Process of Compilation

Program development proceeds as follows:

1. Edit the program in some text editor. The result is a text file.
2. Compile this text file into machine code.
3. Run the resulting machine code.

Interpretation versus Compilation

Some languages, must notably Java and $C\#$ use interpreters.

The compiler does not compile into native machine code, but into an intermediate code. (Java Byte Code).

1. Programmer writes program in some text editor.
2. Compiler translates into Java Byte Code.
3. Interpreter (Java Virtual Machine) reads the Java Byte Code, and executes its instructions. (You can guess that the JVM conists mostly of 'ifs' of the form 'if next instruction is ... then do ...'. Interpretation is necessarily slower than native code. It also uses more memory, and more energy.
4. There exist Java compilers (to native code). I don't know how efficient these are.
5. Main advantage of interpretation is higher portability.

Separate Compilation

- Compilation can be quite time consuming for big programs, even on a fast computer.
- Different parts of a program are written by different people.
- Some contributors may want to hide their sources from you.

Because of this, we have separate compilation.

Separate Compilation (2)

- Program consists of different text files, which can be edited separately.
- For each of the files separately, an object file is created. (They have extension .o in Linux, .obj in Windows.) You can use `nm filename.o --demangle` to see what is inside an object file.
- The linker collects all object files, fills in the cross references (mostly addresses of function calls) and creates a single executable file. (No extension in linux, .exe extension in Windows.)

Separate Compilation (3)

Unfortunately, completely separate compilation is impossible.

File **rational.cpp** defines rational numbers and operations on them.

Another file, **matrix.cpp** defines a matrix as an array of rationals, and defines operations on matrices.

Main file uses matrices and rationals.

- When producing a local variable of some user defined type, compiler needs to know how much space to reserve.
- When compiling `v.x`, the compiler needs to know that `v` has a field `x`, and which type it has.
- When seeing a function call `f(m)` the compiler needs to know that this function exists, and what type it returns.

file.h versus file.cpp

Every program file must be split into two files: **file.h** contains:

- Declarations of available types (classes and structs).
- Declarations of available functions.
- Definitions of small functions that are inlined. Inlining means that the compiler replaces the function call by its definition. This is useful for small functions that are called very often.

The rest stays in file **file.cpp**.

Including

The method for reading the necessary .h files is as primitive as you can imagine. Write `#include "file.h"` at the top of your .cpp file, for every file whose information you need. (For system defined files, it is `#include <library>`)

It you have nested dependencies, then it is quite possible that the same gets included twice, which not good. (The number can even grow exponentially.)

(For example, **main** may use **rational.h** and **matrix.h**, which in turn uses **rational.h**.

In order to avoid this, use include guards.

```
#ifndef MATRIX_INCLUDED
#define MATRIX_INCLUDED 1
   // Name must be unique and in capitals.
#include "rational.h"
struct matrix
{
   rational repr[2][2];
};

matrix operator * ( matrix m1, matrix m2 );
matrix operator * ( rational r, matrix m );
matrix operator + ( matrix m1, matrix m2 );
   // Declare a lot of operators.
};

#endif
```

Linking

The linker collects all the .o files, and makes a complete program from it.

It looks for functions that are used in one of the files, and tries to find a definition in one of the other files.

If the linker cannot find a definition, you will get an error message that is rather unpleasant:

```
test.o: In function ‘rational::rational(int, int)’:
test.cpp:(.text._ZN8rationalC2Eii[_ZN8rationalC5Eii]+0x2d):
               undefined reference to ‘rational::normalize()’
collect2: error: ld returned 1 exit status
```

Make

If you edit a .cpp file, then only its corresponding .o file needs to be recompiled.

If you edit a .h file, only the the .o files of the .cpp files that `#include` it, need to be recompiled.

make does this automatically.

Makefile

```
program:  f1.o f2.o f3.o
        g++ -o program f1.o f2.o f3.o

f1.o: f1.cpp f2.h f3.h
        g++ -c f1.cpp -o f1.o


f2.o: f2.cpp f1.h f3.h
        g++ -c f2.cpp -o f2.o
```

Specify for each file that the compiler/linker constructs, the files from which it is constructed.

After that, give the command that does the construction. (The 8 spaces are a Tab.)

The linker is also called `g++`.

Types of Variables

C^{++} has many types of variables, which is confusing to many people. We have seen before that the default behaviour for assignment, initialization and parameter passing in C^{++} is copying. This ensures that variables are independent of each other, which makes the program easier to understand, and close to logical semantics.

