C++ Code Snippets

Michele Iarossi[1]

11th May 2023 - Version 2.16 - GNU GPL v3.0

[1]`michele@mathsophy.com`

Composed in LaTeX by the author

Contents

Preface

This book provides a collection of code snippets showing the usage of common C++ language features. It is meant to be used as a quick reference or language refresher. In addition, I wanted to collect and document some particular features or constructs, like the apostrophe as a digit separator. Chances are that you are not aware of them. Explanations are kept to a minimum and are provided in the comments where necessary. The LaTeX source files and artifacts are publicly available at the following URL:

`https://github.com/MicheleIarossi/Cpp_code_snippets`

Chapter 1

Basics

In the following code snippets, the standard I/O library and namespace are always used:

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

1.1 Assertions

⇒ The first argument of a **static_assert** is a constant expression that must be true:

```
static_assert( 8<=sizeof(long),"longs␣are␣too␣small" );
```

1.2 Constants

There are two options:

⇒ **constexpr** must be known at compile time:

```
constexpr int max = 200;
constexpr int c = max + 2;
```

⇒ **constexpr** applied to functions instructs the compiler to try to evaluate the function at compile time:

```
constexpr int func(int n) { return n*2+5; }
constexpr int c = func( 122 );  // 149
```

⇒ Integer literal with single quotes for readability:

```
// 1000000
constexpr int k = 1'000'000;
```

⇒ **const** variables don't change at runtime. They cannot be declared as **constexpr** because their value is not known at compile time:

```
// the value of n is not known at compile time
const int m = n + 1;
```

⇒ Immutable class

A **const** attribute is set during construction and cannot be changed afterwards!

```
class ConstInt
{
public:
    ConstInt() : myint{0} {}
    ConstInt(int n) : myint{n} {}
    int get() const { return myint; }
private:
    const int myint;
};
```

1.3 Type Safety

⇒ Universal and uniform initialization, also known as *braced initialization*, prevents narrowing conversions from happening:

```
// safe conversions
double x{54.21};
int a{2342};

// unsafe conversions (compile error!)
int y{x};
char b{a};
```

1.4 Type Casting

The following casts are called *named conversions*:

⇒ Use **static_cast** for normal casting, i.e. types that can be converted into each other:

```
// int 15 to double 15.0
double num;
num = static_cast< double >( 15 );
```

⇒ Use **static_cast** for casting a void pointer to the desired pointer type:

```
// void * pointer can point to anything
double num;
void *p = &num;

// back to double type
double *pd = static_cast< double* >( p );
```

⇒ Use **reinterpret_cast** for casting between unrelated pointer types:

```
// reinterprets a long value as a double one
long n = 53;
double *pd = reinterpret_cast< double * >( &n );

// prints out 2.61855e-322
cout << *pd << endl;
```

⇒ Use **const_cast** for removing the **const** attribute from a reference variable pointing to a non-const variable!

```
// a non-const variable
int a_variable = 23;

// a const reference
const int& ref_constant = a_variable;

// remove the const attribute
int& not_constant = const_cast< int& >( ref_constant );

// change the non-constant variable
not_constant++;

// outputs 24 for both
cout << a_variable << endl;
cout << ref_constant << endl;
```

⇒ Use user-defined type conversions

Conversions can be implicit or require an explicit cast:

```
// User defined type
class MyType
{
public:
    MyType(int y=1) : x{y} {}
    // implicit conversions
    operator int() const { return x; }
    // requires an explicit static cast
    explicit operator double() const { return double{x}; }
private:
    int x{0};
};

MyType a{5};
MyType b{7};

// a and b are converted
// implicitely to int by operator int()
// c = 12
int c = a + b;
```

```
// b is converted to double by operator double()
// but requires explicit static cast
double =   static_cast< double >( b );
```

1.5 Type Traits

⇒ Utilities for inspecting type properties

A type trait is a template class having a single parameter. The boolean `value` member is `true` if the type verifies the property, else it is `false`:

```
#include <type_traits>

// example usage
cout << is_integral< int >::value; // true
cout << is_integral< float >::value; // false
cout << is_floating_point< double >::value; // true
```

1.6 Storage Classes

The storage class defines the memory type where an object is stored. The lifetime of an object is from the time it is first initialized until it is destroyed.

⇒ A simple class for objects:

```
class Object
{
public:
    Object(string obj_name) : name{obj_name} {
        cout << "Created␣object:␣" << obj_name << endl;
    }
    ~Object() {
        cout << "Destroyed␣object:␣" << name << endl;
    }
private:
    string name{};
};
```

⇒ Static storage with global scope, external or internal linkage

Storage is allocated before the program starts and deallocated when the program ends:

```
// static storage, external linkage
Object a{"a"};

// static storage, internal linkage
static Object b{"b"};
```

⇒ Static storage with local scope

Storage is allocated the first time the function is called and deallocated when the program ends:

```
void func(void)
{
    // static storage, local variable
    static Object c{"c"};
}

int main()
{
    // Object c is allocated
    func();
}
```

⇒ Automatic storage with local scope

Storage is allocated on the stack when the local scope is entered and deallocated after execution leaves the scope:

```
int main()
{
     // automatic storage object
    {
        Object d{"d"};
    }
}
```

⇒ Thread-local storage

Each thread is given a separate copy of the variable which is not shared with other threads:

```
// if not specified implies also static
// any thread accessing counter gets its own copy
// of the variable
thread_local int counter;
```

⇒ Dynamic storage

Storage is allocated dynamically on the heap with **new** and deallocated explicitly with **delete**:

```
int main()
{
    // dynamic storage object
    Object* e = new Object{"e"};
    delete( e );
}
```

⇒ Example of object declarations:

```
// static storage object, external linkage
Object a{"a"};

// static storage object, internal linkage
static Object b{"b"};

void func(void)
{
    cout<< "Start␣of␣func()" << endl;

    // static storage, local variable
    static Object c{"c"};

    cout<< "End␣of␣func()" << endl;
}

int main()
{
    cout<< "Start␣of␣main()" << endl;

    func();

    // local scope
    {
        cout<< "Start␣of␣local␣scope" << endl;

        // automatic storage object
        Object d{"d"};

        cout<< "End␣of␣local␣scope" << endl;
    }

    // dynamic storage object
    Object* e = new Object{"e"};
    delete( e );

    cout<< "End␣of␣main()" << endl;
    return 0;
}
```

⇒ Example result of the order of allocation and deallocation:

```
// Output printed
Created object: a
Created object: b
Start of main()
Start of func()
Created object: c
End of func()
Start of local scope
Created object: d
End of local scope
Destroyed object: d
```

```
Created object: e
Destroyed object: e
End of main()
Destroyed object: c
Destroyed object: b
Destroyed object: a
Program ended with exit code: 0
```

1.7 Integer Types

⇒ Integer types having specified widths:

```
#include <cstdint>

// signed integers
int8_t a;
int16_t b;
int32_t c;
int64_t d;

// unsigned integers
uint8_t a;
uint16_t b;
uint32_t c;
uint64_t d;
```

⇒ Secure unsigned addition:

```
uint32_t a, b, c;

if ( (UINT32_MAX - a) < b )
    // Error! Unsigned integer wrapping
    cout << "Error!␣a+b␣causes␣wrapping!\n";
else c = a + b;
```

⇒ Secure signed addition:

```
int16_t a, b, c;

if ( ( a>0 && a>(INT16_MAX-b) ) ||
     ( a<0 && a<(INT16_MIN-b) ) )
    // Error! Signed integer overflow
    cout << "Error!␣a+b␣causes␣overflow!\n";
else c = a + b;
```

⇒ Secure unsigned subtraction:

```
uint32_t a, b, c;

if ( a < b )
    // Error! Unsigned integer wrapping
    cout << "Error!␣a-b␣causes␣wrapping!\n";
else c = a - b;
```

⇒ Secure signed subtraction:

```
int8_t a, b, c;

if ( ( a>0 && a>(INT8_MAX+b) ) ||
     ( a<0 && a<(INT8_MIN+b) ) )
    // Error! Signed integer overflow
    cout << "Error!␣a-b␣causes␣overflow!\n";
else c = a - b;
```

⇒ Secure unsigned multiplication:

```
uint32_t a, b, c;

if ( a >  UINT32_MAX/b )
    // Error! Unsigned integer wrapping
    cout << "Error!␣a*b␣causes␣wrapping!\n";
else c = a*b;
```

⇒ Secure signed multiplication:

```
int16_t a, b;

int16_t c = 0;

if ( a>0 && b>0 ) {
    if ( a > INT16_MAX/b )
        // Error! Signed integer overflow
        cout << "Error!␣a*b␣causes␣overflow!\n";
    else c = a * b;
} else if ( a>0 && b<0 ) {
     if ( b < INT16_MIN/a )
        // Error! Signed integer overflow
        cout << "Error!␣a*b␣causes␣overflow!\n";
    else c = a * b;
} else if ( a<0 && b>0 ) {
     if ( a < INT16_MIN/b )
        // Error! Signed integer overflow
        cout << "Error!␣a*b␣causes␣overflow!\n";
    else c = a * b;
} else if ( a<0 && b<0 ) {
     if ( b < INT16_MAX/a )
        // Error! Signed integer overflow
        cout << "Error!␣a*b␣causes␣overflow!\n";
    else c = a * b;
}
```

⇒ Secure unsigned division:

```
uint64_t a, b, c;

if ( b==0 )
    // Error! Division by zero
    cout << "Error!␣a/b␣causes␣division␣by␣zero!\n";
else c = a/b;
```

⇒ Secure signed division:

```
uint64_t a, b, c;

if ( b==0  ||
   ( (a == INT64_MIN) && (b == -1) ) )
    // Error! Division by zero or signed integer overflow
    cout << "Error!␣a/b␣causes␣division␣by␣zero␣or␣overflow!\n";
else c = a/b;
```

1.8 Limits

⇒ Use `numeric_limits<T>` for checking against built-in type limits:

```
#include <limits>

// int type
cout << numeric_limits< int >::min(); // -2147483648
cout << numeric_limits< int >::max(); // 2147483647

// double type
cout << numeric_limits< double >::min(); // 2.22507e-308
cout << numeric_limits< double >::max(); // 1.79769e+308
cout << numeric_limits< double >::lowest(); // -1.79769e+308
cout << numeric_limits< double >::epsilon(); // 2.22045e-16
cout << numeric_limits< double >::round_error(); // 0.5
```

1.9 Operators

⇒ Unary arithmetic operators promote their operands to **int**:

```
// a variable
short x = 10;

// expression promotes to int!
+x;

// expression promotes to int!
-x;
```

⇒ Increment and decrement operators

The value of the resulting expression depends whether prefix or postfix is used:

```
int x = 5;

// prefix increment:
// expression evaluates to 6
// x evaluates to 6
++x;

// postfix increment:
// expression evaluates to 6
// x evaluates to 7
x++;

// postfix decrement:
// expression evaluates to 7
// x evaluates to 6
x--;

// prefix decrement:
// expression evaluates to 5
// x evaluates to 5
--x;
```

