Overloading, Operators, Constness, Initializer Lists

Testing

It seems that some people don't know how to test.

Test all your components one by one. Always test the borderline cases. (Empty containers, the numbers 0, 1, self assignment.) Make sure that every of the code is executed during the tests. Use print statements to check intermediate results.

Use **for** loops to check for memory leaks.

Move to the next component, only when you believe completely in the previous component.

Avoid big functions. It is better to split them up. If you worry about efficiency, use **inline**.

If your code cannot be tested, it is probably badly written.
Untested code is almost the same as unwritten code.

Const Modifier for Member Functions

Member functions of a class can be declared **const**.

```
const double& top( ) const
```

If variable s is declared as `const stack s` or `const stack& s`, then only member functions with **const** mark can be used on s.

A **const reference** can never be copied into a non-const reference.

The compiler helps with preserving **const**.

Preservation of Const

Inside a **const** member function, the fields of the class are available only as const:

```
double top( ) const
{
   current_size = 4;  // Will be refused because
          // current_size has type const unsigned int
}
```

From a **const** member, it is not possible to call a non-**const** member:

```
   double top( ) const
   {
      change_size(4);
        // Will be refused, unless change_size is const.
   }
```

Preservation of Const

It is still possible to cheat on the heap, because pointer dereference does not preserve **const**:

```
double changetop( ) const
{
   tab[0] = 4;
      // Will be accepted. Still a very bad idea.
      // because we want to treat *tab as a kind of
      // extended local variable.
}
```

In order to prevent this, we should make sure that only member functions can access the heap (which we are treating as extended local variable space) directly.

Overloading with Const

Consider:

```
double& top( )
{
   return tab [ current_size - 1 ];
}

double top( ) const
{
   return tab [ current_size - 1 ];
}
```

Compiler will pick first whenever possible, otherwise second.

Overloading with Const

You can write:

```
      stack s;
      s. push(4); s. push(5);
      s. top( ) = 3;
```

but not

```
std::ostream&
operator << ( std::ostream& out, const stack& s )
{
   s.top( ) = 0;
}
```

Overloading

The previous example was a special case of overloading.

Overloading is when two or more functions or class methods have the same name. In that case, the compiler has to decide which one to use.

Overloading happens all the time. Consider `<<`, `=`, `+`,`*`,`-`,`/` and other operators.

I will define the rules:

Conversion Levels

C^{++} distinguishes levels of conversions:

Level 1 The conversion from T_1 to T_2 has level 1, if $T_1 = T_2$, or T_2 can be obtained from T_1 by inserting **const**.

The conversion from arrays to pointers (which is unavoidable because nothing else can be done with an array) is also level 1.

Application of copy constructors is also on level 1.

Level 2 The conversion from T_1 to T_2 has level 2, if both T_1, T_2 are integral. (**bool, char, int, short, long, unsigned**), and the conversions from T_1 to T_2 is guaranteed to be without loss.

The conversion from **float** to **double** is also level 2.

Level 3 The conversion from T_1 to T_2 has level 3, if both T_1, T_2 are integral, but the conversion from T_1 to T_2 is possibly lossy. (For example from **int** to **char**, from **unsigned int** to **int**, or from **int** to **double**.

Also conversions from a derived class to a base class are level 3.

Level 4 The conversion from T_1 to T_2 is level 4 if it involves a user defined conversion. (A one argument constructor.)

One Argument Functions

Suppose we have a function call $f(t)$ with t of type T. Assume that

$$U_1 \ f(T_1), \ldots, U_n \ f(T_n)$$

are the definitions of f that the compiler has to choose from.

If there is no T_i, such that T is convertible into T_i, then: ‘No matching definition found’.

Otherwise let λ be the level of the lowest conversion.

If there is unique T_i into which T can be converted at level λ, then $U_i \ f(T_i)$ is selected.

If T_i is not unique, then: ‘Ambiguous overload for f’.

Multi Argument Functions

Assume that function f is applied on arguments $t_1, \ldots, t_n$ with types $T_1, \ldots, T_n$.

