Classes versus Structs

In the introductory slides, I wrote that the main achievement of object-oriented programming is the equation:

$$\textbf{datatype} = \textbf{productofothertypes} + \textbf{setaccessfunctions}.$$

For every non-primitive data type D holds the following: There exists data types $D_1, \ldots, D_n$, s.t.

$$D \subseteq D_1 \times \cdot \times D_n,$$

but

$$D = D_1 \times \cdot \times D_n$$

holds rarely.

Remembering which $(d_1, \ldots, d_n)$ are well-formed D can be difficult and unpleasant.

Use of access control allows the user to forget about ensuring that

$(d_1, \ldots, d_n)$ really is a D.

Classes versus Structs

- In case of

  $$\mathbf{date} \subseteq \{1, \ldots, 31\} \times \{\mathbf{jan}, \ldots, \mathbf{dec}\} \times \{-2000, \ldots, 5000\},$$

  it is quite hard to know which triples are well-formed dates.

- In case of

  $$\mathbf{list} \subseteq \mathbf{int} \times \mathbf{pointertolist},$$

  it is hard to keep the memory in order.

- In case of

  $$\mathbf{provenformula} \subseteq \mathbf{formula},$$

  it is hard to know which formulas a proven.

Defining classes in C^{++}

A class in a C^{++} programs should be defined in two files, which are called after the class.

1. The first file should be called **class.h**. It should define the members of the class, and declare the member functions.

   In addition, it is acceptable to implement some small member functions in the **class.h** file. The **class.h** file is called the header file of **class**.

2. The second file should be called **class.cpp** or **class.cc**. It should contain the definitions (implementations of most of the member functions.

Separation between header files and implementation files

There are two reasons for the distinction between **.h** and **.cpp**:

1. It enables separate compilation. A big program can be spread over many files. In case something changes in the implementation of some class, only its corresponding **class.cpp** file needs to be recompiled.

2. It allows a user of the class to read the specification of a class, without having to understand how it is implemented.

Separation between **.h** and **.cpp**

If you put comments in your file (which you should), the comments must be separated accordingly. Comments about specification and usage of the class go into the **class.h** file.

Comments about implementation go into the **class.cpp** to the function to which they belong.

The **.h** file

```
// This is the date.h header file. Normally, such obvious
// things should not be written in a comment, but this
// is the first time.

#ifndef DATA_INCLUDED
#define DATA_INCLUDED  1
   // This is called the include guard. Its purpose is
   // to avoid that the same file is included twice.
   //
   // As far as I can see, nothing bad happens when a
   // header file is included twice, but it is
   // inefficient.
```

```
    class date
    {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    };

#endif
```

Note that there is a design flaw in the language C^{++}. It should be the case that everything that concerns implementation is in the **.cpp** file. This means that also the declarations of **day, month, year** should be in the **.cpp** file.

Unfortunately, this is not the case. This is due to technical reasons. (The compiler needs to know how much space to reserve for the object, and it needs to be able how to fill in default operators)

Alternative Implementations of Date

Alternatively, one could for example define :

```
class date
{
   int day;
      // 1 jan 2000 has 0, 2 jan 2000 has 1, etc.
};
```

One could also make month into a separate class. Considerations: Does month occur alone? How much work is it to make month a separate class? (Take into account that date becomes easier) A fundamental property of month (its length) cannot be determined without knowing the year. Therefore, month seems incomplete.

Essential and Important Member Functions

Whenever you define a class, you should define it in such a way that it can be used in programs in a convenient way.

In order to obtain this, you should give attention to the following member functions:

1. Essential member functions. They involve creation, assignmnent destruction. (C^{++} provides defaults. You need to implement them only, when the defaults are not right)
2. Important Member functions: Printing, operators.

The Essential Member Functions

- Default constructor. The default constructor constructs an object out of nothing. It should construct a special (default) value. Not every class has a reasonable default value. If no default value exists, you should not provide a default constructor. This is the case for **date**.
- Copy constructor. The copy constructor constructs a new object from another object. The copy constructor is used when passing a parameter to a function, and possibly also when returning the result of a function.
- Copying assignment. The assignment copies one object into another object. The difference with the copy constructor is that in case of assignment, an existing object is overwritten.
- Destructor. The destructor is called when the object goes out of scope.

The Essential Member Functions (2)

The copy constructor, copying assignment and destructor are important when the data structure has memory on the heap (like **list** or **vector**) or open files. In case of **date**, they are not important.

