# Overloading, Operators, Constness, Initializer Lists

## Const Modifier for Member Functions

Member functions of a class can be declared **const**. For accessors, one usually defines two versions, **const** and **non-const**.

```
class someclass {
   int* tab;
   int operator[] ( size_t i ) const { return tab[i]; }
   int& operator[] ( size_t i ) { return tab[i]; }
};
```

Compiler picks non-**const** version when possible, otherwise **const** version.

As a general rule, the compiler picks the closest fit.

## Preservation of Const

Compiler takes **const** very seriously:

- A **const reference** can never be copied into a **non-const** reference.
- Fields of a **const reference** are **const** reference by themselves.

Only exception are fields of form `X* tab;` where `tab` is part of the data structure. Such fields need to be controlled by access methods.

## Inlining

Inline functions are functions that are substituted away at compile time. They may make the code longer, and cannot be recursive.

- Calling a function requires some administration:

  Allocating some stack space, saving registers, saving return a address.

- Jumping to a function causes cache misses.

- If the optimizer cannot see the function definition, it cannot optimize. If you call it twice, it doesn't know if the result will be the same.

## Inlining (2)

```
double square( double x )
   { return x*x: }


double d = 4.0;
double s = square(d);
double s1 = square(d); // The same? Who knows.
```

With `inline`, compiler replaces by

```
double d = 4.0;
double s = d*d;  // Optimizer sees it is 16.
double s1 = d*d; // Optimizer sees it is 16.
```

## Inlining (3)

Functions that are defined in class definition, (nearly always in the **.h** file) are always inline.

For other functions, one can use the keyword **inline**.

There is no guarantee that compler will really inline, and it may also decide to inline other functions.

If you define a function in a **.h** file without **inline** keyword, the linker will complain.

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

## Why Macros are Bad

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

```
#define SQUARE(X)    ( X * X )
   // Problem with operator priorities.
   // X is evaluated twice.
```

Macro's ignore 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.

## Why Macros are Useless

- Polymorphism can be obtained better by templates:
  `SQUARE(4); SQUARE( 2.1 ); SQUARE( rational(1,4) );`
- If you worry about the cost of calling a function, it can be solved by using **inline**.
- Arrays should never[a] have fixed size, but shrink and grow with their contents. Use **std::vector** instead of array. Don't add fixed size constants to your own containers.
- Compile time constants can be defined using **constexpr**.

The only remaining use of macros is in include guards, and to switch off print statements or unfinished code. (`#if`).

[a] there is no never in programming

## Private Member Variables

I think that object-oriented programming means: Adding constructions to the language that make it easy to establish and preserve invariants, and to respect equivalences.

**invariant:** Some states should be not allowed.

> Add constructors. Make sure that the user cannot change fields of the class.

**equivalence:** Some states should be indistinguishable.

> Make sure that the user cannot see the fields of the class.

Both can be obtained with **private** fields. There is no direct way of making sure that the user can see, but not change fields.

Remember **rational** class. It has an invariant: (num,denum) is always normalized.

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

Unfortunately, some user may still decide to ignore our delicate invariants, and write

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

(In addition, it uses unnecessary default initialization.)

