C++ code snippets

Michele Iarossi*

9th October 2022 - Version 1.53 - GNU GPL v3.0

.

Contents

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

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

*michele@mathsophy.com

Type safety

⇒ Universal and uniform initialisation prevents narrowing conversions from happening:

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

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

Constants

There are two options:

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

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

⇒ **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;
```

Type casting

⇒ 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;
```

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 initialising an object with a complicated initialiser (we might not know exactly when the object gets initialised):

```
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;
```


Lambda expressions

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*.

⇒ 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:

```
// 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){return expe(x,n);},{-8.0,8.0},0.001,
        {-8.0,8.0},{320,240},400};
```

Namespaces

⇒ **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;
```

Random numbers

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

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

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

Arrays

⇒ Range-based **for** statement:

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

for (int& x : arr)
  x++;

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

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;
```

⇒ 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 initialisation
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;
```

⇒ **unique_ptr**: Holds ownership of a dynamic object according to RAII, i.e. resource acquisition is initialisation. It will automatically destroy the object if needed.

```
#include <memory>

MyVector<int>* my_function()
{
    unique_ptr<MyVector<int>*> p { new MyVector<int>};
    /* ... */
    /* if something goes wrong, deletes the object */
   /* ... */
    return p.release(); // returns the pointer
}
```

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:

```
// safe string copy, at most 10 characters are copied
strncpy(msg, srcStr, 10);

// safe string compare, at most 10 characters are compared
strncmp(msg, srcStr, 10);

// safe string concatenation, at most 10 characters are concatenated
strncat(msg, srcStr, 10);
```

⇒ Conversions:

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

Input-output streams

⇒ Input stream **cin**, output stream **cout**, error stream **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";
```

⇒ Integer format manipulators

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

```
#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*.

```cpp
#include <iomanip>

// 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:

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

// 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:

```cpp
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;
}
```

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):

```
#include <fstream>

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

⇒ Opened explicitly (not recommended):

```
#include <fstream>

// 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):

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

Strings

⇒ Strings as supported by the C++ standard library:

```
#include <string>

// initialization
string s1 = "Hello";
string s2("World");
string s3{"World"};
string s4{string(5,'*')}; // "*****"
```

⇒ Concatenation:

```
// concatenation
string s3 = s1 + ",␣" + s2;
```

⇒ 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);
```

⇒ Append:

```
// append
s1.append(s2);
```

⇒ Size and length:

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

⇒ Substring:

```
// substring from position 5 and length 4 characters
string substring;
substring = s4.substr(5,4);
```

⇒ Find:

```
// find (returns string::npos if not found)
size_t pos;
pos = s3.find("World");
if (pos == string::npos)
    cerr << "Error:␣String␣not␣found!\n";

// find starting from position 5
s3.find("l",5);
```

⇒ C-string:

```
// 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);
```

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**

```
#include <sstream>

// 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();
```

Vectors

⇒ Vectors as supported by the C++ standard library:

```
#include <vector>

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

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

⇒ Access:

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

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

⇒ Add:

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

⇒ Resize:

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

⇒ Loop over:

```
// range-for-loop
for (auto x : v)
  cout << v << 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!
}
```

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);
```

Classes

⇒ Class using dynamic arrays:

```
#include <algorithms>

class MyVector
{
public:
    // constructor
    explicit MyVector();
    // explicit constructor (avoids type conversions)
    explicit MyVector(size_t);
    // constructor with initialiser 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 initialising the vector.

```
// constructor with member initialisation 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 initialiser 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 initialiser list
MyVector v2{1,2,3,4};

// copy constructor
MyVector v3{v2};
```

⇒ 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();
```

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);
    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

```
class Euro
{
public:
    // constructor for  euro
    Euro(int euro);
    // constructor for euro and cents
    Euro(int euro, int cents);
    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;
};
```

⇒ 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};
```

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 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 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 = **0** 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
};
```

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()
```

Exceptions

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

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

try
{
    throw My_exception("error");
}
catch (My_exception& 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()
    }
}
```

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 swap(T& a, T& b)
{
    T temp = a;

    a = b;
    b = temp;
}

int a, b;
char c, d;

// swaps two ints
swap(a, b);

// swaps two chars
swap(c, d);
```

⇒ 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 initialiser 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;
}
```

⇒ Integer 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;
```

⇒ 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 initialiser 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 A_list
{
    // constructor with size of the list
    A_list(int size);
    // destructor
    ~A_list();
    // copy constructor
    A_list(A_list<T>& b);
    // assignment operator
    A_list<T>& operator=(const A_list<T>& b);
    // friend insertion operator
    template <class TT>
    friend ostream& operator<<(ostream& outs, const A_list<TT>& rhs);
```

```
private:
    T *p;
    int size;
}
```

Iterators

⇒ An iterator is a generalisation of a pointer. 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 (only read)
vector<int>::constant_iterator c;

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

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

// 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;
```

Containers

⇒ Sequential containers: **list**

```
#include <list>

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

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

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

// erase
e = data.begin();
++e;
data.erase(e);

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

⇒ Adapter containers: **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**

```
#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**

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

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

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

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

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

// erase
dict.erase(two);

// ranged loop
for (auto n : dict)
    cout << "(" <<  n.first << "," <<  n.second << ")" << endl;
```

Algorithms

⇒ Provided by the C++ standard library:

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

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

// find
```

```cpp
p = find(v.begin(),v.end(),13);

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

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

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

References

[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

Index