Simulation and scientific computing:
C++ Tutorial

Britta Heubeck, Klaus Iglberger

Chair for System Simulation
University Erlangen/Nuremberg

Outline

- C++ Overview
- C++ Classes
- Compiler Generated Functions
- C++ Operators
- C++ Templates
- The Standard Template Library (STL)
- Examples
- References

C++: A First Overwiew

Question: Is C++ an object-oriented language?

C-style programming

Object-oriented C++

Template C++

The Standard Template Library (STL)

No (at least not only). C++ should be seen as a federation of programming languages ⇨ Multiparadigm programming language

First Differences between C and C++

- **The creation of an object is an expression. Therefore a new object can be created everywhere where an expression is expected!**

```
for( int i=0; i<N; ++i ) ...

...

if( int a = GetRandomInteger() ) {
    int b = 2 * a;
    ...
}
```

- **C++ has the new data type `bool`. A `bool` object can only have either the value `true` or `false`.**

```
bool isValid( false );
bool mayBe( true );
```

The new/delete Operators

Dynamic memory allocation should not anymore be done via the `malloc` function family but with the `new` operator family:

```
int* a = new int();    //Dynamically creates an integer
...
delete a;

int* a = new int[5];   //Dynamically creates an integer array
...
delete[] a;
```

Question: What is the following code? A static or dynamic allocation?

```
int a[] = { 1, 2, 3, 4, 5 };
```

The following code follows a noble guideline, but is unfortunately useless:

```
int* p( 0 );
if( p = new int[1000000] ) {
    ...
}
else {
    ...  //Printing an error message that the memory allocation
         // has failed!
}
```

If the new operator really fails to allocate the requested memory (a very rare case), a `std::bad_alloc` exception is thrown. Therefore the else branch will never be executed (even in the case of a failed allocation).

C++ References

A C++ reference is an alias to an existing object, e.g.

```
int i( 1 );      //New integer object with value 1
int& ri = i;     //'ri' is an alias for 'i'
i = 3;           //Assigns 3 to 'i' directly
ri = 2;          //Assigns 2 to 'i' via 'ri'
```

In C++, references should be used in most cases, where pointers were used in C, for example as function parameters, e.g. instead of

```
void f( int* i ) { *i = 4; }
```

use

```
void f( int& i ) { i = 4; }
```

References have to be initialized immediately and refer to the same object until they go out of scope!

```
int& ri;            //Error! Uninitialized reference!
```

There are no null-references! Any reference to a null-value will result in undefined bahavior!!

```
int* GetInteger();

...

int& i = *GetInteger();

if( i == 0 ) {      //Subtle error! Undefined bahavior!
    ...
}
```

A Step Backwards: A Short C Repetition

Question: Is C an object-oriented language?

```
//--Complex.h-------------------------------------------

struct Complex {
  double real, imaginary;
};

void Add( const struct Complex* a, const struct  Complex* b,
          struct Complex* res )
{
  res->real = a->real + b->real;
  res->imaginary = a->imaginery + b->imaginery;
}

//--Complex.c-------------------------------------------

Complex a, b, c;
...
Add( a, b, c );
```

Definition of a class (C/C++ standard):

> **"A class describes a set of data, along with the functions that operate on that data."**

Question: Is C an object-oriented language?

- **C offers data sets (structs)**
- **C functions can work on this data**
- **Structure and functions are provided together (interface)**

⇨ **Object-oriented programming is also possible with C (but much less comfortable)**

C++ Classes

```
//--Headerfile----------------------------------------

class A {};    //Complete definition of class/type A

class B {      //Complete definition of class/type B
  void f();
};

struct C {     //Complete definition of class/type C
  void f();
};

//--Sourcefile----------------------------------------

A a;      //Correct, creates an A object

B b;      //Correct, creates a B object
b.f();    //Error! Access to f is private!

C c;      //Correct, creates a C object
c.f();    //Correct, access to f is public!
```

C++ Classes: An Overview

```
class A;                        //Declaration of class A

...

class A {                       //Definition of class A
  public:                       //Starts a public part of the class
    void f();                   //Declaration of function f
    void f( int );              //Overloading function f
    int g( int );               //Declaration of function g
    int g( int ) const;         //Overloading function g
    void h() const;             //Declaration of function h
  private:                      //Starts a private part of the class
    int i_;                     //Definition of two member variables, one
    double d_;                  // of integer type, the other of double type
};

...

void A::f() {                   //Definition of function f
    ...
}
```