1.10 Namespaces and Aliases

⇒ **using** declarations for avoiding fully qualified names:

```
// use string instead of std::string
using std::string;

// use cin, cout instead of std::cin, std::cout
using std::cin;
using std::cout;
```

⇒ **using namespace** directives for including the whole namespace:

```
using namespace std;
```

⇒ An *alias* is a symbolic name that means exactly the same as what it refers to:

```
using value_type = int; // value_type means int
using pchar = char*; // pchar means char*
```

⇒ *Partial application*

Sets some number of arguments to a template:

```
// template with 2 template parameters
template<class T, class U>
class TwoObjects
{
public:
    TwoObjects() : a{}, b{} {}
    TwoObjects(T x, U y) : a{x}, b{y} {}
    T get_a() const { return a; }
    U get_b() const { return b; }
private:
    T a;
    U b;
};

// partial application which sets the first
// template parameter to char
template <class T>
using OneObject = TwoObjects< char,T >;

// usage
OneObject< float > one('b',6.7);
```

1.11 Enumerations

⇒ **enum class** defines symbolic constants in the scope of the class:

```
// enum definition
enum class Weekdays
{
    mon=1, tue, wed, thu, fri
};

// usage
Weekdays day = Weekdays::tue;
```

⇒ **int**s cannot be assigned to **enum class** and vice versa:

```
// errors!
Weekdays day = 3;
int d = Weekdays::wed;
```

⇒ A conversion function should be written which uses unchecked conversions:

```
// valid
Weekdays day = Weekdays(2);
int d = int(Weekdays::fri);
```

1.12 Volatile

⇒ **volatile** imposes restrictions on access and caching:

```
#include <csignal>

// requires volatile here
volatile int interrupted;

void sigint_handler(int signum)
{
    interrupted = 1;
}

int main(void)
{
    // install signal handler for interruption signal
    signal( SIGINT, sigint_handler );

    cout << "Waiting␣for␣interruption...\n";

    // without volatile the compiler would
    // optimize the while loop as always true
    while ( !interrupted )
    {
        // wait
    }
    cout <<  "Interrupted!\n";
}
```

Even though reordering of accesses to the volatile variable is forbidden, the compiler might still reorder it with reference to other accesses:

```
volatile int done;
char arrray[100];

void init_array()
{
    for ( size_t i=0; i<100; i++ )
        array[i] = 1;

    // the compiler might move this
    // assignment before the initialization
    // loop!
    done = 1;
}
```

Chapter 2

Functions and Lambdas

2.1 Functions

⇒ With default trailing arguments only in the function declaration:

```
// if year is omitted, then year = 2000
void set_birthday(int day, int month, int year=2000);
```

⇒ Omitting the name of an argument if not used anymore in the function definition:

```
// argument year is not used anymore in the function
// definition (doesn't break legacy code!)
void set_birthday(int day, int month, int) { ...}
```

⇒ With read-only, read-write and copy-by-value parameters:

```
// day input parameter passed by const
// reference (read-only)
// month output parameter to be changed by the
// function (read-write)
// year input parameter copied-by-value
void set_birthday(const int& day, int& month, int year);
```

⇒ Use a function for initializing an object with a complicated initializer (we might not know exactly when the object gets initialized):

```
const Object& default_value()
{
  static const Object default{1,2,3};
  return default;
}
```

⇒ Rule of thumb for passing arguments to functions:

- Pass-by-value for small objects
- Pointer parameter type if **nullptr** means no object given
- Pass-by-const-reference for large objects that are not changed
- Pass-by-reference for large objects that are changed (output parameters)
- Return error conditions of the function as return values

⇒ Function pointer type definition:

```
// pointer to a function returning a void and
// having parameters a pointer to a Fl_Widget and
// a pointer to a void
typedef void ( *Callback_type )( Fl_Widget*, void* );

// cb is a callback defined as above
Callback_type cb;
```

⇒ C-style linkage:

```
// to be put in the header file to be
// shared between C and C++
#ifdef __cplusplus
extern "C" {
#endif

// legacy C-function to be shared
void legacy_function(int p);

#ifdef __cplusplus
}
#endif
```

⇒ Uniform container for callable objects

Empty function and exception:

```
#include <function>

// empty function
function< void() > func;

// this causes an exception because func
// doesn't point to anything
try {
    func();
} catch ( const bad_function_call& e ) {
    cout << "Exception:␣" << e.what() << endl;
}
```

Assignment and initialization:

```
// example function
void print_sthg() { cout << "Hello" << endl; }

// assignment and call
func = print_sthg;

// call
func();

// works also with lambdas
function< void() > func2 { []() { cout << "Hello" << endl; }
};

// call
func2();
```

⇒ Structured binding:

```
struct Data {
    float a;
    int b;
};

// this function returns a struct
Data func(void) {
    return Data{5.4,2};
}

// assigns 85.4 to a and 2 to b
auto [a,b] = func();
```

2.2 Modifiers

Modifiers declare specific aspects of functions. This information can be used by compilers to optimize the code. There are prefix and suffix modifiers.

⇒ Prefix: **static**, **inline**, **constexpr**

```
// internal linkage
static void sum(void);

// inserting the content of the function
// in the execution path directly
inline int sum(int a, int b) { return a+b; }

// the value of the function shall be evaluated
// at compile time if possible
constexpr int sum(int a, int b) { return a+b; }
```

⇒ Suffix: `final`, `override`, **noexcept**

```

class Fruit
{
public:
    // a final method cannot be overridden anymore
    virtual int price(void) final {
       // ...
    }
};

class Apple : public Fruit
{
public:
    // override explicitly tells the compiler you are
    // overriding a virtual function
    int price(void) override {} // compiler error!{
};

// a final class doesn't allow derived classes
class Fruit final{
{
public:
    virtual int price(void) final {
       // ...
    }
};

// this function doesn't throw exceptions
int sum(int a, int b) noexcept { return a+b; }
```

2.3 Lambda Expressions

A lambda expression is an unnamed function that can be used where a function is needed as an argument or object. It is introduced by `[ ]` which are called *lambda introducers*. In the following code snippets we consider a `Function` class whose constructor requires a function as an argument:

```
// callable object returning a double
// and requiring a double parameter
typedef function< double(double) > Function_type;

// the function class
class Function : public Shape
{
public:
    // constructor
    Function (Function_type f,pair< double,double > rx,
        double d, pair< double,double > ry,
        Point p, int lx, double ar=1);
    // virtual destructor
    virtual ~Function();
    // ...
private:
    Function_type func;                // function
    // ...
}
```

Given the class above, the following lambda expressions are possible:

⇒ Without access to local variables:

```
// Instantiates a Function object where the first argument
// is an unnamed function having one double parameter x
// and returning a double. The return type is inferred
Function e_gr{[](double x){return exp(x);}, {-8.0,8.0},
        0.001, {-8.0,8.0}, {320,240}, 400};
```

⇒ With access to local variables (copy by value):

```
// Same as above, but the variable n inside the lambda
// introducer is available for the function to be used
int n = 5;
Function ee_gr{[n](double x) { n++; return expe(x,n);},
    {-8.0,8.0}, 0.001, {-8.0,8.0}, {320,240}, 400};
// now n is still 5
```

⇒ With access to local variables (copy by reference):

```
// Same as above, but the variable n inside the lambda
// introducer is available for the function to be used
// and modified
int n = 5;
Function ee_gr{[&n](double x){n++; return expe(x,n);},
    {-8.0,8.0}, 0.001, {-8.0,8.0}, {320,240}, 400};
// now n is 6!
```

⇒ With access to all local variables (default copy by value):

```
// Same as above, but all local variables
// are available for the function to be used
int n = 5;
int m = 6;
Function ee_gr{[= ](double x){n++; m++;
    return expe(x,n+m);}, {-8.0,8.0}, 0.001, {-8.0,8.0}, {320,240}, 400};
// n stays 5 and m stays 6
```

⇒ With access to all local variables (default copy by reference):

```
// Same as above, but all local variables
// are available for the function to be used and modified
int n = 5;
int m = 6;
Function ee_gr{[& ](double x){n++; m++;
    return expe(x,n+m);}, {-8.0,8.0}, 0.001, {-8.0,8.0}, {320,240}, 400};
// now n is 6 and m is 7!
```

Chapter 3

Pointers and References

3.1 Pointers

⇒ Simple object:

```
// simple pointer to double
double *d = new double{5.123};

// read
double dd = *d;

// write
*d = -11.234;

// delete the storage on the free store
delete d;

// reassign: now d points to dd
d = &dd;
```

⇒ Exception handling for `new`: bad allocation

Allocation failure throwing an exception:

```
// throws bad_alloc if allocation fails
double *d;

try {
    d = new double{5.123};
} catch ( const bad_alloc& e ) {
    cout << e.what() << '\n';
}
```

Allocation failure with `nothrow`:

```
// returns 0 if allocation fails
double *d = new(nothrow) double{5.123};

if ( d ) {
    // allocation successful
} else {
    // allocation failed
}
```

⇒ Dynamic array:

```
// dynamic array of 10 doubles
double *dd = new double[10] {0,1,2,3,4,5,6,7,8,9};

// delete the storage on the free store
delete [] dd;
```

⇒ Dynamic matrix:

```
// dynamic matrix of 5 x 5 doubles memory allocation
double **m = new double*[5];
for ( int i=0; i<5; i++ )
    m[i] = new double[5];

// memory initialization
for ( int i=0; i<5; i++ )
    for ( int j=0; j<5; j++ )
        m[i][j] = i*j;

// memory deallocation
for ( int i=0; i<5; i++ )
    delete[] m[i];
delete[] m;
```

⇒ Exception handling for `new`: invalid array length

```
// throws bad_array_new_length if allocation fails
double *d;

try {
    d = new double[-1]; // negative size
    d = new double[1] {0,1,2,3}; // too many initializers
    d = new double[INT_MAX]; // too large
} catch ( const bad_array_new_length& e ) {
    cout << e.what() << '\n';
}
```

3.2 Smart Pointers

⇒ `unique_ptr`

Holds exclusive ownership (cannot be copied!) of a dynamic object according to RAII, i.e. resource acquisition is initialization. It will automatically destroy the object if needed. Ownership can be transferred, i.e. move is supported but copy not.

```
#include <memory>

// unique pointer to an int having value 5
unique_ptr< int> p_int{ new int{5} };

// alternative declaration
auto p_int = make_unique< int >(5);

// evaluates to false
bool empty = ( p_int ) ? false : true;
```

```
// empty pointer
unique_ptr< int > p_int{};

// evaluate to true
bool empty = ( p_int ) ? false : true;
bool empty = ( p_int == nullptr );
bool empty = ( p_int.get() == nullptr );
```

Supports pointer semantics, i.e. * or ->:

```
// unique pointer to an int having value 5
unique_ptr< int > p_int{ new int{5} };

// prints 5
cout << *p_out << endl;
```

Supports swapping:

```
unique_ptr< int > p_int{ new int{5} };
unique_ptr< int > q_int{ new int{7} };

p_int.swap( q_int );

// prints 7 and 5
cout << *p_int << endl;
cout << *q_int << endl;
```