Assume that

$$U_1 \ f(T_{1,1}, \ldots, T_{1,n}), \ldots, U_n \ f(T_{m,1}, \ldots, T_{m,n})$$

are the definitions of f that the compiler has to choose from.

If there is no i, such that each T_j is convertible into its corresponding $T_{i,j}$, then 'No matching definition found'.

Multi Argument Functions (2)

Otherwise, find an i such that each T_j is convertable into its corresponding $T_{i,j}$, and for every other i', s.t. each T_j is convertable into $T_{i',j}$, the following holds:

1. For every argument position j, the level of the conversion from T_j into $T_{i,j}$ is not higher than the level of the conversion from T_j into $T_{i',j}$.
2. There is at least one argument position j, for which the level of the conversion from T_j into $T_{i,j}$ is lower than the level of the conversion from T_j into $T_{i',j}$.

It probably took several years to find these rules. They are key to the success of C^{++}.

The rules work so well that you never notice them.

Overload Resolution in C^{++}

```
double f( int, double );
double f( double, int );

...
f(0,0);   // Ambiguous.

double g( int, double ),
double g( double, int ),
int g( int, int );

g(1,1) + g(1.0,4) + g(5,5.0); // Fine.
```

Private Member Variables

In the **rational** class, we wanted to use controlled access to the fields as tool to enforce the class invariants/equivalences.

```
struct rational
{
   int num;
   int denum;
   rational( ); // Default constructor.
   rational( int i );
   rational( int num, int denum )
      : num( num ), denum( denum )
   {
      normalize( );
   }
};
```

If we could be sure that the fields **num**,**denum** cannot be overwritten, we are sure that the invariant is preserved.

Unfortunately, somebody may still not know about our delicate class invariants, and write

```
rational operator + ( const rational& r1,
                      const rational& r2 )
{
   rational r;
   r. num = ..
   r. denum = ...
}
```

(In addition, it uses unnecessary default initialization.)

Private Fields

Solution is to declare the fields **num** and **denum** private:

```
struct
{
private:
    int num;
    int denum;
public:
    ... constructors.
};
```

Private members (fields and functions) can be accessed only from member functions.

One can also write

```
class rational
{
    int num;
    int denum;
public:
    (constructors)
};
```

`class` and `struct` are exactly the same. In a class, the fields are `private` until the word `public:` appears. In a struct, the fields are `public` until the `private:` appears.

The `private/public` distinction also applies to class methods.

Getters

Unfortunately, making **num** and **denum** private also blocks reading, so that `operator+` cannot be implemented anymore.

Reading should be allowed because it cannot spoil the invariant, and we have nothing to hide.

```
int getnum( ) { return num; }
int getdenum( ) { return denum; }
```

The functions make it possible to read the fields without changing them.

Functions that are defined in the class definition, (nearly always in the **.h** file) are inlined, i.e. substituted away by the compiler. They have no computational cost at run time.

C used macros for this. Don't use macros in C^{++}!

Why macro's are bad

Macro's are syntactically not safe, and may evaluate their argument more than once.

```
   #define SQUARE(X)    ( X * X )

   SQUARE( 4 + 4 );
```

Macro's ignore all scoping rules:

```
   class mysecrets
   {
#define NRSECRETS 100
   };
```

No private/public distinction, scope is always global, no way to control overloading. Different programmers may use them with different definitions. Don't use macro's!

This

Inside a member function of a class, the class object that we are a member of, is accessible as this.

Very unfortunately, **this** is always a pointer. It would be much better if it were a reference. I cannot help it.

this should be used in three situations:

- A local variable (or a parameter) in a class method has the same name as a field or method of the class.
- You want to apply a defined operator, which is defined as member, on the current class object.
- You want to make a copy of the current class object. (This happens all the time in `X operator ++ ( int )`)

```
class rational
{
   int num, denum;
   // Unnecessary assignment operator, created only
   // for the example:
   void operator = ( const rational& r ) { ... }

   // Square the current rational:
   void square( )
   {
      operator =
         ( rational( num * num, denum, denum ) );
      (*this) = rational( num * num, denum, denum );
         // Looks better.
   }
};
```

In most cases, class fields and class methods can be accessed without **this**.