Sometimes you don't want to copy for two possible reasons: A function must be able to change a variable, and copying large objects can be inefficient.

A reference is a short-lived variable that doesn't have a contents of its own, but which shares its contents with another variable.

References are used for parameter passing and for abbreviating large variable expressions.

Use of References in Parameter Passing

```
matrix operator + ( const matrix& m1, const matrix& m2 );
   // There is no need to make a copy of matrix m1, m2
   // in order to add them. The `const` keyword indicates
   // that the reference will never change the value
   // it refers to.

std::ostream& operator << ( std::ostream& stream,
                            const matrix& m );
   // Similarly, there is no need to copy a matrix
   // in order to print it.
```

If you have a parameter that could be copied in principle, but you worry about efficiency, then use `const&`.

Use of References in Parameter Passing

```
void operator += ( matrix& m1, const matrix& m2 );
   // The += operator adds the second matrix to the
   // first. The second parameter could be just 'm2'
   // but that would be inefficient, so we made it
   // const reference.
   // The first parameter must be a reference,
   // because += must be able to change it.
```

If you want a function to be able to change something through one of its parameters, then use `&` without `const`.

Note that functions should be preferred over non-const references in general.

Use of References as Abbreviation

Sometimes, variable expressions are long and repeated:

```
for( i = 0; i < 100; ++ i )
   for( j = 0; j < 100; ++ j )
      p [i][j]. field = p[i][j]. field + 1.0;


 ==>


 for( i = 0; i < 100; ++ i )
    for( j = 0; j < 100; ++ j )
    {
       double& d = p[i][j]. field;
       d = d + 1.0;
    }
```

You should never do this for efficiency, only for readability.

The compiler finds this optimization, when optimization is on.

References should always be short-lived!

Rvalue References

Dilemma:

```
matrix inverse( const matrix& m );
   // No copying. Efficient, but we cannot change m
   // during computation.
matrix inverse( matrix m );
   // Copying. More costly, but we can use m as
   // scratch area during computation.
```

Another type of reference: We take a reference to some variable, and the reference is the last user of the current value of the variable.

```
matrix inverse( matrix&& m );
   // Not copied, but we can use m as scratch.
   // Nobody cares, because we are its last user.
```

Rvalue References (2)

Rvalue references can be used when main variable either gets overwritten, or goes out of scope:

```
{
   matrix m1 = ...
   std::cout << inverse( m1 );
      // m1 is going to be overwritten.
   {
      matrix m2 = ...
      m1 = inverse( m2 );
         // m2 goes out of scope.
   }
}
```

Don't worry about Rvalue references now. We will come back to this later.

Pointers and Iterators

Other forms of sharing variables are:

- Pointers: A pointer can be 0 (not referring to anything at all), or share with different variables during its life time.

  Pointers are also used to create objects on the heap. (Not in a local variable.) This may cause memory leaks. We will discuss this delicate topic later.

- Iterators: An iterator always shares with an element of a container. (array, hashmap, vector, list). Iterators are a convenient way of processing elements of a container. For many container types, they are the preferred way of reaching the elements. Iterators should be preferred over pointers.

Classes

Class = Representation + Invariants + Equivalences.

Suppose that we want to implement rational numbers of form $\frac{p}{q}$.

A rational number can be represented by a pair of integers (p, q).

(p, q) and (pn, qn) represent the same number.

- Either we use an invariant, that p, q have no common factors and $q \geq 0$. In that case we do not need equivalences.
- Or we use no invariant, and we design our class in such a way that (p, q) and (pn, qn) are indistinguishable.

Constructors

Methods with the same name as the class they occur in, are called constructors.

Constructors can have different types:

- The default constructor has no arguments.
- The copy constructor has one argument of type `const C&`, `C&&`, or `C&`.

Assignment versus Initialization

In C^{++}, there is distinction between initialization and assignment.

```
{
   rational r = rational( 1, 2 ); // Initialization.
   r = rational( 3,4 ); // Assignment.
}
```

In the first case, variable r is initialized for the first time. In the second time, an existing value is overwritten.

This distinction is important, because in the case of assignment, the old object may hold resources that need to be returned to the system.

Static Member Functions

Static functions are functions that do not have access to the fields of the class.

Because of this, they can be called without class object.

There are are two reasons to do this.

1. The function is very much connected to the class.
2. The function acts as additional constructor.