Member functions

Member functions are called with the . operator (static objects) or the -> operator (dynamic objects).

```
A a1;                          //Creates a static object of type A
const A* a2 = new A();         //Creates a dynamic object of type 'const A'

a1.f();                        //Calls the function 'void f()'
a1.f( 1 );                     //Calls the function 'void f(int)'

a1.g( 2 );                     //Calls the function 'int g(int)'
a2->g( 2 );                    //Calls the function 'int g(int) const'

a1.h();                        //Error! No non-const function 'h' defined!
a2->h();                       //Calls the function 'void h() const'

delete a2;                     //Destroys the object 'a2' and frees the memory
```

Question: What is the difference between a standard C function and a C++ member function?

Answer: There is no difference!

```
class A {
  public:
    void f( int i );
    void f( int i ) const;
};
```

is the same as

```
class A { ... };
f( A* const this, int i );
f( const A* const this, int i );
```

The `this` parameter is an implicit parameter in every member function. You may not declare this parameter explicitly!

Name Lookup and Overload resolution

Question: Would the following code compile correctly without any warning or error message?

```
class A {
    public:
        void f( int );
    private:
        void f( double );
};

...

A a;
float value;

a.f( value );  //Calling f, but which function?
```

Calling a function is performed in three steps: name lookup, overload resolution and checking the access permission.

Compiler generated functions

Question: Can I really create an empty class?

```
class A {};    //Is this really an empty class??
```

Answer: Hardly, because of the compiler generated default functions.

```
class A {
  public:
    A() {}                          //Default constructor
    ~A() {}                         //Default destructor
    A( const A& a ) {}              //Default copy constructor
    A& operator=( const A& a ) {}   //Default copy assignment operator
};
```

At least these four functions are automatically created if you are not explicitly declaring them.

The Constructor

The constructor is called, when a new object is created. Its task is to initialize the object and to perform necessary setup operations.

```
class A {                       //Definition of class A
  public:
    A();                        //Default constructor
    A( const char* c );         //Overloading the constructor
    ...
  private:
    uint len_;                  //The two member variables of class A, one
    std::string name_;          // of type 'uint', the other a 'std::string'
};

A::A():len_(0),name_()          //Definition of the default ctor
{                               // with initializer list
  ...
}
```

- **The compiler only generates a default constructor, if you are not declaring a constructor yourself!**
- **The choice which constructor to use is solved with the standard overload resolution:**

```
A a;                    //Default constructor is called
A b( "Message" );       //Second constructor is called

A c();          //Error! This is a function declaration!!

int f();        //As comparison: another function declaration.
```

- **The order of initialization is the same as in the class definition, not as in the initializer list!!**

```
A::A( const char* c )
:name_( c ),len_( name_.size() )    //Severe, subtle error!
{}
```

The Destructor

The destructor is called, when an object is destroyed. Its task is to clean up, to free resources and to perform necessary finalization operations. The destructor works in reverse order of the constructor.

```
class A {                  //Definition of class A
  public:
    A();                   //Default constructor
    ~A();                  //Default destructor
    ...
  private:
    uint len_;
    std::string name_;
};

A::~A()                    //Definition of the default destructor
{
    ...
}
```

- **The compiler only generates a default destructor, if you are not declaring the destructor yourself! You have to declare and define your own destructor, if you have dynamically allocated resources in your class!**

```
class B {                 //Definition of class A
  public:
    B();                  //Default constructor
    ~B();                 //Default destructor
    ...
  private:
    int* p_;              //Array of a number of integers
};

B::~B()
{
    delete[] p_;
}
```

The Copy Constructor

The copy constructor is called, if a new object is constructed as a copy of another object of the same type.

```
class A {                  //Definition of class A
  public:
    ...
    A( const A& a );  //Declaration of the copy constructor
    ...
  private:
    uint len_;
    std::string name_;
};

//Definition of the copy constructor with initializer list
A::A( const A& a ):len_(a.len_),name_(a.name_)
{
    ...
}
```

The Copy Assignment Operator

The copy assignment operator is called, if one object is copied into another object of the same type.

```
class A {              //Definition of class A
  public:
    ...
    A& operator=( const A& a );  //Declaration of the copy
    ...                          // assignment operator
};

A& A::operator=( const A& a )  //Definition of the copy
{                              // assignment operator
    if( this == &a )           //Self-protection against
      return *this;            // self-assignment
    ...
    return *this;
}
```

