &QPushButton::clicked, this, &Widget::pressedButton);
```

QLabel

A QLabel displays text or an image.

QLineEdit

Allows the user to enter a single line of plain text. The text can be accessed by text() and be modified by setText(). A QLineEdit may be used to only display text, in this case set is to be read-only via the member function setReadOnly(true).

QTextEdit

Same as QLineEdit, but with an unlimited number of lines. I.e. useful to program your own text editor.

...

See http://doc.qt.io/qt-5/qtwidgets-index.html for more examples.

16.6 The Event System

See http://doc.qt.io/qt-5/eventsandfilters.html for more information.
Qt allows applications to react to events such as mouse or keyboard input. In addition it has its own events that can be sent at any time (e.g. QTimerEvent). All events are objects from the abstract QEvent class. Whenever an event occurs, Qt creates an event object representing that particular event. This object is delivered to a particular instance of QObject (i.e. the active window) by calling its event() member function. The event() member function is not resposible for handling the event, instead it only calls a specific event handler depending on the event type. These event handlers are implemented as virtual member functions.

In most cases it is sufficient to re-implement the appropriate virtual member functions without having to deal with the event() function itself at all.

Example

```
void Widget::keyPressEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_0)
    {
        // Handle pressing key "0"
        return;
    }
    //....

    // Pass on other keys to base class
    QWidget::keyPressEvent(event);
}
```

Sending/Posting an event

See QCoreApplication::sendEvent() and QCoreApplication::postEvent().

The event() function

Sometimes it might be necessary to modify the event() member function, e.g. to override the default behavior of the Tab key.

```
bool Widget::event(QEvent *event)
{
    if(event->type() == QEvent::KeyPress)
    {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if(keyEvent->key() == Qt::Key_Tab)
        {
            //....
            // Return true to accept the event
            return true;
        }
    }

    // Call the base class event() member function for any other event.
    return QWidget::event(event);
}
```

QTimer

QTimer is an alternative to the QTimerEvent that, instead of using events, relies on the signal and slot mechanism. A QTimer can be used to repetitive, or only once, call a function after a certain time.

```
// Create a new timer
QTimer *timer = new QTimer(this);

// Connect its signal to a slot
connect(timer, &QTimer::timeout, this, &Widget::updatePosition)

// Start the timer to call updatePosition every 0.5 second
timer->start(500);
```

Warning

For a member function that is connected to a repetitive QTimer you must assure that the function is finished within the QTimer's timeout.

16.7 QGraphicsView

See http://doc.qt.io/qt-5/graphicsview.html for more information.
Graphics View can manage a large number of 2D graphical items. It consists of a "Scene", that is actually holding the items, and a "View" to draw the scene. See http://doc.qt.io/qt-5/qgraphicsitem.html for an overview of available QGraphicsItems.

```
// Create a GraphicsScene and set minimum scene size
QGraphicsScene *scene = new QGraphicsScene;
scene->setSceneRect(0,0,800,400);

// Create a circle at postition 0,0
QGraphicsEllipseItem *circle = new QGraphicsEllipseItem(0,0,100,100);
scene->addItem(circle);

// Create an ellipse at postition 200,200
QGraphicsEllipseItem *ellipse = new QGraphicsEllipseItem(200,200,50,100);
scene->addItem(ellipse);

// Create a GraphicsView
QGraphicsView *view = new QGraphicsView(scene);

// Add GraphicsView to existing layout
layout->addWidget(view);

// Move circle
circle->setRect(100,0,100,100);

// Does the circle collide with ellipse?
if(circle->collidesWithItem(ellipse))
{
    // ....
```