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

## Private Fields

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

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

   ... a few methods that can acces num and denum.
};
```

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 (no equivalences).

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

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

Using getters instead of the fields is free of cost, because the functions are inlined.

## This

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

Very very unfortunately, **this** is always a pointer. It would be much nicer if **this** 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 )` )

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

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

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

In older (pre 2011) versions of $C^{++}$, you will find `( )` for initialization. It does the same, but it does not detect narrowing.

## Defining User Operators

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

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

## Simple Operators

Simple operators `+`, `-`, `*`, `/`, `%`, `&&`, `||`, `^^`, `<<`, `>>` can be defined.

One can also define `++`,`--`.

Most operators can be defined 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.

## 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.

## Overload Resolution

Overloading means that a function (or method) with one name, has different definitions. We have seen quite some examples:

1. Indexing [ ] can have two definitions **const** and **non-const**. This applies to all member functions.
2. The user can add definitions to existing operators:
   `+, -, |, +=, -=, .`
3. The user can define a function multiple times with different argument types, e.g. `f(int), f(double), f(char).`
   For example, `<<` has many definitions.

When a program uses a function with many definitions, the compiler has to decide which one to use. This is called overload resolution.

## Candidates, Overload Resolution

If you call a function, the compiler first decides which functions fit in principle.

The compiler considers implicit type conversions, user defined conversions, user defined constructors, copy constructors, and CV-qualifiers.

The resulting functions are called the candidates.

## Overload Resolution

In case, there is more than one candidate, the compiler has to choose one.

This choice is based on conversion levels.

In general, less conversion is preferred over more conversion, but lvalue transformations (loads and copy constructors) are ignored in counting conversions.

## Conversion Levels

$C^{++}$ distinguishes levels of conversions:

**Level 1A** The conversion from $T_1$ to $T_2$ has level 1A if $T_1 = T_2$ (exact fit) or it involves conversion from arrays or functions to pointers (which is unavoidable because nothing else can be done with them).

**Level 1B** The level from $T_1$ to $T_2$ is level 1B if consists of a level 1A conversion, followed by possible insertions of **const**.

**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.)

## Multi Argument Functions

Assume that function $f$ is applied on arguments $a_1, \ldots, a_n$ :

Assume that

$$f_1(t_{1,1}, \ldots, t_{1,n})$$

$$\ldots$$

$$f_m(t_{m,1}, \ldots, t_{m,n})$$

are the candidates.

If there is no candidate ($n = 0$), then 'No matching definition found'.

## Multi Argument Functions (2)

Otherwise, find a candidate $f_i$, s.t. for all $f_{i'}$ with $i' \neq i$, we have:

1. For every argument position $j$, the level of the conversion from $a_j$ into $t_{i,j}$ is not worse (the same or better) than the level of the conversion from $a_j$ into $t_{i',j}$.
2. There is at least one argument position $j$, where the level of the conversion from $a_j$ into $a_{i,j}$ is really better than the level of the conversion from $a_j$ into $t_{i',j}$.

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

The rules work so well that you almost 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.
```

## Argument Dependent Look Up (ADL)

In expression `f(t1,...tn)`,

1. `f` is looked up in current context,
2. also in context of types of `t1,...tn`.

Don't underestimate the importance of this rule. It is important when you use namespaces.

It is essential for usefulness of operator overloading.

## Type Checking is Top Down Only

In $C^{++}$, type checking is always top down (with the exception of **initializer_list**).

This means that expected result plays no role in determining the candidates.

```
int f( int x );
string f( int x );
...
int i = 4;
string s = f(i);
```

Theoretically, compiler could see that only second **f** fits, but it won't. It will tell that both **f** are candidates, and that the call is ambiguous.

## 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 with 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( double d: init )
      write d to the proper position in tab.
}
```

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

## Overloading Assignment Operators

Assignment operators `+=, -=, *=` can be defined. They are not defined by default. Define them only when they have meaningful definitions that deserve to called `'+','-'`, etc.

If both `+=,+` are defined, the meaning of `+=` should be `x=x+a;`

It is fine to define one in terms of the other:

```
void operator += ( rational& r1, const rational& r2 )
{ r1 = r1 + r2; }


rational operator + ( rational r1, const rational& r2 )
{ r1 += r2; return r1; }
```

Sometimes it is more efficient to define them separately.

## Default Operators

If you don't declare a copy constructor, the compiler will define a default copy constructor that copies the members, when this is possible.

If you don't declare an assignment operator, the compiler defines a default assignment (that assigns the members).

If you don't declare a destructor, the compiler defines a default destructor that destroys the members.

The compiler declares a default (0-argument) constructor, only when you define no other constructors.

## Default Operators

Don't declare an operator (copy constructor, assignment or destructor) when the default works. You may write

```
rational( const rational& r ) = default;
rational& operator = ( const rational& r ) = default;
   // etc.
```

If you don't want a copy constructor or assignment (sometimes this is useful), you can write:

```
rational( const rational& r ) = delete;
rational& operator = ( const rational& r ) = delete;
```