Supports reset:

```
unique_ptr< int > p_int{ new int{5} };

// destroys p_int
p_int.reset();

// evaluates to true
bool empty = ( p_int == nullptr );

unique_ptr< int > p_int{ new int{5} };

p_int.reset( new int{7} );
```

Supports replacement:

```
unique_ptr< int > p_int{ new int{5} };

// replacement
p_int.reset( new int{7} );

// prints 7
cout << *p_int << endl;
```

Supports move transferability:

```
unique_ptr< int > p_int{ new int{5} };
unique_ptr< int > q_int{ new int{7} };

// move
p_int = move( q_int );

// prints 7
cout << *p_int << endl;

// evaluates to true
bool empty = ( q_int == nullptr );
```

⇒ `shared_ptr`

Has transferable non exclusive ownership (can be copied!) of a dynamic object according to RAII, i.e. resource acquisition is initialization. It will automatically destroy the object if needed. Ownership can be transferred, i.e. move and copy are supported.

```
// shared pointer to an int having value 5
shared_ptr< int > p_int{ new int{5} };

// alternative declaration
auto p_int = make_shared< int >(5);
```

Supports copying:

```
// shared pointer to an int having value 5
shared_ptr< int > p_int{ new int{5} };

// copy
shared_ptr< int > q_int{ p_int };

// assignment
q_int = p_int;
```

Safe deallocation even when pointers are shared:

```
#include <memory>
#include <map>
#include <list>

// a simple class
class Fruit {
public:
    Fruit(string s) : name{s} {
        cout << "Fruit:␣" << name << "␣created\n";
    }
    ~Fruit() {
        cout << "Fruit:␣" << name << "␣destroyed\n";
    }
    string get_name() const { return name; }
private:
    string name;
};

// allocate objects in a map
map< string,shared_ptr< Fruit > > fruits;
try {
    fruits["banana"] = make_shared< Fruit >("banana");
    fruits["apple"]  = make_shared< Fruit >("apple");
    fruits["kiwi"]   = make_shared< Fruit >("kiwi");
    fruits["mango"]  = make_shared< Fruit >("mango");
    fruits["ananas"] = make_shared< Fruit >("ananas");
} catch ( bad_alloc& e ) {
    cout << "Allocation␣failure:␣" << e.what() << '\n';
}

// share the object pointers in a new list
list< shared_ptr< Fruit > > fruit_list;

fruit_list.push_back( fruits["kiwi"] );
fruit_list.push_back( fruits["banana"] );

// print out items
cout << "\n\nItems␣on␣the␣fruit␣list:\n";
for ( auto item : fruit_list )
    cout << "␣->" << item->get_name() << '\n';
cout << "\n\n";

// allocated objects are destroyed safely even though some
// of them are shared
```

⇒ `weak_ptr`

Has no ownership of the object pointed to. Tracks an existing object, allows conversion to a shared pointer only if the tracked object still exists. They are movable and copyable.

```
// shared pointer to an int having value 5
shared_ptr< int > p_int{ new int{5} };

// tracks the shared object
weak_ptr< int > wp_int { p_int };

// lock returns a shared pointer owning the object now
auto sh_ptr = wp_int.lock();
```

If the tracked object doesn't exist, lock returns an empty pointer:

```
// creates an empty weak pointer
weak_ptr< int > wp_int {};

{
    // shared pointer to an int having value 5
    shared_ptr< int > p_int{ new int{6} };

    // tracks the shared object
    wp_int = p_int;

    // p_int expires here
}

// attempt to own an expired object returns a null pointer
auto sh_ptr = wp_int.lock();

// evaluates to true
bool empty = ( sh_int == nullptr );
```

⇒ Optional deleter parameter:

```
// deleter function object for a pointer to an int
auto int_deleter = [](int *n) { delete n; };

// unique pointer with deleter
unique_ptr< int, decltype(int_deleter) > p_int{ new int{5}, int_deleter };
```

3.3 References

⇒ A variable reference must be initialized with a variable being referred to:

```
// an integer amount
int amount = 12;

// reference to amount
int& ref_amount = amount;

// outputs 12
cout << "amount␣=␣" << amount << endl;
cout << "ref_amount␣=␣" << ref_amount << endl;
```

⇒ A variable reference cannot be made referring to another variable at runtime:

```
// a new integer amount
int new_amount = 24;

// ref_amount still refers to amount
ref_amount = new_amount;

// outputs 24
cout << "amount␣=␣" << amount << endl;
cout << "ref_amount␣=␣" << ref_amount << endl;
```

⇒ A variable reference can be used as input and output parameter of a function:

```
// function definition taking a reference
void do_something(int& in_out_var)
{
    in_out_var *= 2;
}

int a_variable = 24;

do_something( a_variable );

// outputs 48
cout << "a_variable␣=␣" << a_variable << endl;
```

Chapter 4

Elementary Data Structures

4.1 Arrays

⇒ Declaration and initialization:

```
// array of length 4 initialized to all zeros
int array[4]{};

// array of length 4 initialized to 2,4,6,8
int array[]{2, 4, 6, 8};
```

⇒ Length of an array using **sizeof**:

```
// array of length 4 initialized to 2,4,6,8
int array[]{2, 4, 6, 8};

// works because array is not a pointer
// returns 4
size_t array_size = sizeof( array ) / sizeof( int );

// doesn't work because an array as a parameter is a pointer!
void func(int array[]) {
    // evaluates to 1!
    size_t array_size = sizeof( array ) / sizeof( int );
}
```

⇒ Range-based **for** statement:

```
// changes the values and outputs 3579
int array[]{2, 4, 6, 8};

for ( int& x : array )
    x++;
```

⇒ **auto** lets the compiler use the type of the elements in the container because it knows the type already:

```
for ( auto x : arr )
    cout << x;
```

⇒ Structured binding:

```
int v[3] = {1,2,3};

// assigns 1 to a, 2 to b, 3 to c,
auto [a,b,c] = v;
```

4.2 Vectors

Vectors as supported by the C++ standard library. Vector elements are guaranteed to be stored sequentially in memory.

⇒ Constructor with initialization list:

```
#include <vector>

// vector with base type int
vector< int > v = {2, 4, 6, 8};
```

⇒ Fill constructors

Note that since `vector` has an initializer list constructor, you need to call the fill constructor with parentheses.

```
// vector with 10 elements all initialized to 0
vector< int > v(10);

// vector with 10 elements all initialized to 2
vector< int > v(10,2);
```

⇒ Access:

```
// unchecked access to the ith element
cout << v[i];

// checked access to the ith element
cout << v.at( i );

// access to the front element
cout << v.front();

// access to the last element
cout << v.back();

// pointer to the first element of the vector
cout << *v.data << endl;
```

⇒ Add:

```
// add an element
v.push_back( 10 );
```

⇒ Insertion:

```
// iterator pointing at the beginning of the vector
auto iter = v.begin();

// now pointing to the second element
++iter;

// inserting element 10 before the second element
v.insert( iter, 10 );
```

⇒ Resize:

```
// resize to 20 elements
// new elements are initialized to 0
v.resize( 20 );
```

⇒ Loop over:

```
// range-for-loop
for ( auto x : v )
    cout << x << endl;

// auto gives to x the same type of the element on the
// right hand side of the assignment, in this case
// a vector::iterator
for ( auto x = v.begin(); x<v.end(); x++ )
    cout << *x << endl;
```

⇒ Size and capacity:

```
// size
cout << v.size();

// capacity: number of elements currently allocated
cout << v.capacity();
```

⇒ Reserve more capacity:

```
// reserve (reallocate) more capacity, e.g. at least
// 64 ints
v.reserve( 64 );
```

⇒ Throws an `out_of_range` exception if accessed out of bounds:

```
// out of bounds access
vector< int > v = {2, 4, 6, 8};

try
{
    cout << v.at( 7 );
} catch (out_of_range e)
{
    // access error!
}
```

⇒ Clearing the vector:

```
// clear the vector
v.clear();

// the vector is now empty
if ( v.empty() )
    cout << "the␣vector␣is␣empty!" << endl;
else
    cout << "the␣vector␣is␣not␣empty" << endl;
```

4.3 Strings

Strings as supported by the C++ standard library for ASCII character sets. Note that it allows almost the same methods as `vector`.

⇒ Constructors:

```
#include <string>

// initialization
string s1 = "Hello";
string s2(",␣world!");
```

⇒ Fill constructor:

```
string s3(5,'*'); // fill constructor "*****"
```

⇒ Substring constructor from a certain position and a given optional length:

```
string hello("Hello,␣world!");
string s4(hello,0,5); // "Hello"
string s5(hello,7); // "world!"
```

⇒ Constructor from C-style literal string of a certain length given an optional position:

```
string s6("Hello,␣world!",5); // "Hello"
string s7("Hello,␣world!",7,6); // "world!"
```

⇒ Concatenation:

```
// concatenation
string hello = s1 + ",␣" + s2;
```

⇒ Literal string constructor `operator""s`:

```
using namespace string_literals;

// the literal string has type string (it is not a
// C-style literal string!) and it can have zeros
string s8 = "hellohello\0\0hello"s;
```

⇒ Read a line:

```
// read a line
string line;
getline( cin,line );
```

⇒ Access to a character:

```
// access to the ith character (no illegal index checking)
s1[i];

// access to the ith character (with illegal index
// checking)
s1.at( i );
```

⇒ Add:

```
// add a single character at the end
s1.push_back( '!' ); // "Hello!"
```

⇒ Append:

```
// append
s1.append( s2 ); // "Hello, world!"

// append with an open range
string s10("Hello");
string s11(",␣world!");
s10.append( s11.begin()+7,s11.end() ); // "Hello!"
```

⇒ Remove:

```
// remove a single character from the end
s1.pop_back( 'c' ); // "Hello"
```

⇒ Clear:

```
// clear the string
s1.clear();

// print string cleared
if ( s1.empty() )
    cout << "string␣cleared" << endl;
```

⇒ Erase:

```
string s1("Hello,␣world!")

// erase from position 5 until the end
s1.erase( s1.begin()+5,s1.end() ); // "Hello

// erase from position 2 a length of 3 characters
s1.erase( 2,3 ); // "He"
```

⇒ Replace:

```
string s1("Hello!")

// replace with range
// "Hello, world!
s1.replace( s1.begin()+5,s1.end(),",␣world!" );

// replace with position and length
s1.replace( 5,8,"␣world!" ); // "Hello world!
```

⇒ Size and length:

```
// size and length
s1.size();
s1.length();
```

⇒ Substring:

```
// substring from position 6 and length 5 characters
string substring;
substring = s1.substr( 6, 5 ); // "world"
```

⇒ Find:

```
// find (returns string::npos if not found)
size_t pos;
pos = s1.find( "world" ) ;
if ( pos == string::npos )
    cerr << "Error:␣String␣not␣found!\n";

// find starting from position 5
pos = s1.find( "l", 5 ); // pos equals 9
```