- **The compiler only generates a default copy assignment operator, if you are not declaring one by yourself! You have to (should?) declare and define your own copy assignment operator, if your class has dynamically allocated resources.**

```
class B {
    ...
    int* p_;
    ...
};
```

- **The return value should be a reference to the object itself. This enables the reuse of the object, e.g.**

```
A a1, a2, a3;     //Three times the default constructor
a1 = a2 = a3;     //Two times the copy assignment operator
```

- **If a protection against self-assignment is necessary depends on the class itself!**

Copy control

Questions:

- When do I need a destructor?
- When do I need a copy constructor?
- When do I need a copy assignment operator?

Answer:

In most cases, if you need one of these functions you also need the other two. You need these functions (in most cases), if you have dynamically allocated resources, e.g.

```
class B {
    ...
    int* p_;
    ...
};
```

Initialization or Assignment?

```
A a, b;

A c( a );    //Direct initialization via the copy constructor

b = a;       //Copy assignment

A d = a;     //Copy assignment to a default constructed object

int e[5];    //Default initialization of each array element

int f[] = { 1, 2, 3, 4, 5 };  //Copy initialization of each element
```

- **Prefer direct initialization over assignment (even for built-in types). This might save you half the number of operations!**

C++ Operators

- **Operators that can be overloaded:**

```
+       -       *       /       %       ^
&       |       ~       !       ,       =
<       >       <=      >=      ++      --
<<      >>      ==      !=      &&      ||
+=      -=      /=      %=      ^=      &=
|=      *=      <<=     >>=     []      ()
->      ->*     new     new[]   delete  delete[]
```

- **Operators that cannot be overloaded:**

```
::          .*          .          ?:
```

- **Use operator overloading wisely and judiciously!**

Example of a 3D-vector:

```
class Vector {
  public:
    Vector() { v_[0]=0.0; v_[1]=0.0; v_[2]=0.0; }
    Vector( const Vector& v ) { v_[0]=v[0]; v_[1]=v[1]; v_[2]=v[2]; }

    Vector& operator=( double set );

    double& operator[]( uint index );

    Vector& operator+=( const Vector& vec );

    const Vector operator+( const Vector& vec ) const;

    const Vector operator*( double scalar ) const;

    ...

  private:
    double v_[3];
};
```

```
//Definition of the assignment operator
// (does NOT replace the copy assignment operator)
Vector& Vector::operator=( double set ) {
    v_[0] = set; v_[1] = set; v_[2] = set;
    return *this;
}


//Definition of the subscript operator
double& Vector::operator[]( uint index ) {
    return v_[index];
}


//Definition of the addition assignment operator
Vector& Vector::operator+=( const Vector& vec ) {
    v_[0] += vec[0];
    v_[1] += vec[1];
    v_[2] += vec[2];
    return *this;
}
```

```
//Definition of the addition operator
const Vector Vector::operator+( const Vector& vec ) const {
    Vector tmp( *this );
    tmp += vec;
    return tmp;
}


//Alternative definition of the addition operator with a
// computational constructor
const Vector Vector::operator+( const Vector& vec ) const {
    return Vector( v_[0]+vec.v_[0], v_[1]+vec.v_[1], v_[2]+vec.v_[2] );
}


//Definition of the multiplication operator (again
// with a computational constructor)
const Vector Vector::operator*( double mul ) const {
    return Vector( mul*v_[0], mul*v_[1], mul*v_[2] );
}
```

- **Best practice: Define all =, op=, [] and () operators within the class and make all other operators global operators, e.g.**

```
const Vector operator+( const Vector& v1, const Vector& v2 ) {
    return Vector( v1[0]+v2[0], v1[1]+v2[1], v1[2]+v2[2] );
}

const Vector operator*( const Vector& v, double mul ) {
    return Vector( mul*v[0], mul*v[1], mul*v[2] );
}

const Vector operator*( double mul, const Vector& v ) {
    return Vector( mul*v[0], mul*v[1], mul*v[2] );
}

...

Vector a, b, c;
b = a * 2.0;
c = 3.0 * a;
```

