Library design in C++

Pablo Nicolás Martínez

Barcelona, May 18, 2021

Outline

- Compilation
- Data structures
  - Structures and classes
  - Fields
  - Classes
  - Operator overflowing
  - Special functions
- Libraries
- Makefiles

Compilation in C++

The compilation in C++ (and, indeed, in C) is divided in several steps due to how the language works. This process is complicated and is not of our interest.

In an intermediate step in compilation, compilers produce what are called *object files* with the extension `.o`. An object file contains different functions from the code which have not been already linked.

A linker patches together individual functions with others to produce an executable file.

Compilation in C++

Object files are important for solving cross-dependencies between code. When compiling multiple files with dependencies, object files have to be produced first and then linked together to produce the executable file.

Compilation in C++

Object files are important for solving cross-dependencies between code. When compiling multiple files with dependencies, object files have to be produced first and then linked together to produce the executable file.

The following code gives as output a list of unlinked functions:

```
objdump -dr filename.o
```

Outline

Compilation

Data structures

- Structures and classes
- Fields
- Classes
- Operator overflowing
- Special functions

Libraries

Makefiles

Motivation

Many times in programming we find that certain codes or routines can be identified as part of more general structures (for example, bfs or dfs are algorithms that are naturally described over graphs). In more complex algorithms and larger systems, having these kind of structures and identifications becomes essential.

Motivation

Many times in programming we find that certain codes or routines can be identified as part of more general structures (for example, bfs or dfs are algorithms that are naturally described over graphs). In more complex algorithms and larger systems, having these kind of structures and identifications becomes essential.

Our objective is to describe and implement these abstract objects, which are called *abstract data types*. They provide the following features:

- *Abstraction:* Considerations can be made over complete algorithms without focusing on details.
- *Specification and modularity:* Operations have a concrete and independent meaning, being able to change a part of code without affecting the rest.
- *Reuse:* Code can be called from other programs.

Example 1: Identification card

People can be classified by many different parameters. We can choose some of them (which are not exhaustive) as

- National identification document.
- Date of birth.
- Name and surname.

Example 2: Vectors

Vectors are data structures which store information. Therefore, they can be completely characterized by the following information:

- Length.
- Real numbers in each position (doubles).

However, we also have algebraic operations defined over them. Some are

- Addition and subtraction of vectors.
- Product and division by a scalar.
- (Standard) Scalar product of vectors.
- Hölder norms for different integer numbers.

Example 3: Chess

Chess (and many board games, actually) can be characterized by some parameters

- The complete board.
- A position on the board.
- The pieces and their elementary movements.
- A list of active and inactive pieces.

Note that chess would not likely be implemented using a single class. A complete chess would be composed by many interacting classes and structures.

Structures and classes

The objects in which we implement these levels of abstraction are `structures` and `classes`. Classes are only different from structures in the sense that there are certain types of data that can only be accessed by methods of the class, and not by external procedures.

Structures and classes

The objects in which we implement these levels of abstraction are `structures` and `classes`. Classes are only different from structures in the sense that there are certain types of data that can only be accessed by methods of the class, and not by external procedures.

We can distinguish different categories inside of a data structure:

- Special members.
- Fields.
- Overloaded operators.
- Functions and external functions.

Privacy rules

The privacy rules for classes are the following:

- *Public:* Data that can be accessed, used or modified by any other structure.
- *Private:* Data that can only be accessed or modified by other members of the same class, and that is forbidden for external routines.
- *Protected:* Data that can only be accessed by special classes derived from the initial one, called *derived* classes.

The declaration of derived classes and other concepts like inheritance are beyond the scope of this course.

Fields

Fields are the basic data types that a data structure can have. Even though they are not required to have a data structure, they are normally used to store information relevant to the class.

Fields

Fields are the basic data types that a data structure can have. Even though they are not required to have a data structure, they are normally used to store information relevant to the class.

A field is declared within a class or structure as any regular variable within the code. Note that any data structure can be declared as a member of another class.

Functions

Functions are routines that are fully contained within the class. They can modify elements from the class or act on other structures.

Functions

Functions are routines that are fully contained within the class. They can modify elements from the class or act on other structures.

Functions can be declared inside or outside the class. If they are declared outside of the class, they may not access members of a class which are private. To be able to do these things there are *friend* functions to a class. These functions may access the elements of the class and modify them.

Functions

Functions are routines that are fully contained within the class. They can modify elements from the class or act on other structures.

Functions can be declared inside or outside the class. If they are declared outside of the class, they may not access members of a class which are private. To be able to do these things there are *friend* functions to a class. These functions may access the elements of the class and modify them.

A general practice when working with private members is to declare particular functions which only serve as information. This way, a user can call them to see the information stored in a class but it cannot modify it.

Operator overloading

Operators are special symbols in C++ which can perform actions on classes. Among the more typical ones we can find the arithmetic operations +, -, * and /, or the assignation operator =.

Operator overloading

Operators are special symbols in C++ which can perform actions on classes. Among the more typical ones we can find the arithmetic operations +, -, * and /, or the assignation operator =.

One can *overload* operators to act on defined classes. This is called operator overloading, and would allow, for example, to define the sum of two graphs.

Operator overloading

Operators are special symbols in C++ which can perform actions on classes. Among the more typical ones we can find the arithmetic operations +, -, * and /, or the assignation operator =.

One can *overload* operators to act on defined classes. This is called operator overloading, and would allow, for example, to define the sum of two graphs.

Operators can be declared inside or outside of a class, just like functions. The syntax to overload an operator depends on where they are declared, so be careful!

Operator overloading

Operators are special symbols in C++ which can perform actions on classes. Among the more typical ones we can find the arithmetic operations +, -, * and /, or the assignation operator =.