⇒ C-style null-terminated string of type `const char *`:

```
// C-string
s3.c_str();
```

⇒ Conversions:

```
// from string to int, long, float
int    n = stoi( "456" );
long   n = stol( "1234567" );
double n = stod( "12.345" );

// from numeric type to string
string s = to_string( 123.456 );
```

4.4 C-Strings

⇒ Legacy strings from C:

```
#include <cstring>
#include <cstdlib>

// C-string for max 10 characters
// long string + null char '\0'
const int SIZE = 10 + 1;
char msg[SIZE] = "Hello!";
```

⇒ Checking for end of string when looping:

```
// correct looping over C-strings
int i = 0;
while ( msg[i] != '\0' && i < SIZE )
{
   // process msg[i]
}
```

⇒ Safe C-string operations: `strncpy_s`

```
// null terminated string
char src1[100] = "hello";
char dst1[6];
int r1;

// copy a string without the danger that the result will not be
// null terminated or that characters will be written past
// the end of the destination array
r1 = strncpy_s( dst1, sizeof( dst1 ), src1, sizeof( src1 ) );

if ( r1 != 0 )
   // error!
```

⇒ Safe C-string operations: `strncmp`

```
// null terminated strings
char src1[100] = "hello";
char src2[100] = "hello,␣world";

// make sure strings are null terminated!
src1[sizeof(src1)-1] = '\0';
src2[sizeof(src2)-1] = '\0';

// safe string compare, at most 12 characters are compared
strncmp( src1, src2, strlen( src2 ) );
```

⇒ Safe C-string operations: `strncat_s`

```
// null terminated strings
char s1[100] = "good";
char s5[1000] = "bye";
int r1;

// copy a string without the danger that the result will not be null
// terminated or that characters will be written past the
// end of the destination array.
r1 = strncat_s( s1, sizeof( s1 ), s5, sizeof( s5 ) );

if ( r1 != 0 )
   // error!
```

⇒ Safe C-string reading from `cin`:

```
char buffer[10];

// limits the reading to 9 characters
// leaving space for the null terminator
cin.width( 10 );

// no overflow can happen here
cin >> buffer;
```

⇒ Conversions:

```
// from C-string to int, long, float
int    n = atoi( "567" );
long   n = atol( "1234567" );
double n = atof( "12.345" );
```

Chapter 5

Input-Output

5.1 Input-Output Streams

⇒ Global input stream object `cin`, global output stream object `cout`, global error stream object `cerr`:

```
int number;
char ch;

// read a number followed by a character
// from standard input (keyboard)
// (ignores whitespaces, newlines, etc.)
cin >> number >> ch;

// write on standard output (display)
cout << number << "␣" <<  ch << endl;

// write error message on standard error (display)
cerr << "Wrong␣input!\n";
```

⇒ Boolean format manipulators

Once a manipulator is set, it stays until another one is set, i.e. manipulators are sticky:

```
#include <iomanip>

// prints true as true
cout << boolalpha << "True:␣" << true << endl;

// prints true as 1
cout << noboolalpha << "True:␣" << true << endl;
```

⇒ Integer format manipulators:

```
#include <iomanip>

// set decimal, octal, or hexadecimal notation,
// and show the base, i.e. 0 for octal and 0x for
// hexadecimal
cout << showbase;
cout << dec << 1974 << endl;
cout << oct << 1974 << endl;
cout << hex << 1974 << endl;
cout << noshowbase;

// values can be read from input in decimal, octal
// or hexadecimal format previous unsetting
// of all the flags
cin.unsetf( ios::dec );
cin.unsetf( ios::oct );
cin.unsetf( ios::hex );

//  now val can be inserted in any format
cin >> val;
```

⇒ Floating point format manipulators

Once a manipulator is set, it stays until another one is set, i.e. manipulators are sticky:

```
// set default, fixed, or scientific notation
cout << defaultfloat << 1023.984;
cout << fixed << 1023.984;
cout << scientific << 1023.984;

// set precision
cout << setprecision( 2 ) << 1023.984;

// set character text width
cout << setw( 10 );

// set left or right alignment
cout << left  << 1023.984;
cout << right << 1023.984;

// always show decimal point and zeros
cout << showpoint << 0.532;

// always show plus sign
cout << showpos << 3.64;
```

⇒ Single characters read and write:

```
// read any character from cin (doesn't skip spaces,
// newlines, etc.)
char nextChar;
cin.get( nextChar );

// loop for keeping reading
// stops when end of line control character (control-d)
// is inserted
while ( cin.get( nextChar ) )
{
    // process character
}

// write a character to cout
cout.put(nextChar)

// read a whole line of 80 chars
```

```
char line[80+1];
cin.getline( line, 81 );

// put back nextChar to cin, nextChar will be the next
// char read by cin.get()
cin.putback( nextChar );

// put back the last char got from cin.get() to cin
cin.unget();
```

⇒ If the input pattern is unexpected, it is possible to set the state of `cin` to failed:

```
try
{
    // check for unexpected input
    char ch;
    if ( cin >> ch && ch != expected_char )
    {
        // put back last character read
        cin.unget();

        // set failed bit
        cin.clear( ios_base::failbit );

        // throw an exception or deal with failed stream
        throw runtime_error( "Unexpected␣input" );
    }
}
catch ( runtime_error e )
{
    cerr << "Error!␣" << e.what() << "\n";

    // check for failure
    if ( cin.fail() )
    {
        // clear failed bit
        cin.clear();

        // read wrong input
        string wrong_input;
        cin >> wrong_input;

        cerr << "Got␣'" << wrong_input[0] << "'\n";
    }
     // End of file (eof) or corrupted state (bad)
    else return 1;
}
```

5.2 Files

⇒ Accessed by means of `ifstream` (input) or `ofstream` (output) objects:

```
#include <fstream>

// open input file
ifstream in_stream{"infile.dat"};
// open output file
ofstream out_stream{"outfile.dat"};
```

⇒ Accessed both in input and output mode by means of `fstream` objects (not recommended):

```
// open file in both input and output mode
fstream fs{"inoutfile.dat", ios_base::in | ios_base::out};
```

⇒ Opened explicitly (not recommended):

```
// input file
ifstream in_stream;
// output file
ofstream out_stream;

// open files
in_stream.open( "infile.dat" );
out_stream.open( "outfile.dat" );
```

⇒ When checking for failure, the status flag needs to be cleared in order to continue working with the file:

```
// check for failure on input file
if ( !in_stream )
{
    if ( in_stream.bad() ) error("stream␣corrupted!");

    if ( in_stream.eof() )
    {
        // no more data available
    }

    if ( in_stream.fail() )
    {
        // some format data error, e.g. expected
        // an integer but a string was read
        // recovery is still possible

        // set back the state to good
        // before attempting to read again
        in_stream.clear();

        // read again
        string wrong_input;
        in_stream >> wrong_input;
    }
}
```

⇒ As for the standard input, if the input pattern is unexpected, it is possible to set the state of the file to failed and try to recover somewhere else, e.g. by throwing an exception:

```
try
{
    // check for unexpected input
    char ch;
    if ( in_stream >> ch && ch != expected_char )
    {
        // put back last character read
        in_stream.unget();
        // set failed bit
        in_stream.clear(ios_base::failbit);

        // throw an exception or deal with failed stream
        throw runtime_error( "Unexpected␣input" );
    }
}
catch ( runtime_error e )
{
    cerr << "Error!␣" << e.what() << "\n";

    // check for failure
    if ( in_stream.fail() )
    {
        // clear failed bit
        in_stream.clear();

        // read wrong input
        string wrong_input;
        in_stream >> wrong_input;

        cerr << "Got␣'" << wrong_input[0] << "'\n";
    }
    // end-of-file or bad state
    else return 1;
}
```

⇒ Read and write:

```
// read/write data
in_stream >> data1 >> data2;
out_stream << data1 << data2;
```

⇒ Read a line:

```
string line;
getline( in_stream, line );
```

⇒ Ignore input (extract and discard):

```
// ignore up to a newline or 9999 characters
in_stream.ignore( 9999, '\n' );
```

⇒ Move the file pointer:

```
// skip 5 characters when reading (seek get)
in_stream.seekg( 5 );
// skip 8 characters when writing (seek put)
out_stream.seekp( 8 );
```

⇒ Checking for end of file:

```
// the failing read sets the EOF flag but avoids
// further processing
while ( in_stream >> next )
{
    // process next
}

// check the EOF flag
if ( in_stream.eof() )
    cout << "EOF␣reached!" << endl;
```

⇒ When a file object gets out of scope, the file is closed automatically, but explicit close is also possible (not recommended):

```
// explicititly close files
in_stream.close();
out_stream.close()
```

5.3 String Streams

A string is used as a source for an input stream or as a target for an output stream.

⇒ Input string stream: `istringstream`

```
#include <sstream>

// input string stream
istringstream data_stream{"1.234␣-5643.32"};

// read numbers from data stream
double val;
while ( is >> val )
    cout << val << endl;
```

⇒ Output string stream: `ostringstream`

```
// output string stream
ostringstream data_stream;

// the same manipulators of input-output streams
// can be used
data_stream << fixed << setprecision(2) << showpos;
data_stream << 6.432 << "␣" << -313.2134 << "\n";

// the str() method returns the string in the stream
cout << data_stream.str();
```

Chapter 6

Object-Oriented Programming

6.1 Classes

⇒ Class using dynamic arrays:

```
#include <algorithm>

class MyVector
{
public:
    // explicit constructor (avoids type conversions)
    explicit MyVector();
    // explicit constructor with size parameter
    explicit MyVector(size_t);
    // explicit constructor with initializer list
    explicit MyVector(initializer_list<double>);
    // copy constructor (pass by reference, no copying!)
    MyVector(const MyVector&);
    // move constructor
    MyVector(MyVector&&);
    // copy assignment
    MyVector& operator=(const MyVector&);
    // move assignment
    MyVector& operator=(MyVector&&);
    // virtual destructor
    virtual ~MyVector() { if ( e ) delete[] e; }
    // subscript operators
    // write
    double& operator[](size_t i) { return e[i]; }
    // read
    const double& operator[](size_t i) const { return e[i]; };
    // size
    size_t size() const { return n; }
    // capacity
    size_t capacity() const { return m; }
    // reserve
    void reserve(size_t);
    // resize
    void resize(size_t);
    // push back
    void push_back(double);
private:
    size_t n{0}; // size
    size_t m{0}; // capacity
    double *e{nullptr};
};
```

⇒ Constructors definitions

By using the **explicit** qualifier, undesired type conversions are avoided. If you give no constructor, the compiler will generate a default constructor that does nothing. If you give at least one constructor, then the compiler will generate no other constructors. Notice the use of `double()` as the default value (0.0) when initializing the vector:

```
// constructor with member initialization list
MyVector::MyVector(size_t s) : n{s}, m{s}, e{new double[n]}
{
     for ( int i=0; i<n; i++ )
          e[i] = double();
}

// constructor with initializer list parameter
MyVector::MyVector(initializer_list<double> l)
{
    n = m = l.size();
    e = new double[n];
    copy( l.begin(),l.end(),e );
}
```