Increment and Decrement Operators

Let's assume we have the following class that needs increment and decrement operators:

```
class Integer {
  public:

    ...

    Integer& operator++();          //Prefix-increment operator

    Integer operator++( int );      //Postfix-increment operator

    Integer& operator--();          //Prefix-decrement operator

    Integer operator--( int );      //Postfix-decrement operator

    ...

  private:
    int i_;
};
```

```
Integer& Integer::operator++() {
    ++i_;
    return *this;
}

Integer Integer::operator++( int ) {
    Integer tmp( *this );
    ++*this;
    return tmp;
}

Integer& Integer::operator--() {
    --i;
    return *this;
}

Integer Integer::operator--( int ) {
    Integer tmp( *this );
    --*this;
    return tmp;
}
```

- **Prefer to use prefix operators instead of postfix operators, even in case of built-in types:**

```
...
for( int i=0; i<N; ++i ) {
  ...
}
...
```

- **Implement the postfix operators in terms of the prefix operators**

C++ Templates

There are two different kinds of templates: function templates and class templates.

- **Function templates**

```
//Definition of a generic comparison function
template< typename T >
int compare( const T& t1, const T& t2 ) {
    ...
}


//Special treatment of char arrays
int compare( const char* c1, const char* c2 ) {
    ...
}
```

Class templates

```
//Generic implementation of a N-dimensional vector
template< typename T, uint N >
class Vector {
  public:

    ...

    T& operator[]( uint index );

    ...

  private:
    T v_[N];
};

//Definition of the subscript operator
template< typename T, uint N >
T& Vector<T,N>::operator[]( uint index ) {
    return v_[index];
}
```

Argument Deduction and Instantiation

- **In case of a function template, the template arguments can be deduced from the actual function parameters, e.g.**

```
int i( 2 ), j( 3 );
...
if( compare( i, j ) ) { ... } //Use of the compare function template
```

In the case of class templates, the template arguments have to be explicitly specified, e.g.

```
Vector<int,2> v;                    //Use of the Vector class template
```

- **Templates get instantiated at the moment they are used. In case of a class template, only those functions are instantiated that are used.**

```
Vector<double,3> v;     //Instantiation of 3D-double-vector
double d = v[0];        //Instantiation of the subscript operator
```

Template compilation

If a template (function or class) is used, the entire definition has to be visible in order to create a specific instance of the template. Two ways seem to be most practical:

- **Put the entire template definition in one header file**
- **Create a header and a source file as usual, but include the source file at the end of the header file (inclusion model)**

```
//--Vector.h-----------------------------------------------

   ...

   //Including the source file at the end of the header file
   #include "Vector.cpp"


//--Vector.cpp---------------------------------------------
   ...
```

The Standard Template Library (STL)

- **The `std::ostream` and `std::istream` classes**

  I/O operations in C++ are handled by the classes based on `std::istream` and `std::ostream`.

Standard output stream	`std::cout`
Standard input stream	`std::cin`
Standard error output	`std::cerr`

The strength of C++ streams lies in its capability of error control and type safety.

```
int i;
sprintf( buffer, "%4ld", i );   //Is this safe code??
```

Example for the controlled input of two positive integer values:

```
#include <iostream>

...

int i, j;

std::cout << "Checked input of two positive integers:\n";

if( !(std::cin >> i >> j) || i <= 0 || j <= 0  ) {
    std::cerr << "Invalid integer values!" << std::endl;
}
else {
    std::cout << "No error during integer input!\n";
    std::cout << " i=" << i << " , j=" << j << "\n";

    ...  //Safe use of the two positive integer values
}
```

Reading input from a file is handled similarly:

```
#include <fstream>

...

int i, j;
std::ifstream in( "Data.in", std::ifstream::in );
std::ofstream out( "Data.out", std::ofstream::out |
                               std::ofstream::trunc );

if( !(in >> i >> j) || i <= 0 || j <= 0 ) {
    out << "Invalid integer values!" << std::endl;
}
else {
    out << "No error during integer input!\n";
    out << " i=" << i << " , j=" << j << "\n";
}

...

in.close();
out.close();
```

A type safe conversion to characters can be achieved by using the `std::stringstream` classes.

```
#include <sstream>

std::string datastring;

...

int i, j;
std::istringstream in( datastring );
std::ostringstream out;

if( !(in >> i >> j) || i <= 0 || j <= 0 ) {
    out << "Invalid integer values!" << std::endl;
}
else {
    out << "No error during integer input!\n";
    out << " i=" << i << " , j=" << j << "\n";
}

...
```