```
}
```

17 Project

Part III

Advanced C++

18 Smart Pointers

As we have seen dynamic memory is problematic because it is hard to ensure we free memory at he right time. To make using dynamic memory easier and safer, the standard library provides two smart pointer types that manage dynamic objects. A smart pointer acts like a regular pointers with the important exception that it automatically deletes the object to which it points. Thus we can e.g. dereference or check smart pointers like regular pointers. Smart pointers are template classes with one template argument. The template argument specifies the type it points to. Smart pointers and associated functions are defined in the `memory` header.

- `std::shared_ptr<T>`: Allows muliple pointers to refer to the same object.
- `std::unique_ptr<T>`: Allows only one pointer to refer to the object, i.e. it owns the object.

Supported operations

`std::shared_ptr<T> sp`	Null smart pointer that can point to objects of type `T`
`std::unique_ptr<T> up`	
`p`	Use `p` as a condition (`true` if `p` points to an object).
`*p`	Dereference `p` to get the object to which `p` points.
`p->`	Synonym for `(*p).`
`p.get()`	Returns the regular pointer to `p`. Use with caution!
`std::swap(p, q)`	Swaps the smart pointers in `p` and `q`.
`p.swap(q)`	

Note

A default initialized smart pointer holds a null pointer.

18.1 The shared_ptr template class

We can think of a `std::shared_ptr` as if it has an associated counter, usually refered to as reference count. The reference count indicates how may `shared_ptr`s point to the same object. Whenever we copy a `shared_ptr` its reference count is incremented. The reference counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed. When the reference counter of a `shared_ptr` goes to zero, it automatically frees the object that it manages. I.e. when we copy or assign a `std::shared_ptr`, each shared pointer keeps track of how many other shared pointers point to the same object.

Supported operations

`std::make_shared<T>(args)`	Returns a `shared_ptr` pointing to a dynamically allocated object of type `T` initialized using `args`.
`std::shared_ptr<T> p(q)`	`p` is a copy of the `shared_ptr q`. Increments the count in `q`.
`p = q`	`p` and `q` are `shared_ptr`s. Decrements `p`'s reference count and increments `q`'s count. Deletes `p`'s existing memory if `p`'s count goes to 0.
`p.unique()`	Return `true` if `p.use_count()` is one.
`p.use_count()`	Returns the number of objects sharing with `p`. Caution: May be a slow operation, only use to debug.

The make_shared Function

The safest way to allocate dynamic memory is to use the standard library function `make_shared`. This functions allocates and initializes an object in dynamic memory and returns a `shared_ptr` that points to that object. `std::make_shared` is a template function. The compiler is not able to deduce the type of the object we want to allocate, hence we must specify the type in angle brackets. The passed arguments are used to construct an object of the given type. This we must pass arguments to `make_shared` that fit a constructor of the specified type. Not specifying any argument, the object is value initialized.

Factory example

```
std::shared_ptr<float> factory(float init)
{
    return std::make_shared<float>(init);
}
```

Note

If you put `std::shared_ptr`s in a container, and you subsequently need to use some, but not all, of the elements, remember to erase the elements you no longer need.

Note

One common reason to use dynamic memory is to allow multiple objects to share the same state.

Using shared_ptr with new

We can initialize a `std::shared_ptr` from a regular pointer, e.g. a pointer returned by `new`.

Additional operations

`std::shared_ptr<T> p(q)`	`p` manages the object to which the regular pointer `q` points. `q` must point mmory allocated by `new`.
`std::shared_ptr<T> p(u)`	`p` assumes ownership from the `std::unique_ptr u`. Makes `u` nullptr.
`p.reset()`	Releases the ownership of the object. Resets `p` to nullptr.
`p.reset(q)`	And makes `p` to manage the object pointed to by `q`.

Note

By default, a pointer used to initialize a smart pointer must point to dynamic memory because, by default, smart pointers use delete to free the associated object.

Note

The smart pointer constructors that take pointers are `explicit`. Hence we must use the direct form of initialization.