⇒ Copy constructor

The argument is passed by **const** reference, i.e. no copies and no changes. If not defined, C++ automatically adds the default copy constructor. This might not be correct if dynamic variables are used, because class members are simply copied:

```
// copy constructor
MyVector::MyVector(const MyVector& v)
{
    n = v.n;
    m = v.m;
    e = new double[n];
    copy( v.e,v.e+v.n,e );
}
```

⇒ Move constructor:

```
// move constructor
MyVector::MyVector(MyVector&& v)
{
    n = v.n;
    m = v.m;
    e = v.e;
    v.n = 0;
    v.m = 0;
    v.e = nullptr;
}
```

⇒ Copy assignment

If not defined, C++ automatically adds the default assignment operator. It might not be correct if dynamic variables are used, because class members are simply copied:

```
// copy assignment
MyVector& MyVector::operator=(const MyVector& rv)
{
    // check for self assignment
    if ( this == &rv )
        return *this;
    // check if new allocation is needed
    if ( rv.n > m ) {
        if ( e )
            delete[] e;
        e = new double[rv.n];
        m = rv.n;
    }
    // copy the values
    copy( rv.e,rv.e+rv.n,e );
    n = rv.n;
    return *this;
}
```

⇒ Move assignment:

```
// move assignment
MyVector& MyVector::operator=(MyVector&& rv)
{
    delete[] e;
    n = rv.n;
    m = rv.m;
    e = rv.e;
    rv.n = 0;
    rv.m = 0;
    rv.e = nullptr;
    return *this;
}
```

⇒ Reserve (reallocation), resize and push back:

```
// reserve
void MyVector::reserve(size_t new_m)
{
    if ( new_m <= m )
        return;
    // new allocation
    double* p = new double[new_m];
    if ( e ) {
        copy( e,e+n,p );
        delete[] e;
    }
    e = p;
    m = new_m;
}

// resize
void MyVector::resize(size_t new_n)
{
    reserve( new_n );
    for ( size_t i = n; i < new_n; i++ )
        e[i] = double();
    n = new_n;
}

// push back
void MyVector::push_back(double d)
{
    if ( m == 0 )
        reserve( 8 );
    else if ( n == m )
        reserve( 2*m );
    e[n] = d;
    ++n;
}
```

⇒ Constructor invocations:

```
// constructor with size
MyVector v1(4);

// constructor with initializer list
MyVector v2{1,2,3,4};

// copy constructor
MyVector v3{v2};

// copy constructor
MyVector v3 = v2;

// pass by value with copy constructor (prefer const reference!)
void func(MyVector v4)
{
    // do something
}
```

⇒ Move invocations

Avoids copying when moving is sufficient, e.g. when returning an object from a function:

```
// example of a function returning an object
MyVector func()
{
    MyVector v4{11,12,13,14,15};
    for ( size_t i=0; i<v4.size(); i++ )
        v4[i] += i;
    return v4;
}

// move constructor
MyVector v5 = func();

// move assignment
v4 = func();
```

⇒ Compiler generated methods

If not implemented or deleted, a compiler will generate default implementations for the destructor, copy constructor, copy assignment, move constructor, move assignment (*rule of five*):

```
// basic class with default copy and move semantics
// the compiler generates the default implementation
class Basic
{
public:
    // default constructor and destructor
    Basic() = default;
    ~Basic() = default;
    // default copy constructor
    Basic(const Basic& b) = default;
    // default copy assignment
    Basic& operator=(const Basic& b) = default;
     // default move constructor
    Basic(const Basic&& b) = default;
     // default move assignment
    Basic& operator=(const Basic&& b) = default;
}

// fancy class with deleted copy and move semantics
// the compiler generates no default implementation
class Fancy
{
public:
    // no constructor and destructor
    Basic() = delete;
    ~Basic() = delete;
    // no copy constructor
    Basic(const Basic& b) = delete;
    // no copy assignment
    Basic& operator=(const Basic& b) = delete;
     // no move constructor
    Basic(const Basic&& b) = delete;
     // no move assignment
    Basic& operator=(const Basic&& b) = delete;
}
```

6.2 Operator Overloading

The behaviour is different if an operator is overloaded as a class member or friend function.

⇒ As class members

```
class Euro
{
public:
    // constructor for  euro
    Euro(int euro);
    // constructor for  euro and cents
    Euro(int euro, int cents);
    // overloaded addition operator
    Euro operator+(const Euro& amount);
private:
    int euro;
    int cents;
};
```

⇒ The definition above requires a calling object:

```
// works, equivalent to Euro{5}.operator+( Euro{2} )
Euro result = Euro{5} + 2;

// doesn't work, 2 is not a calling object of type Euro !
Euro result = 2 + Euro{5};
```

⇒ As friend members

```
#include <istream>
#include <ostream>

class Euro
{
public:
    // constructor for  euro
    Euro(int euro);
    // constructor for euro and cents
    Euro(int euro, int cents);
    // overloaded addition operator
    friend Euro operator+(const Euro& amount1, const Euro& amount2);
    // insertion and extraction operators
    friend ostream& operator<<(ostream& outs, const Euro& amount);
    friend istream& operator>>(istream& ins, Euro& amount);
private:
    int euro;
    int cents;
};

// definition of the addition operator
Euro operator+(const Euro& amount1, const Euro& amount2)
{
    return Euro( amount1.euro + amount2.euro, amount1.cents + amount2.cents );
}

// insertion operator
ostream& operator<<(ostream& outs, const Euro& amount)
{
    return (outs << amount.euro << "␣Eur␣" << amount.cents << "␣cents");
}
```

⇒ The definition above works for every combination because **int** arguments are converted by the constructor to `Euro` objects:

```
// works, equivalent to Euro{5} + Euro{2}
Euro result = Euro{5} + 2;

// works, equivalent to Euro{2} + Euro{5}
Euro result = 2 + Euro{5};
```

6.3 Inheritance

⇒ Abstract base class (excerpt):

```
class Shape : public Widget
{
public:
    // no copy constructor allowed
    Shape(const Shape&) = delete;
    // no copy assignment allowed
    Shape& operator=(const Shape&) = delete;
    // virtual destructor
    virtual ~Shape() {}
    // overrides the pure virtual function Fl_Widget::draw()
    void draw();
    // moves a shape relative to the current
    // top-left corner (call of redraw() might be needed)
    void move(int dx, int dy);
    // setter and getter methods for color, style, font, transparency
    // (call of redraw() might be needed)
    void set_color(Color_type c);
    void set_color(int c);
    Color_type get_color() const { return to_color_type(new_color); }
    void set_style(Style_type s, int w);
    Style_type get_style() const { return to_style_type(line_style); }
    void set_font(Font_type f, int s);
protected:
    // Shape is an abstract class, no instances of Shape can be created!
    Shape() : Widget() {}
    // protected pure virtual methods to be overridden by derived classes
    virtual void draw_shape() = 0;
    virtual void move_shape(int dx, int dy) = 0;
    // protected setter methods
    virtual void set_color_shape(Color_type c) {
        new_color = to_fl_color( c );
    }
    virtual void set_color_shape(int c) {
        new_color = to_fl_color( c );
    }
    virtual void set_style_shape(Style_type s, int w);
    virtual void set_font_shape(Font_type f, int s);
    // helper methods for FLTK style and font
    void set_fl_style();
    void restore_fl_style();
    void set_fl_font();
    void restore_fl_font() { fl_font(old_font,old_fontsize); }
    // test method for checking resize calls
    void draw_outline();
private:
    Fl_Color new_color{Fl_Color()};   // color
    Fl_Color old_color{Fl_Color()};   // old color
    Fl_Font new_font{0};              // font
    Fl_Font old_font{0};              // old font
    Fl_Fontsize new_fontsize{0};      // font size
    Fl_Fontsize old_fontsize{0};      // old font size
    int line_style{0};                // line style
    int line_width{0};                // line width
};
```

⇒ A base class can be a derived class itself:

```
// Shape is a base class for Line but Shape is derived from Widget
class Line : public Shape
{
    ...
};
```

⇒ Disabling copy constructors and assignment

Notice the `= delete` syntax for disabling them. If they were allowed, slicing might occur when derived objects are copied into base objects. Usually, **sizeof**`(Shape) <=` **sizeof**`(derived classes from Shape)`. By allowing copying, some attributes are not be copied, which might lead to crashes when member functions of the derived classes are called! Note that slicing is the class object equivalent of integer truncation.

```
class Shape : public Widget
{
public:
    // no copy constructor allowed
    Shape(const Shape&) = delete;
    // no copy assignment allowed
    Shape& operator=(const Shape&) = delete;
    ...
};
```

⇒ Virtual destructor

Destructors should be declared **virtual**. When derived objects are referenced by base class pointers, the destructor of the derived class is called if it is declared **virtual**.

```
class Shape : public Widget
{
public:
    ...
    // virtual destructor
    virtual ~Shape() {}
    ...
};
```

⇒ Protected constructor

By declaring the constructor as **protected**, no instances of this class can be created by a user. Since `Shape` is an abstract class, it should be used only as a base class for derived classes.

```
class Shape : public Widget
{
    ...
protected:
    ...
    // Shape is an abstract class
    // no instances of Shape can be created!
    Shape() : Widget() {}
    ...
};
```

⇒ Protected member functions

By declaring member functions as protected, access is restricted only to the class itself or to derived classes, a user cannot call such functions. This is useful for helper functions which are not supposed to be called directly outside the class.

```
class Shape : public Widget
{
    ...
protected:
    ...
    // helper methods for FLTK style and font
    void set_fl_style();
    void restore_fl_style();
    void set_fl_font();
    void restore_fl_font() {
        fl_font( old_font,old_fontsize );
    }
    ...
};
```

⇒ Pure virtual functions

The protected member functions `draw_shape()` and `move_shape()` are pure virtual functions, i.e. a derived class must provide an implementation for them. Notice the syntax which signals that the function is a pure virtual function. When a class has function members that are declared as pure virtual functions, then the class becomes an abstract class.

```
class Shape : Widget
{
    ...
protected:
    ...
    // protected virtual methods to be overridden by
    // derived classes
    virtual void draw_shape() = 0;
    virtual void move_shape(int dx, int dy) = 0;
    ...
};
```

⇒ Virtual functions

The protected member functions `set_color_shape()` is declared as a virtual function and an implementation is provided. This means that if a derived class does not override the implementation of the base class, the derived class inherits the implementation from the base class.

```
class Shape : Widget
{
    ...
protected:
    ...
    // protected setter methods
    virtual void set_color_shape(Color_type c) {
        new_color = to_fl_color( c );
    }
    virtual void set_color_shape(int c) {
        new_color = to_fl_color( c );
    }
    ...
};
```