Classes can be given an input and/or output operator in order to facilitate the input/output of a class.

```
//Definition of class A
class A {
    ...
};

//Definition of the output operator for class A
std::ostream& operator<<( std::ostream& os, const A& a ) {
    ...  //Writing all necessary information to the output stream
}

//Use of the output operator
#include <iostream>

A a;

...

std::cout << " Object a:\n" << a << std::endl;
```

The `std::string` class

The `std::string` class is the C++ way to handle character arrays. In contrast to character arrays, this class contains a memory control to avoid buffer overruns etc.

```
#include <iostream>
#include <string>

...

std::string hello( "Hello" );  //Direct initialization
std::string world = "World";   //Default ctor and copy assignment

std::cout << "Number of characters in 'hello' = "
          << hello.size() << "\n";
std::cout << hello + world << std::endl;  //Displays "HelloWorld"

std::string message( 50, " " );
if( !(std::cin >> message) ) {
    ...
}
```

The `std::vector` class

The `std::vector` is one of the most useful classes of the STL. The `std::vector` is the C++ array for all kinds of types.

```
#include <iostream>
#include <vector>

std::vector<int> intVector;

for( int i=0; i<10; ++i ) {
    intVector.push_back( i );
}

std::cout << "Number of elements in the vector: "
          << intVector.size() << "\n";

//Printing all elements
for( std::vector<int>::size_type i=0; i<intVector.size(); ++i ) {
    std::cout << "Element " << i << " " << intVector[i] << "\n";
}
```

```
#include <vector>

...

typedef std::vector<std::string> PhoneBook;
PhoneBook phonebook;

...   //'phoneBook' is filled with names

//Erasing a specific element from the vector
for( PhoneBook::iterator it = phoneBook.begin();
                          it != phoneBook.end(); ++it ) {
    if( *it == "Meier" ) {
        phoneBook.erase( it );
        break;
    }
}

...
```

The using Keyword

- **The `using` declaration**

  A `using` declaration creates a local synonym for a name actually declared in another namespace.

```
#include <vector>
using std::vector;

vector<int> intVector;
```

- **The `using` directive**

  A `using` directive allows all names in another namespace to be used in the scope of the using directive.

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> intVector;

cout << "Testing the using directive!" << endl;
```

A Word about Inlining

- Inlining is a request to the compiler to place the content of a function at the location of the function call to replace the function call.
- An inlined function can increase performance because of the omitted function call.
- An inlined function can reduce performance because of a larger amount of instructions.
- Best practice: Inline all single expression functions and gather profiling information for all other functions.
- All functions defined inside the class body are automatically declared inline functions. All other functions have to be explicitly declared inline.
- Inline functions have to be declared and defined in a header file in order to make the complete definition available for the inlining.

```
//--A.h---------------------------------------------------
class A {
  public:
    void f() { ... }    //Automatically declared as inline function

    inline void g();    //Not automatically, but
                        // explicitly declared inline

    void h();           //Not declared inline
};

//Either declare the function inline inside the class or declare
// the function inline with the definition (no need to do both)
inline void A::g() {
    ...
}

//--A.cpp-------------------------------------------------

//Definition in the source file, no inlining possible
void A::h() {
    ...
}
```

Example: class Complex

The following code shows the implementation of a class `Complex`. Your task is to:

- **Find all errors/flaws in the implementation of this class!**
- **Judge the quality of this class. Would you use it in your own code?**

This class is by far not complete. It is supposed to demonstrate some guidelines in the development of C++ classes. So the lack of several functions should not be counted as error/flaw.

```
class Complex {
  public:

    Complex( double real, double imaginary = 0.0 )
    :_real(real),_imaginary(imaginary)
    {}

    void operator+( Complex other )
    {
      _real = _real + other._real;
      _imaginary = _imaginary + other._imaginary;
    }

    void operator<<( ostream& os )
    {
      os << "(" << _real << "," << _imaginary << ")";
    }

    ...
```

```
    ...

    Complex operator++()
    {
      ++_real;
      return *this;
    }

    Complex operator++( int )
    {
      Complex temp = *this;
      ++_real;
      return temp;
    }

  private:
    double _real, _imaginary;
};
```

Example class: Possible improvements

1.) Use std::complex instead of implementing your own class

Reuse code as often as possible, especially the code from the STL.