One can *overload* operators to act on defined classes. This is called operator overloading, and would allow, for example, to define the sum of two graphs.

Operators can be declared inside or outside of a class, just like functions. The syntax to overload an operator depends on where they are declared, so be careful!

One can overload operators even from other libraries, like the standard library.

Operator overloading

Expression	As member	As non-member	Operators
@a	(a).operator@()	operator@(a)	Prefix
a@	(a).operator@(0)	operator@(a,0)	Posfix
a@b	(a).operator@(b)	operator@(a,b)	Infix
a=b	(a).operator=(b)	Cannot be non-member	=
a(b...)	(a).operator()(b...)	Cannot be non-member	()
a[b]	(a).operator-¿()	Cannot be non-member	->

There are special properties or conventions regarding some operators. See `https://en.cppreference.com/w/cpp/language/operators`.

Special functions

Special functions are intrinsic to classes and structures. They serve to declare internal procedures on how a class is created and destroyed on the execution of the code.

Special functions

Special functions are intrinsic to classes and structures. They serve to declare internal procedures on how a class is created and destroyed on the execution of the code.

The first block are *constructors*. They are functions that specify how the variables are initialized and the different declarations that a class accepts. They are declared as a function with no return type, with the name of the class and with a set of parameters. A function of this kind with no input arguments is called the *default constructor* of the class.

Special functions

Special functions are intrinsic to classes and structures. They serve to declare internal procedures on how a class is created and destroyed on the execution of the code.

The first block are *constructors*. They are functions that specify how the variables are initialized and the different declarations that a class accepts. They are declared as a function with no return type, with the name of the class and with a set of parameters. A function of this kind with no input arguments is called the *default constructor* of the class.

The second type is the *destructor* of the class. It handles the destruction of the class and all of its elements when the code ends. It is declared as the constructors with a tilde (~) beforehand.

The rule of three

There is a rule called the *rule of three*, which states that

"If any of the copy constructor, the copy assignment or the destructor are declared, then all of them should be declared and be compatible among them."

The rule of three

There is a rule called the *rule of three*, which states that

"If any of the copy constructor, the copy assignment or the destructor are declared, then all of them should be declared and be compatible among them."

This is a convention, but it is rather useful as it allows us to write formulas like:

```
matrix A;
matrix B;

matrix C = A + B;
```

Outline

Compilation

Data structures

- Structures and classes
- Fields
- Classes
- Operator overflowing
- Special functions

Libraries

Makefiles

Libraries

Libraries are collections of code which implement structures, routines or procedures. They can be used to solve specific problems, to have general structures for multiple purposes or for simply reusing code.

Libraries

Libraries are collections of code which implement structures, routines or procedures. They can be used to solve specific problems, to have general structures for multiple purposes or for simply reusing code.

The structure of libraries is (mostly) imposed by the structure of classes and structures. Structures are *defined* in what are called *header* files (extension `.hpp`). A header file contains the declaration of the class.

Libraries

Libraries are collections of code which implement structures, routines or procedures. They can be used to solve specific problems, to have general structures for multiple purposes or for simply reusing code.

The structure of libraries is (mostly) imposed by the structure of classes and structures. Structures are *defined* in what are called *header* files (extension `.hpp`). A header file contains the declaration of the class.

The routines of code of a class can be implemented in one of the two following ways:

- Declare them in the header file.
- Create another file containing the methods. This file should have the extension `.cpp` and the same name as the header file.

Libraries

Libraries are structured commonly following this structure:

- Include file, containing the header files with the declaration of the procedures and classes.
- The source files, which store the routines and procedures associated to a given class.
- A temporary directory, in which the auxiliary `.o` files are stored.

The rest of the structure is mostly up to the programmer's will. We will see this when we analyze different repositories of code.

Outline

Compilation

Data structures

- Structures and classes
- Fields
- Classes
- Operator overflowing
- Special functions

Libraries

Makefiles

Makefiles

We have seen that compiling code is hard and tedious most of the times. Moreover, many dependencies have to be handled manually.

C++ offers a way to handle the compilation of code in a repository through the use of a *makefile*. It is a special file that can store directories, routines and orders. It provides a simple and unified way to declare dependencies, access to specific directories and declare routines for workflows (for example, execute a set of test files for the code).

Makefiles are commonly tailored for repositories, and thus we shall analyze them with examples.

Orders are declared typing the name of the order followed by two semicolons. Rules for specific files are declared with the name of the respective file.

After the semicolon we specify the dependencies for the action, necessary files for the action to be performed.

We can also specify concrete actions bound to the order in a new line, properly indented. These consist of shell scripts.

Phony targets

One can declare orders which are not bound to specific files, but rather to arbitrary actions. These are called *phony* targets.

Phony targets are are declared as regular ones. A special line called `.PHONY` declares the targets which are phony commands.

```
clean:
   rm -f foo.c foo.o bar.c bar.o
.PHONY: clean
```

Makefiles can use *environmental variables* to perform tasks. These variables allow to declare parameters to be used in other parts of the code, following the philosophy of modularity and specification.

Environmental variables are declared by their name followed by an equal sign and some arguments. They are called with a dollar sign and between parentheses:

```
SNOZFLAGS=--with-extra-xyzzyobjdump -dr filename.o
\%.c: \%.snoz
    snozzle \$(SNOZFLAGS) \$< -o \$@
```

Functions are instructions that can be used to perform several tasks. For more information see `http://wiki.wlug.org.nz/MakefileHowto`.

We can also include an order to not regenerate a file if its dependencies have not been modified since the date of compilation. This is achieved with a `Makefile` line after the dependencies.

Moreover, compilers like `gcc` can automatically compute dependencies.

```
DEPS := \$(patsubst \%.o,\%.d,\$(OBJS))
```