⇒ A derived class from the base class `Shape`:

```
class Line : public Shape
{
public:
    Line(pair< Point,Point > line) : l{line} {
        resize_shape( l.first,l.second );
    }
    virtual ~Line() {}
    pair< Point,Point > get_line() const { return l; }
    void set_line(pair<Point,Point> line) { l = line; }
protected:
    void draw_shape() {
        fl_line( l.first.x, l.first.y, l.second.x, l.second.y );
    }
    void move_shape(int dx, int dy) {
        l.first.x  += dx;  l.first.y += dy;
        l.second.x += dx; l.second.y += dy;
        resize_shape( l.first, l.second );
    }
private:
    pair< Point,Point > l;
};
```

⇒ `Line` is derived from `Shape`, it models the relationship that a `Line` is a `Shape`

```
class Line : public Shape
{
    ...
};
```

⇒ `Line` has its own getter and setter functions for accessing its own internal private representation:

```
class Line : public Shape
{
public:
    ...
    pair< Point,Point > get_line() const { return l; }
    void set_line(pair< Point,Point > line) { l = line; }
    ...
private:
    pair<Point,Point> l;
};
```

⇒ `Line` specialises the virtual functions `draw_shape()` and `move_shape()` according to its representation:

```
class Line : public Shape
{
public:
    ...
protected:
    void draw_shape() {
        fl_line( l.first.x, l.first.y, l.second.x, l.second.y );
    }
    void move_shape(int dx, int dy) {
        l.first.x  += dx;  l.first.y += dy;
        l.second.x += dx; l.second.y += dy;
        resize_shape( l.first, l.second );
    }
    ...
};
```

⇒ `Circle` is also derived from `Shape`, a `Circle` is also a `Shape`.

```
class Circle : public Shape
{
public:
    Circle(Point a, int rr) : c{a}, r{rr} {
        resize_shape( Point{c.x-r,c.y-r}, Point{c.x+r,c.y+r} );
    }
    virtual ~Circle() {}
    Point get_center() const { return c; }
    void set_center(Point p) {
        c = p;
        resize_shape( Point{c.x-r,c.y-r}, Point{c.x+r,c.y+r}) ;
    }
    int get_radius() const { return r; }
    void set_radius(int rr) {
        r = rr;
        resize_shape( Point{c.x-r,c.y-r}, Point{c.x+r,c.y+r} );
    }
protected:
    void draw_shape() {
        Point tl = get_tl();
        Point br = get_br();
        fl_arc( tl.x, tl.y, br.x-tl.x, br.y-tl.y, 0, 360 );
    }
    void move_shape(int dx,int dy) {
        c.x += dx; c.y += dy;
        resize_shape( Point{c.x-r,c.y-r}, Point{c.x+r,c.y+r} );
    }
private:
    Point c{}; // center
    int r{0};  // radius
};
```

6.4 Polymorphism

⇒ From a window perspective, it is possible to attach and draw any type of widget, and the window just needs to call the `Fl_Widget::draw()` method:

```
void Window::draw(Fl_Widget& w) {
    w.draw();
}
```

⇒ Since `Fl_Widget::draw()` is a pure virtual function, it is overridden by `Shape::draw()`, which in turn calls the pure virtual function `Shape::draw_shape()`, which gets specialised in every derived class, e.g. as in `Line` or `Circle`:

```
void Shape::draw() {
    set_fl_style();
    if ( is_visible() )
        draw_shape();
    restore_fl_style();
}

void Circle:: draw_shape() {
    Point tl = get_tl();
    Point br = get_br();
    fl_arc( tl.x, tl.y, br.x-tl.x, br.y-tl.y, 0, 360 );
}

void Line::draw_shape() {
    fl_line( l.first.x, l.first.y, l.second.x, l.second.y );
}
```

⇒ Polymorphism is allowed by the **virtual** keyword which guarantees late binding: the call `w.draw()` inside `Windows::draw()` binds to the `draw_shape()` function of the actual object referenced, either to a `Line` or `Circle` instance.

```
Window win;
Line diagonal { {Point{200,200},Point{250,250}} };
Circle c1{Point{100,200},50};

win.draw( diagonal ); // calls Line::draw_shape()
win.draw( c1 ); // calls Circle::draw_shape()
```

Chapter 7

Advanced Topics

7.1 Exceptions

⇒ The value thrown by **throw** can be of any type:

```
// exception class
class MyException
{
public:
    MyException(string s);
    virtual ~MyException();
    friend ostream& operator<<(ostream& os, const MyException& e);
protected:
    string msg;
};

try
{
    throw MyException("error");
}
catch ( MyException& e )
{
    // error stream
    cerr << e;
}
// everything else
catch (...)
{
    exit(1);
}
```

⇒ The standard library defines a hierarchy of exceptions. For example `runtime_error` can be thrown when runtime errors occur:

```
try
{
    throw runtime_error("unexpected␣result!");
}
catch ( runtime_error& e )
{
    // error stream
    cerr << "runtime␣error:␣" << e.what() << "\n";
    return 1;
}
```

⇒ Functions throwing exceptions should list the exceptions thrown in the exception specification list. These exceptions are not caught by the function itself!

```
// exceptions of type DivideByZero or OtherException are
// to be caught outside the function. All other exceptions
// end the program if not caught inside the function.
void my_function( ) throw (DivideByZero, OtherException);

// empty exception list, i.e. all exceptions end the
// program if thrown but not caught inside the function.
void my_function( ) throw ( );

// all exceptions of all types treated normally.
void my_function( );
```

⇒ *Basic guarantee*

Any part of your code should either succeed or throw an exception without leaking any resource:

```
// Does local cleanup avoiding leaking of resources
// if exception occurs
void my_function(void)
{
    void *p;
    socket *s;

    try
    {
        /* code that acquires some resource (memory,
           socket, etc.) and might throw an exception */
    }
    catch (...)
    {
        // local cleanup here
        delete p;      // free memory
        s.release();  // release socket
        // re-throw because function didn't succeed
        throw()
    }
}
```

7.2 Templates

Types are used as parameters for a function or a class. C++ does not need the template declaration. Always put the template definition in the header file directly!

⇒ Function template:

```
// generic swap function
template<class T>
void generic_swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

int a, b;
char c, d;

// swaps two ints
generic_swap< int >(a, b);

// swaps two chars
generic_swap< char >(c, d);
```

⇒ Template type deduction

The compiler infers the template parameter from the usage:

```
double e, f;

// swaps two doubles
// compiler infers the template parameter from usage
generic_swap( a, b );
```

⇒ Constrain template types with assertions and type traits:

```
#include <type_traits>

template<class T>
void generic_swap(T& a, T& b)
{
    static_assert( std::is_copy_constructable<T>(),
        "Type␣must␣be␣copy␣constructable" );
    static_assert( std::is_assignable<T&,T>(),"Type␣must␣allow␣T&␣=␣T" );

    T temp = a;

    a = b;
    b = temp;
}
```

⇒ Class templates

Extending `MyVector` with templates. Class templates are also called *type generators*:

```
template<class T>
class MyVector
{
public:
    // constructor
    explicit MyVector();
    // constructor with size
    explicit MyVector(size_t);
    // constructor with initializer list
    explicit MyVector(initializer_list<T>);
    // copy constructor (pass by reference, no copying!)
    MyVector(const MyVector&);
    // move constructor
    MyVector(MyVector&&);
    // copy assignment
    MyVector& operator=(const MyVector&);
    // move assignment
    MyVector& operator=(MyVector&&);
    // virtual destructor
    virtual ~MyVector() { if (e) delete[] e; }
    // subscript operators
    // write
    T& operator[](size_t i) { return e[i]; }
    // read
    const T& operator[](size_t i) const { return e[i]; };
    // size
    size_t size() const { return n; }
    // capacity
    size_t capacity() const { return m; }
    // reserve
    void reserve(size_t);
    // resize
    void resize(size_t);
    // push back
    void push_back(T);
private:
    size_t n{0}; // size
    size_t m{0}; // capacity
    T *e{nullptr};
};
```

⇒ Method definition with templates:

```
// copy assignment
template<class T>
MyVector<T>& MyVector<T>::operator=(const MyVector<T>& rv)
{
    // check for self assignment
    if ( this == &rv )
        return *this;
    // check if new allocation is needed
    if ( rv.n > m )
    {
        if ( e )
            delete[] e;
        e = new T[rv.n];
        m = rv.n;
    }
    // copy the values
    copy( rv.e,rv.e+rv.n,e );
    n = rv.n;
    return *this;
}
```

⇒ Specialisation or template instantiation:

```
// MyVector of double
MyVector< double > v4{11,12,13,14,15};

// function returning a MyVector of double
MyVector< double > func()
```

```
{
    MyVector< double > v4{11,12,13,14,15};
    for ( size_t i=0; i<v4.size(); i++ )
        v4[i] += i;
    return v4;
}
```

⇒ Non-type template parameters

```
// Wrapper class for an array
template<class T,size_t N>
class Wrapper
{
public:
    Wrapper() { for( T& e : v )  e=T(); }
    ~Wrapper() {}
    T& operator[](int n) { return v[n]; };
    const T& operator[](int n) const { return v[n]; };
    size_t size() const { return N; }
private:
    T v[N];
};

// usage
Wrapper< double,5 > array;
Wrapper< char,3 > array;
```

⇒ Allocator as a class template parameter

```
// Usage of an allocator as a class template parameter
// Generalises MyVector for data types without a default
// constructor and with customised memory management
template<class T, class A=allocator<T>>
class MyVector
{
public:
    // constructor
    explicit MyVector();
    // constructor with size and default value
    explicit MyVector(size_t,T def = T());
    // constructor with initializer list
    explicit MyVector(initializer_list<T>);
    // copy constructor (pass by reference, no copying!)
    MyVector(const MyVector&);
    // move constructor
    MyVector(MyVector&&);
    // copy assignment
    MyVector& operator=(const MyVector&);
    // move assignment
    MyVector& operator=(MyVector&&);
    // virtual destructor
    virtual ~MyVector();
    // subscript operators
    // write
    T& operator[](size_t i) { return e[i]; }
    // read
    const T& operator[](size_t i) const { return e[i]; };
    // size
    size_t size() const { return n; }
    // capacity
    size_t capacity() const { return m; }
    // reserve
    void reserve(size_t);
    // resize
    void resize(size_t,T def = T());
    // push back
    void push_back(T);
private:
    A alloc;
    size_t n{0}; // size
    size_t m{0}; // capacity
    T *e{nullptr};
};

// reserve
template<class T,class A>
void MyVector<T,A>::reserve(size_t new_m)
{
    if ( new_m <= m )
        return;
    // new allocation
    T* p = alloc.allocate( new_m );
    if ( e ) {
        // copy
        for ( size_t i=0; i<n; ++i )
            alloc.construct( &p[i], e[i] );
        // destroy
        for ( size_t i=0; i<n; ++i )
            alloc.destroy( &e[i] );
        // deallocate
        alloc.deallocate( e, m );
    }
    e = p;
    m = new_m;
}
```