```
#include <complex>
std::complex a;
...
```

This code has been developed by professionals and thoroughly tested. If possible, reuse this code. It will save you much time and many troubles!

2.) Constructor allows implicit conversion

Better approach: use an explicit constructor and provide the necessary operators! (also avoid conversions)

```
Complex( double real, double imaginary = 0.0 )
:_real(real),_imaginary(imaginary)
{}
...
Complex a( 1.0, 2.0 );
Complex b = 1.0 + a;
```

⇨ **Possible temporaries and possible ambiguities!**

Better use explicit constructors:

```
explicit Complex( double real, double imaginary = 0.0 )
:_real(real),_imaginary(imaginary)
{}
```

3.) Addition operator: Call-by-value parameter in operator+

```
void operator+( Complex other )
{
  _real = _real + other._real;
  _imaginary = _imaginary + other._imaginary;
}
```

Better use call-by-reference-to-const to avoid a temporary value

```
void operator+( const Complex& other )
{
  _real = _real + other._real;
  _imaginary = _imaginary + other._imaginary;
}
```

4.) Addition operator: Prefer `a op= b` instead of `a = a op b`

```
void operator+( const Complex& other )
{
  _real = _real + other._real;
  _imaginary = _imaginary + other._imaginary;
}
```

should better be implemented as

```
void operator+( const Complex& other )
{
  _real += other._real;
  _imaginary += other._imaginary;
}
```

This is only a trivial change for built-in types, but could be much more efficient for class types.

5.) Addition operator: Prefer non-member functions

```
class Complex {
  ...
  void operator+( const Complex& other )
  {
    _real += other._real;
    _imaginary += other._imaginary;
  }
  ...
};
```

Using a member function, the first argument has to be a `Complex` object. In order to use the `Complex` object naturally, we should be able to do both the following operations:

```
Complex a;
Complex b = a + 1.0;
Complex c = 1.0 + a;
```

Better use non-member functions if possible.

6.) Addition operator: should not change the object itself!!

```
void operator+( const Complex& other )
{
  _real += other._real;
  _imaginary += other._imaginary;
}
```

Best approach: provide both `operator+` and `operator+=` and implement `operator+` in terms of `operator+=`

```
Complex& operator+=( const Complex& other )
{
  _real += other._real;
  _imaginary += other._imaginary;
  return *this;
}

Complex operator+( const Complex& c1, const Complex& c2 )
{
  Complex tmp( c1 );
  c1 += c2;
  return tmp;
}
```

7.) Output operator: Should not (cannot?) be a member function

```
void operator<<( ostream& os )
{
  os << "(" << _real << "," << _imaginary << ")";
}
```

Defining the operator like this, no natural use is possible:

```
Complex a;
std::cout << a;    //Error! Will not compile!
a << std::cout;    //Correct, but highly unintuitive!
```

Better define a member function `Print` and define the `operator<<` as a global function using the `Print` member function

```
ostream& operator<<( ostream& os, const Complex& c )
{
  Print( os );
  return os;
}
```

8.) Prefix-increment operator: wrong return type

```
Complex operator++()
{
  ++_real;
  return *this;
}
```

Increment operator should return a reference to the object itself

```
Complex& operator++()
{
  ++_real;
  return *this;
}
```

9.) Postfix-increment operator: wrong return type

```
Complex operator++( int )
{
  Complex temp = *this;
  ++_real;
  return temp;
}
```

Increment operator should return a const object and should be implemented in terms of the prefix-increment operator.

```
const Complex operator++( int )
{
  Complex temp( *this );
  ++*this;
  return temp;
}
```

10.) General problem: avoid reserved names

```
class Complex {
  ...
  private:
    double _real, _imaginary;
};
```

Avoid names with leading underscore (especially names with leading underscore and following capital letter) and names with two underscores (e.g. _AvoidThis, Avoid__This).