Defaults of the Essential Member Functions

- If no other constructors are defined, then the default of the default constructor is: Apply on every member of the class its default constructor. If one of the members has no default constructor, the result is a compile error.
- If no copy constructor is defined, the default is to call the copy constructor for every member.
- If no copying assignment is defined, the default is to call copying assignment for each of the members.
- If no destructor is defined, the default is to call the destructor of every member.

Other important member functions

- An exchange operator. For big objects that exists on the heap, this may be useful. Otherwise, you can always write the code

```
{
    date s = d1;
    d1 = d2;
    d2 = s;
};
```

  There is no default exchange operator.

- A print operator `<<` . There is no default print operator. (Attempts to print an object without print operator causes long, incomprehensible error messages)

- Other constructors.

- Member functions (the access functions)

- Other operators. (Like `++`, `--`, `+`, `=`, `*`)

Where the Essential Member Functions are Used

```
{
   date d;
      // Default constructor.
   date d1(1,2,3);
      // Other constructor from three integers.
   date d2 = date::date(1,2,3);
      // Other constructor, the same as previous.
   date d3 = d2;
      // Copy constructor, because d3 is constructed
      // for the first time.
   d2 = d3;
      // Copying assignment, because d2 exists already.
}   // Destructors for d1, d2, d3 are called.
```

Where the Essential Member Functions are Used (2)

```
date something( date d )
{
   // When called, the copy constructor is used to
   // make a copy into d.

   date result = d;
      // Copy constructor is used to make a copy of d.
   return result;
      // Dependent on how clever the compiler is,
      // the result is returned with a copy
      // constructor, or computed on the right place.
}
   // Destructor for d.
   // If result was computed on the right place, no
      // destructor for res is called.
```

Where the Essential Member Functions are Used (3)

```
{
   date *p = new date( );
      // Default constructor.
   date *q = new date::date(1,2,3):
      // Other constructor.

   *p = *q;
      // Copying assignment.

   delete p;
      // Destructor.
   delete q;
      // Destructor.
}
```

Where the Essential Member Functions are Used (4)

```
{
   unsigned int k = 40;

   date *p = new date [k];
      // fourty times the default constructor.

   delete p;
      // fourty times the destructor.
}
```

Also, in all of the standard containers, like
`std::list`, `std::vector`, `std::map`, etc.

Where the Essential Member Functions are Used (5)

```
{
   date *p = new date( );
      // Default constructor.
   date *q = *p;
      // Copy constructor.

   p = q;
      // At this point, *p is simply lost. No special
      // function is called.
}
   // The pointers p and q go out of scope, but *q
   // remains in memory. No special function is
   // called.
```

Member Functions for Date

What member functions should **date** have?

- A constructor from a triple of unsigned integers.
- A constructor from a single integer?
- Functions

```
unsigned int getyear( );
unsigned int getmonth( );
unsigned int getday( );
unsigned int yearlength( );
unsigned int monthlength( );
bool isleapyear( );
```

- Increasement/descreasement $++, --$
- A function $-$ that computes the difference in days between two dates.
- Functions $+, -$ that can add an integer to a date?
- Operators `<`, `>`, `<=`, `>=`, `==`, `!=` ?
- Operators `+=`, `-=` ?

Essential Operators

Like all methods, the essential operators can be short or long.

Short definitions are in the .h file.

Long definitions go into the .cpp file.

Default Constructor for Date

```
date( )
   : day(1),
     month(12),
     year(1972)
        // These are initializers.
{
   // Other code.
}
```

Default Constructor for Date

A constructor can be recognized from the fact that it has the same name as its class.

The default constructor can be recognized by the fact that it has no arguments.

You should always give initializers. Fields that you don't initialize, are initialized by their default constructors.

Initizations happens in the order determined by the class definition, not by the order in which you write them in the constructor.

Bad Default Constructor for Date

```
date( )
{
   day = 1;
   month = 12;
   year = 1972;
}
```

This may appear equivalent, but now the fields will be initialized with their default constructors, and after that overwritten with copying assignment.

(Note that date does not need a default constructor, the constructors are illustration only)

Copy Constructor for Date

As said before, **date** does not need a copy constructor, because copying the integers is perfectly fine.

```
date( const date& d )
   : day( d. day ),
     month( d. month ),
     year( d. year )
        // Initializers.
{
}
```

Copying Assignment for Date

All of the following declaration are possible:

```
R operator = ( D d );

// R is either:
//    void,
//    date
//    date&
//    const date&
// D is either:
//    date
//    date&
//    const date&
```

Destructor for Date

```
~date( )
{
   // Do what should be done in order to
   // clean up a date: In this case:
   // nothing.
}
```