⇒ Template friend operator:

```
// Note the declaration of the template friend operator.
template<class T>
class SimpleNode
{
    // constructor with size of the list
    SimpleNode(int size);
    // destructor
    ~SimpleNode();
    // copy constructor
    SimpleNode(ListNode<T>& b);
    // assignment operator
    SimpleNode<T>& operator=(const SimpleNode<T>& b);
    // friend insertion operator
    template <class TT>
    friend ostream& operator<<(ostream& outs, const SimpleNode<TT>& rhs);
private:
    T *p;
    int size;
}
```

7.3 Iterators

⇒ An iterator is a generalisation of a pointer. It is an object that identifies an element of a sequence. Different containers have different iterators.

```
#include <vector>

vector< int > v = {1,2,3,4,5};
// mutable iterator
vector< int >::iterator e;

// bidirectional access
e = v.begin();
++e;
// print v[1]
cout << *e << endl;
--e;
// print v[0]
cout << *e << endl;

// random access
e = v.begin();
// print v[3]
cout << e[3] << endl;

// change an element
e[3] = 9;
```

⇒ Constant iterator

```
// constant iterator (only read)
vector< int >::const_iterator c;

// print out the vector content (read only)
// end() points one element beyond the last one!
for ( c = v.begin(); c != v.end(); c++ )
    cout << *c << endl;

// not allowed
// c[2] = 2;
```

⇒ Reverse iterator

```
// reverse iterator
vector< int >::reverse_iterator r;

// print out the vector content in reverse order
for ( r = v.rbegin(); r != v.rend(); r++ )
    cout << *r << endl;
```

⇒ Example iterator class for a power range:

```
#include <cstdint>

// power range class
class PowerRange
{
public:
    PowerRange(uint32_t m) : max{m} {
        for ( auto n=0; n<max; n++ )
            v.push_back( n*n );
    }
    ~PowerRange() {}
    class iterator;
    iterator begin();
    iterator end();
private:
    uint32_t max{0};
    vector< uint32_t > v;
};

// iterator class
class PowerRange::iterator
{
public:
    iterator(uint32_t *p) : curr{p} { }
    // postfix
    iterator operator++(int) {
        ++curr;
        return iterator{curr-1};
    }
    // prefix
    iterator& operator++() {
        ++curr;
        return *this;
    }
    uint32_t operator*() { return *curr; }
    bool operator!=(const iterator& e) {
        return curr != e.curr;
    }
    bool operator==(const iterator& e) {
        return curr == e.curr;
    }
private:
    uint32_t *curr{nullptr};
};

// returns the first element
PowerRange::iterator PowerRange::begin()
{
    return PowerRange::iterator(&v[0]);
}

// returns one element beyond the end
PowerRange::iterator PowerRange::end()
{
    return PowerRange::iterator{&v[max]};
}

// example usage
PowerRange r{10};

// normal for loop
for ( auto x = r.begin(); x != r.end(); x++ )
    cout << *x << endl;

// range-based for loop
```

```
for ( auto x:r )
    cout << x << endl;
```

⇒ A linked list class:

```
#include <ostream>
#include <algorithm>

// node of the linked list
template <class T>
class LListNode
{
public:
    // constructor for a new node
    LListNode(T new_data = T(), LListNode<T>* new_next = nullptr) :
        data(new_data), next(new_next) {};
    // friends
    friend class LList<T>;
    template <class TT>
    friend ostream& operator<<(ostream& outs, const LList<TT>& rhs);
private:
    // data element
    T data{T()};
    // next pointer
    LListNode<T>* next{nullptr};
};

// linked list declaration
template <class T>
class LList
{
public:
    // default constructor
    LList() : head(nullptr) {};
    // copy constructor
    LList(const LList<T>& rhs) { *this = rhs; };
    // assignment operator
    LList<T>& operator=(const LList<T>& rhs);
    // virtual destructor
    virtual ~LList() { clear(); };
    // clear (free) the list
    void clear();
    // get head
    LListNode<T>* get_head() const { return head; };
    // get node
    LListNode<T>* get_node(int n=0) const;
    // insert a new data element at the head of the list
    void insert_at_head(T new_data);
    // insert a new data element at the end of the list
    void insert_at_end(T new_data);
    // insert a new element at a given pointed node
    void insert_at_point(LListNode<T>* ptr, T new_data);
    // remove the data element at the head of the list
    T remove_head();
    // test for empty list
    bool is_empty() const { return head == nullptr; };
    // count of the elements stored in the list
    int size() const;
    // insertion operator
    template <class TT>
    friend ostream& operator<<(ostream& outs, const LList<TT>& rhs);
    // iterator type
    class iterator;
    // iterator to first element
    iterator begin() { return iterator(head); }
    // iterator to one beyond last element
    iterator end() { return iterator(nullptr); }
private:
    // head pointer
    LListNode<T>* head{nullptr};
    // recursive copy list function
    LListNode<T>* recursive_copy(LListNode<T>* rhs);
};
```

⇒ Example iterator class for the custom linked list class above

```
// iterator class for the linked list
template <class T>
class LList<T>::iterator
{
public:
    iterator(LListNode<T>* p) : curr{p} {}
    // prefix increment, returns a reference!
    iterator& operator++() {
        curr = curr->next;
        return *this;
    }
    T& operator*() const {
        return curr->data;
    }
    bool operator==(const iterator& b) const {
        return curr == b.curr;
    }
    bool operator!=(const iterator& b) const {
        return curr != b.curr;
    }
private:
    LListNode<T>* curr{nullptr};
};

// example usage
LList< int > data_list;

// inserts element into the list
data_list.insert_at_head( 45 );
data_list.insert_at_head( -21 );
data_list.insert_at_end( 127 );

// prints data_list = (-21) -> (45) -> (127)
cout << "data_list␣=␣" << data_list << endl;
// prints  data_list.size() = 3
cout << "data_list.size()␣=␣" << data_list.size();

// applies standard algorithms on the custom linked list
LList< int >::iterator p = find(data_list.begin(), data_list.end(),45);

// checks if the element has been found
// standard algorithms return the end of a sequence,
// i.e. the end iterator, to indicate failure
if ( p != second_list.end() )
    cout << "found␣element␣" << *p << "\n\n";
else
    cout << "cannot␣find␣element␣" << 45 << "\n\n";

// write access
*p = 180;

// prints data_list = (-21) -> (180) -> (127)
cout << "data_list␣=␣" << data_list << endl;
```

7.4 Containers

⇒ Sequential containers: `list`

```
#include <list>

list< double > data = {1.32,-2.45,5.65,-8.93,2.76};

// adds elements
data.push_back( 9.23 );
data.push_front( -3.94 );

// bidirectional iterator, no random access
list< double >::iterator e;

// advance
e = data.begin();
advance( e, 2);

// erases element 5.65
data.erase( e );

// print out the content
for ( e = data.begin(); e != data.end(); e++ )
    cout << *e << endl;

// range-for-loop
for ( auto x : data )
    cout << x << endl;
```

⇒ Container adapters: `stack`

```
#include <stack>

stack< double > numbers;

// push on the stack
numbers.push( 5.65 );
numbers.push( -3.95 );
numbers.push( 6.95 );

// size
cout << numbers.size()

// read top data element
```

```
double d = numbers.top();

// pop top element
numbers.pop();
```

⇒ Associative containers: `set`, ordered according to its unique keys

```
#include <set>

set< char > letters;

// inserting elements
letters.insert( 'a' );
letters.insert( 'd' );

// no duplicates!
letters.insert( 'd' );
letters.insert( 'g' );

// erase
letters.erase( 'a' );

// const iterator
set< char >::const_iterator c;
for ( c = letters.begin(); c != letters.end(); c++ )
    cout << *c << endl;
```

⇒ Associative containers: `map`, ordered according to its key in pairs (key,value)

```
#include <map>
#include <utility>
#include <string>

// initialization
map< string,int > dict = { {"one",1}, {"two",2} };
pair< string,int > three("three",3);

// insertion
dict.insert( three );
dict["four"] = 4;
dict["five"] = 5;

// make_pair
pair< string,int > six;
six = make_pair< string,int >( "six",6 );
dict.insert( six );

// iterator
map< string,int >::iterator two;

// find
two = dict.find( "two" );

// erase
dict.erase( two );

// range-for-loop
// inside the loop n is a pair
// the key is n.first and the value is n.second
for ( auto n : dict ) {
    cout << "(" <<  n.first << "," <<  n.second << ")";
    cout << endl;
}
```

⇒ Associative containers: `multimap`, ordered according to its key in pairs (key,value), keys can be repeated

```
#include <map>
#include <utility>
#include <string>

multimap< string,int > mm;

// insert
mm.insert( make_pair< string,int >( "Mary",1) );
mm.insert( make_pair< string,int >( "Dick",6) );
mm.insert( make_pair< string,int >( "Mary",7) );
mm.insert( make_pair< string,int >( "John",1) );
mm.insert( make_pair< string,int >( "Mary",4) );

// search for Mary, returns a pair of iterators
auto pp = mm.equal_range( "Mary" );

// prints out Mary :  1, Mary :  7, Mary :  4
for ( auto p = pp.first; p !=pp.second; ++p )
    cout << p->first << "␣:␣" << p->second << ",␣";
```

7.5 Algorithms

Provided by the C++ standard template library (STL).

⇒ `find`

```
#include <algorithm>
#include <vector>

vector< int > v = {6,2,7,13,4,3,1};
vector< int >::iterator p;

// find
// points to the first occurrence of 13 in v
p = find( v.begin(),v.end(),13 );
```

⇒ `find_if`

```
bool test_greater_than_5(int x) { return x>5; }
// find_if
// general search, stops as soon as the predicate is
// satisfied points to the first occurrence of an
// element greater than 5 in v
p = find_if( v.begin(),v.end(),test_greater_than_5 );
```

⇒ Function object

```
// function object
class TestGreater
{
public:
    TestGreater(int x) : n{x} {}
    bool operator() (const int x) const { return x>n; }
private:
    int n{0};
};

// find_if
// general search, stops as soon as the predicate is
// satisfied points to the first occurrence of an
// element greater than 7 in v
p = find_if( v.begin(),v.end(),TestGreater(7) );
```

⇒ accumulate

```
#include <algorithm>
#include <vector>
#include <list>
#include <set>
#include <numeric>
#include <functional>

// accumulate
// adds the values from a sequence
// the last parameter is the initial value
// the return type is the type of the initial value!
long res = accumulate( v.begin(),v.end(),long(0) );

// generic accumulate performing multiplication instead
// of sum passes a function object
// multiplies<int>() defined in <functional>
double res = accumulate( v.begin(),v.end(), 1.0,multiplies<int>() );
```

⇒ Inner product

```
// inner product
vector< int > v1 = {-2,2,4,11,-4,3,1};
vector< int > v2 = {4,2,4,18,5,3,1,7,9,};

long res = inner_product( v1.begin(), v1.end(), v2.begin(),long(0) );
```

⇒ copy between different containers

```
list< double > data = {1.32,-2.45,5.65};
vector< int> int_data(5);

// copy data from the list of doubles to the vector of integers
copy( data.begin(),data.end(),int_data.begin() );
```