```
class Complex {
  ...
  private:
    double real_, imaginary_;
};
```

Example: class Grid

Implementation of a 2D grid class for maximum flexibility and performance

```
template< typename T >
class Grid {
  public:
    Grid()
    :xsize_(0),ysize_(0),grid_(0)
    {}

    Grid( uint xsize, uint ysize )
    :xsize_(xsize),ysize_(ysize),grid_(new T[xsize_*ysize_])
    {}

    ~Grid() { delete[] grid_; }

    Create( uint xsize, uint ysize ) {
      xsize_ = xsize;
      ysize_ = ysize;
      grid_ = new T[xsize_*ysize_];
    }
```

```
    Resize( uint xsize, uint ysize ) {
      xsize_ = xsize;
      ysize_ = ysize;
      delete[] grid_;
      grid_ = new T[xsize_*ysize_];
    }

    Delete() {
      xsize_ = 0;
      ysize_ = 0;
      delete[] grid_;
      grid_ = 0;
    }

    uint XSize() const { return xsize_; }
    uint YSize() const { return ysize_; }

    T& operator( uint x, uint y ) { return grid_[ y*xsize_ + x ]; }
    const T& operator( uint x, uint y ) const { return grid_[ y*xsize_ + x ]; }

  private:
    uint xsize_, ysize_;
    T* grid_;
};
```

General remarks

1.) Avoid the use of macros and pseudo functions

```
#define X 0
#define Y 1
#define Z 2

#define MAGIC_NUMBER 10.23

#define SQR(a) ((a)*(a))
#define MIN(a,b) ((a<b)?(a):(b))
#define MAX(a,b) ((a>b)?(a):(b))
```

Better use const values, enums or plain functions:

```
enum { X=0, Y=1, Z=2 };

const double magicNumber( 10.23 );
```

```
template< typename T >
inline T Sqr( const T& a )
{
  return a * a;
}

template< typename T >
inline T Min( const T& a, const T& b )
{
  return ( a < b )?( a ):( b );
}

template< typename T >
inline T Max( const T& a, const T& b )
{
  return ( a > b )?( a ):( b );
}
```

2.) Use comments wisely and judiciously

```
...
while( ... ) {
  if( a < b ) {
    c = a + b;  // Adding a and b
  } // if
  else {
    c = a - b;  // Subtracting a and b
  } // else
} // while
...
```

Don't comment on what the code is documenting itself, but also don't save comment when they are necessary for the user of the code.

```
VeryVeryComplexFunction( ComplexClass& complex )
{
  ...
}
```

3.) The use of NULL

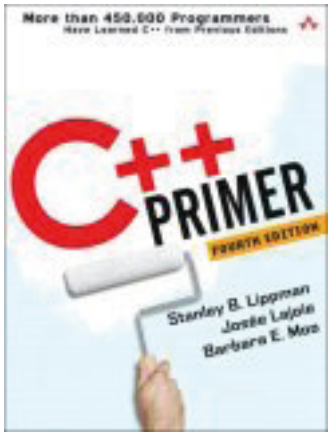
Avoid to use the macro NULL. NULL is nothing else than either `(char*)0`, `(void*)0`, or `0`. Just use `0` in all cases.

4.) Possible name convention

- variables, parameters, etc.: likeThis
- member variables: likeThis_
- Functions, member functions: LikeThis
- Macros: LIKE_THIS

This name convention is supposed to be a suggestion (although a good one). More important than to use this convention is the consistent use of any convention in your code!

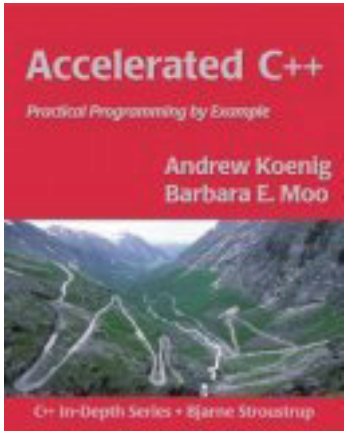
References



- **Stanley B. Lippman, Josée Lajoie**

C++ Primer

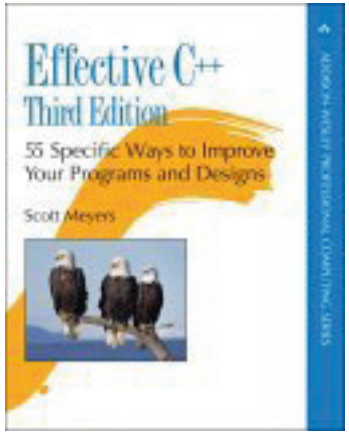
Addison Wesley, 2005
ISBN: 0201721481



- **Andrew König, Barbara E. Moo**

Accelerated C++

Addison Wesley, 2000
ISBN: 020170353X



- **Scott Meyers**

Effective C++

Addison Wesley, 2005
ISBN: 0321334876