```
std::shared_ptr<float> p(new float(5.0));
std::shared_ptr<float> p = new float(5.0);  // error! Must use direct initialization.

std::shared_ptr<float> clone(float init)
{ return new float(p); } // error! Must use direct initialization.

std::shared_ptr<float> clone(float init)
```

```
{ return std::shared_ptr<float>(new float(init)); }
```

Note

A `std::shared_ptr` can coordinate destruction only with other `shared_ptr`s that are copies of itself.

Hint

Use `std::make_shared`

Warning

It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.

Hint

Do not mix regular and smart pointers.

Warning

Use `p.get()` only to pass access to the pointer to code that you know will not delete the pointer. In particular, never use get to initialize or assign to another smart pointer.

18.2 The weak_ptr template class

A `std::weak_ptr` is a smart pointer that does not control the lifetime of the object it points to. Instead, a `std::weak_ptr` points to an object that is managed by a `std::shared_ptr`. Binding a `std::weak_ptr` to a `std::shared_ptr` does not change its reference count. An object managed by `std::shared_ptr` will be deleted even if there are `std::weak_ptr`s pointing to that object.

Supported operations

`std::weak_ptr<T> w`	Null `std::weak_ptr` that can point to objects of type `T`.
`std::weak_ptr<T> w(s)`	`std::weak_ptr` that points to the same object as `std::shared_ptr s`.
`w = p`	`p` can be `std::shared_ptr` or `std::weak_ptr`.
`w.reset()`	Makes `w` nullptr.
`w.use_count()`	Number of `std::shared_ptr`s that point to the object `w` points to.
`w.expired()`	Returns `true` if `w.use_count()` is zero.
`w.lock()`	Returns a `std::shared_ptr` pointing to the object `w` points to or a `null std::shared_ptr` if `w.expired()` is `true`.

18.3 The unique_ptr template class

A `std::unique_ptr` "owns" an object. Only one `std::unique_ptr` can point to a given object. The object it points to is destroyed when the pointer itself is destroyed.

Supported operations

`std::unique_ptr<T> u(q)`	`p` manages the object to which the regular pointer `q` points.
`u = nullptr`	Deletes the object `u` points to.
`u = std::move(v)`	Moves the objects from `unique_ptr v` to `unique_ptr u`. `unique_ptr v` afterwards points to null, i.e. does not own any object.
`u.release()`	Releases the ownership of the managed object. Returns a regular pointer to the object. `u` is nullptr afterwards.
`u.reset()`	Deletes the object `u` manages.
`u.reset(q)`	Replace the managed object by the object the regular pointer `q` points to.

Hint

A similar function, `std::make_unique`, to `std::make_shared` has been introduced in C++14.

Note

Two `unique_ptr`s cannot refer to the same object, hence `unique_ptr`s cannot be copied.

Move semantics

To take over ownership we use move semantics. Very simplified:

```
u = std::move(v);
```

19 Basic OO Design Principles

19.1 Key principles

- **Seek for low coupling and high cohesion**
  Minimize dependencies. Dependencies complicate changes/modifications.
- **Test early, test often, test automatically**
- **Do not reapeat yourself (DRY)**
  Code duplication causes code and maintenance overhead, and increases the opportunity for bugs.
- **Adhere to SOLID principles**
- **Refactor early and often**
- **Use RAII**
  Resource Acquisition Is Initialization. Resources are tied to object lifetime.
  Constructor Acquires, Destructor Releases (CADRe). Scope-based Resource Management (SBRM).
- **Use the standard library**
  Avoid duplication.

19.2 SOLID principles

- **Single Responsibility Principle (SRP)**
  Each design component should have a single, well-defined purpose.
- **Open-Closed Principle (OCP)**
  Component are open for extension, but closed for modification.
- **Liskov Substitution Principle (LSP)**
  Subtypes must be substitutable for their base types.
  A derived class can be used wherever a base class is expected.
- **Interface-Segregation Principle (ISP)**
  No client should be forced to depend on methods it does not use.
  Interfaces should have a single, well-defined purpose.
- **Dependency Inversion Principle (DIP)**
  High-level components should not depend on low-level components. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.
  A component should be in control of the interfaces it depends on.

Literature

"Agile Software Development. Principles, Patterns, and Practices", Robert C. Martin