⇒ copy between a container and an output stream. A container can be initialized by the elements retrieved via a pair of iterators.

```
// from the input character stream cin elements are
// read as strings and used to populate a set of words

// a set doesn't allow any duplicates and keeps
// elements ordered
set< string > words{ istream_iterator< string >{cin},
                             istream_iterator< string >{} };

// copy the words from the set to the output stream cout
// and add a new line after each word
copy( words.begin(),words.end(), ostream_iterator< string >{cout,"\n"} );
```

⇒ Merge sort

```
// merge sort
sort( v.begin(),v.end() );
```

⇒ Binary search

```
// binary search
bool found;
found = binary_search( v.begin(), v.end(), 3 );
```

⇒ `reverse`

```
// reverse
reverse( v.begin(),v.end() );
```

7.6 Utilities

⇒ Random numbers

C-style:

```
#include <cstdlib>
#include <ctime>

// seed the generator
srand( time(0) );

// integer random number between 0 and RAND_MAX
int n = rand();
```

C++ style:

```
#include <random>
#include <functional>

// bind constructs a function object that calls its
// first argument with its second argument

// normal distribution by using the default random engine
auto gen_def = bind( normal_distribution<double>{15,4.0},
                    default_random_engine{} );
```

```
// normal distribution by using the Mersenne Twister
// engine with seed 91586
auto gen_mt = bind( normal_distribution<double>{15,4.0},
                    mt19937_64{91586} );

// call the function objects to get random numbers
cout << gen_def() << gen_mt() << endl;
```

⇒ `bitset`

Bits and bit operations:

```
#include <bitset>

bitset<8> a{87};   // 01010111
bitset<8> b{0x87}; // 10000111

cout << a << endl << b << endl;

// boolean and
bitset<8> c = a & b;

cout << c << endl; // 00000111
```

Access to single bits:

```
// prints out single bits reverting the order
// 11100000
for ( auto i=0; i<8; ++i )
     cout << c[i];
```

⇒ Chrono

Run-time measurement in milliseconds:

```
#include <chrono>

using namespace std::chrono;

// returns a value of type time_point<system_clock>
auto t1 = system_clock::now();

// ...

// returns a value of type time_point<system_clock>
auto t2 = system_clock::now();

// run-time in milliseconds
cout << duration_cast< milliseconds >( t2-t1 ).count();
```

Sleep for a certain amount of time:

```
#include <thread>
#include <chrono>

using namespace std::chrono;
using namespace std::literals::chrono_literals;
```

```
// returns a value of type time_point<system_clock>
auto start = system_clock::now();

// pause thread for 100 ms
this_thread::sleep_for( 100ms );

// returns a value of type time_point<system_clock>
auto end = system_clock::now();

// sleep time in milliseconds
cout << duration_cast< milliseconds >(end-start).count();
```

Example stopwatch implementation for timing code execution in a given scope:

```
#include <chrono>

class Stopwatch {
public:
    Stopwatch(nanoseconds& total_time) :
        result { total_time },
        start {high_resolution_clock::now()} {}
    ~Stopwatch() {
        result = duration_cast<nanoseconds>(
                     high_resolution_clock::now() - start);
     }
private:
    nanoseconds& result;
    const time_point< high_resolution_clock > start;
};

// code portion to be measured
nanoseconds total_time{0};
{
    Stopwatch elapsed{ total_time };
    this_thread::sleep_for( 100ms );
}

// number of nanosecond ticks
cout << "total_time.count()␣=␣" << total_time.count();
```

⇒ pair

Class template containing 2 objects of different types:

```
#include <utilities>

// temperature pairs
pair< float,string > temp1{ 23.4, "␣degrees␣C" }{

pair< float,string > temp2;
temp2 = make_pair< float,string >( 21.7,"␣degrees␣C" );

// prints temperatures
cout << temp1.first << temp1.second << endl;
cout << temp2.first << temp2.second << endl;
```

⇒ Regular expressions

First match:

```
#include <regex>
#include <string>

// regular expression to look for
// in raw string format
regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" };
smatch results;
string zip("The␣string␣NJ07936-3173␣and␣NJ07936-3175␣are␣ZIP␣codes");

// first match
bool found = regex_search( zip,results,regex );

// this prints position 11
if ( found ) {
    cout << "First␣match␣at␣position:␣";
    cout << results.position(0) << endl;
}
```

Iterative search:

```
// regular expression to look for
// in raw string format
regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" };
smatch results;
string zip("The␣string␣NJ07936-3173␣and␣NJ07936-3175␣are␣ZIP␣codes");

// start from the beginning
string::const_iterator start( zip.cbegin() );

while ( regex_search( start, zip.cend(), results, regex ) ) {
     // match has been found
     cout << ( start == zip.cbegin() ? "" : "␣" );
     cout << results[0];

     // continue searching from the suffix sequence
     // after the match
     start = results.suffix().first;
}
cout << endl;
```

Replace:

```
#include <regex>
#include <string>

// regular expression to look for
// in raw string format
regex regex{ R"((\w{2})(\d{5})(-\d{4})?)" };
string zip("The␣string␣NJ07936-3173␣and␣NJ07936-3175␣are␣ZIP␣codes");

// replaces the ZIP codes with XXX
string zip_hidden = regex_replace( zip,regex,"XXX" );
cout << zip_hidden << endl;
```

⇒ Filesystem

Declaring a `path` object:

```
#include <filesystem>

using namespace std::filesystem;

// path object
path current_dir;
```

Retrieving the current folder:

```
// stores the program current directory
current_dir = current_path();
```

Declaring a directory iterator:

```
// directory iterator
directory_iterator dir{current_dir};
```

Loop over directory elements:

```
// loop over the current directory entries
for (  auto& dir_entry : dir ) {
    if ( dir_entry.is_directory() ) {
        cout << "Found␣directory:␣";
        cout << dir_entry.path().filename() << endl;
    }
    else if ( dir_entry.is_regular_file() ) {
        cout << "Found␣a␣file:␣";
        cout << dir_entry.path().filename() << ",␣";
        cout << dir_entry.file_size() << "␣bytes␣big␣" << endl;
    }
}
```

Copy a file:

```
// path target object
path target_file{source_file};
target_file.replace_filename("Mycopy");

// copy file
copy_file( source_file,target_file );
```

Read a file and print out its content:

```
// open the file
ifstream input_file{target_file};

// read the file and print it out
char c;
while ( input_file >> c )
    cout << c;
cout << endl;

// close file
input_file.close();
```

Change permission of a file:

```
// removes permissions for group and others
permissions( "Mycopy", perms::group_all | perms::others_all,
    perm_options::remove );
```

Delete a file:

```
// delete the file
remove( target_file );
```

⇒ Threads

Defining a function to be run in a separate thread:

```
#include <future>
#include <thread>
#include <chrono>
#include <vector>

using namespace std::chrono;
using namespace std::literals::chrono_literals;

// sleeps for 5s
int sleeping_thread(int n) {
    cout << "Sleeping␣thread␣#" << n << "␣started...";
    cout << endl;
    // pause thread for 5s
    this_thread::sleep_for( 5s );
    cout << "Sleeping␣thread␣ended..." << endl;

    return 0;
}
```

Start an asynchronous thread and wait for its result:

```
// start the sleeping thread with parameter 1
auto sleeping_res = async( launch::async, sleeping_thread, 1 );

// blocks until the thread finishes
if ( sleeping_res.get() == 0 )
    cout << "Sleeping␣thread␣#1␣finished!␣" << endl;
```

Start an asynchronous thread and check periodically until result is available:

```
// start the sleeping thread
sleeping_res = async( launch::async, sleeping_thread, 1 );

cout << "Checking␣on␣sleeping␣thread..." << endl;
auto sleeping_stat = sleeping_res.wait_for( 1s );
while ( sleeping_stat != future_status::ready ) {
    cout << "Checking␣on␣sleeping␣thread..." << endl;
    sleeping_stat = sleeping_res.wait_for( 1s );
}

if ( sleeping_stat == future_status::ready )
    cout << "Sleeping␣thread␣#1␣finished!␣" << endl;
```

Usage of a mutex for synchronizing threads:

```
#include <mutex>

// global counter to be incremented
int counter=0;

// mutex for synchronizing the access
mutex m_counter;

// incrementing thread
template<char c>
int incrementing_thread(int n) {
    // enter infinite loop
    while ( 1 ) {
        // blocking call for getting the mutex
        m_counter.lock();
        // got the mutex
        // check current counter value and exit if needed
        if ( counter >= n )
            break;
        cout << "incrementing_thread<" << c << ">():␣";
        cout << counter << endl;
        counter++;

        // release the mutex
        m_counter.unlock();

        // yield thread execution
        this_thread::yield();
    }
    cout << "incrementing_thread<" << c;
    cout << ">():␣done" << endl;

    // release the mutex
    m_counter.unlock();

    return n;
}

// start the incrementing threads
auto one_res = async( launch::async, incrementing_thread<'1'>, 6);
auto two_res = async( launch::async, incrementing_thread<'2'>, 9);

if ( one_res.get() == 6 )
    cout << "Incrementing␣thread␣#1␣finished!␣" << endl;

if ( two_res.get() == 9 )
    cout << "Incrementing␣thread␣#2␣finished!␣" << endl;

cout << "Counter␣value:␣" << counter << endl;
```

Usage of atomic variables:

```
#include <atomic>

// atomic counter
atomic_int counter_atomic = 0;

template<char c>
int atomic_incrementing_thread(int n) {
    while ( 1 ) {
        cout << "atomic_incrementing_thread<";
        cout << c << ">():␣"
        cout << counter_atomic << endl;

        // atomic read and increment
        if ( counter_atomic < n )
            counter_atomic++;
        else break;

         // yield thread execution
        this_thread::yield();
    }
    cout << "atomic_incrementing_thread<";
    cout << c << ">():␣done" << endl;

    return n;
}

// start the incrementing threads
auto one_res = async( launch::async, atomic_incrementing_thread<'1'>, 6);
auto two_res = async( launch::async, atomic_incrementing_thread<'2'>, 9);

if ( one_res.get() ==  6)
    cout << "Incrementing␣thread␣#1␣finished!␣" << endl;

if ( two_res.get() == 9 )
    cout << "Incrementing␣thread␣#2␣finished!␣" << endl;

cout << "Counter␣value:␣" << counter_atomic << endl;
```

Bibliography

[1] Walter Savitch. *Problem Solving with C++*, 10th edition. Pearson Education, 2018

[2] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*, 2nd edition. Addison Wesley, 2015

[3] Josh Lospinoso. *C++ Crash Course: A Fast-Paced Introduction*, 1st edition. No Starch Press, 2019

[4] Robert C. Seacord. *Secure Coding in C and C++*, 2nd edition. Addison Wesley, 2013

[5] ISO/IEC. *Programming languages – C++*, Sixth edition, ISO/IEC 14882:2020(E)

Appendix A

GNU GENERAL PUBLIC LICENSE

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
   (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

   A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

   A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

   "Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

   If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

   The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
   (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
   (e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
   (f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

   All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

   If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

   Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read
`https://www.gnu.org/licenses/why-not-lgpl.html`.

Index