In initializers, there is no need to use **this**:

```
rational( unsigned int num, unsigned int denum )
   : num( num ), denum( denum )
{ }
```

Defining User Operators

C^{++} allows the definition of user operators.

There is no way to extend syntax, only operators that that already exist, can be overloaded.

Simple Operators

Simple operators `+,-.*,/,%,&&,||,^^` can be overloaded, either as member, or stand alone.

Definition as Member

In file **rational.h**:

```
class rational
{
   int num, denum;

   rational operator + ( const rational& r ) const;
};
```

A member operator is like a normal member function. It has access to the private variables.

This means that more discipline is required when writing them.

It also causes asymmetry between first and second argument, which is not always nice.

If you give a full definition, it will be inlined.

Definition as Member (2)

In file **rational.cpp**:

```
rational rational::operator + ( const rational& r ) const
{
   return rational( num * r. denum + r. num * denum,
                    denum * r. denum );
}
```

Stand Alone Definition

In **.h**:

```
class rational
{
   int num, denum;
};

rational
operator + ( const rational& r1, const rational& r2 );
```

Stand Alone Definition (2)

In **.cpp**:

```
rational
operator + ( const rational& r1, const rational& r2 )
{
   return rational(
      r1. num * r2. denum + r1. denum * r2. num,
      r1. denum * r2. denum );
}
```

No unwanted access to private variables. Nicely symmetric.

If you want it inlined, you can define it in the **.h** file, and write **inline** in front of it. (If you forget the word **inline**, you will get linker complaints.)

Overloading of Assignment

We have already seem overloadings of assignment. The syntax is the same as for the other binary operators:

```
struct rational
{
   ...
   void operator = ( const rational& r );
   void operator ( rational r );
      // Results in a call of copy constructor.
};

void operator = ( rational& r1, const rational& r2 );
void operator = ( rational& r1, rational r2 );
   // Both are possible, but less natural.
```

Define assignment only when its behaviour is non-trivial. (Not when you are just copying the fields.)

Return Value of Assignment

It is allowed that assignment returns anything. Also, **string** cannot be used in expressions, because we won't be able to call `clearstring()` on temporary variables.

An expression of form
`concatstring( readstring( ), readstring( ) )` will cause memory leaks.

Overload Resolution

The overloading rules for defined operators are the same as for usual functions (uniquely defined, best fit):

```
rational operator + ( const rational& , int );
rational operator + ( int, const rational& );


rational r = 1 + rational( 1,2 ); // Second.
r = r + 4; // First;
r = r + r; // Refuses.
```

Overload Resolution (2)

The compiler also tries to insert conversions. Every 1-argument constructor is a potential conversion.

```
rational operator + ( const rational& , const rational& );

r = 1 + 2; // int + is unique best fit.
   // r = rational(1 + 2 );

r = 1 + rational(1,2);
    // rational(1) + rational(1,2);
r = r + 1;
    // r + rational(1);
```

If you think that a unary constructor is not suitable as conversion (because the constructed object does not mean the same as its argument in the new type), then add the **explicit** keyword.

Initializer Lists

Consider class **stack**. If you want to build a stack with something on it, you have to write

```
stack s;
s. push(1); s. push(2); s. push(3);
```

This is ugly and inefficient. It breaks the rule that direct initialization should always be preferred over default initialization + reassignment.

Using initializer lists, one can write

```
stack s( { 1, 2, 3 } );
stack s = { 1, 2, 3 };
```

Initializer Lists

An initializer list is a simple datastructure that can hold a sequence of elements of unbounded length.

Initializer lists are automatically created from a list of form `{ t1, ... tn }`.

They should only be used for parameter passing, never for permanent storage!

Constructor with Initializer List

```
#include <initializer_list>

stack( std::initializer_list< double > init )
   : current_size( init. size( )),
     current_capacity( init. size( )),
     tab = new double [ init. size( ) ];
{
   for( auto p = init. begin( ); p != init. end( ); ++ p )
      write *p to the proper position in tab.
}
```

The same syntax `{ ... }` can be used to call the other, fixed length constructors.