High Level Parallel Programming

C++ Addendum

Alessandro Savino and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Const type qualifier

- The keyword const prevent objects from being changed
- The keyword const is very powerful as it allows an advance code control, with fine tuning and optimization
  - Const member functions
  - Const parameters
  - Const references

Extras at:
https://isocpp.org/wiki/faq/const-correctness

Const member functions

- Constant member functions are
  - Member function that are guarantees will not modify the object or call any non-const member functions (as they may modify the object)

```
class Rectangle {
  public:
    ...
    double getW() const;
    double getL() const;
    double getArea() const;
    double getPerimeter() const;
```

Getters that will not modify the values

Const member functions

- We need to repeat the const keyword in the implementation
  - There will not be any change in the private data, otherwise the compiler will produce an error

```
double Rectangle::getArea() const {
    return m_width * m_length;
}
```

Const parameters

- You can set as const any parameter of member function
  - It means you expect not to modify it

```
class Rectangle {
  public:
    Rectangle();
    Rectangle(const double w, const double l);
  ...
```

If the parameter is passed by value, that does not have much sense (the original value is not change anyway)

Const parameters

But const can be used for by-reference parameters as well

- Const references are meant to replace passing the parameter by value to avoid to copy it
  - Still you must guarantee that no further modification will be done!

```
class Rectangle {
  public:
    void setW(const double &w);
    void setL(const double &l);
  ...
```

Thus you use a parameter by-reference, to avoid any data-duplication, but you can only read it

```
void Rectangle::setW(const double &w) {
  m_width = w;
}
void Rectangle::setL(const double &l) {
  m_length = l;
}
```

Const parameters

- Use
  - Call-by-value for very small objects
  - Call-by-const-reference for large objects
  - Call-by-reference only when you have to return a result rather than modify an object

Examples

```
class Image {
  /* objects are potentially huge */
  ...
};
```

```
void f(Image i);
```

By-value
Oops … this could be slow and occupy a lot of memory

```
f(my_image);
```

```
void f(Image& i);
```

By-reference
No copy, but f() can modify the image

```
f(my_image);
```

```
void f(const Image&);
```

By const-reference
No copy, and f() cannot modify (mess with) the image

```
f(my_image);
```

Friend declarations

- A class body can contain friend declarations
  - A friend declaration grants a function or another class access to the private and protected members of the class which contains the declaration
- Notes
  - Friendship is non-transitive and cannot be inherited
  - Access specifiers have no influence on friend declarations
    - They can appear in private or public sections

Friend declarations

```
friend function_declaration;
friend function_definition;
friend class_specifier;
```

- Syntax
  - Declares a function as a friend of the class
  - Defines a non-member function and declares it as a friend of the class
  - Declares another class as a friend of this class

Example

```
class A {
  int a;
  friend class B;
  friend void foo(A&);
};
```

Friend class

Friend method

```
void foo(A& a) {
    a.a=42;
}
```

Even if a is a private member foo can access it

```
class B {
  friend class C;
  void foo(A& a) {
    a.a = 42;
  }
};
```

Class B is a friend of A thus the definition of foo is correct

```
class C {
  void foo(A& a) {
    a.a = 42;
  }
};
```

Thee friend attribute is not transitive
This definition of foo is invalid

Functions overloading

- It is possible to have multiple definitions for the same function in the same scope
  - The definitions of the function **must** differ from each other for
    - The types of its arguments
    - The number of its arguments
  - You **cannot** overload function declarations that differ only by return type
- The idea is the same applied constructors
  - It is possible to define multiple constructors differentiating them with the argument list

Example

```
class Rectangle {
public:
  Rectangle();
  Rectangle(const double &w, const double &l);
  Rectangle(const double &w_l);
  ~Rectangle() {};
...
  void setW(const double &w);
  void setW(const int &w);
  void setL(const double &l);
  void setL(const int &l);
...
}
```

Different constructors

Different methods

The compiler decides

```
Rectangle r5, r6;
r5.setW(2);
r5.setW(4.5);
...
```

Operators overloading

- What is an operator?

Operators	Symbol
Assignment	=
Arithmetic	* - * / %
Coumpund assignment	+= -= *= /= %= >>= <<= &= ^& |=
Increment and decrement	++ --
Relational and comparison	== != > < >= <=
Logical	! && ||
Conditional	?
Comma	,
Bitwise	& | ^ << >>

Operators overloading

- What about using it for my own classes …
  - Would it make sense writing the following?

```
int main() {
  ...
  Rectangle r4(r3);
  ...
  Rectangle r5, r6;
  ...
  r6 = r5 + r4;
}
```

Rectangle = Rectangle

Rectangle == Rectangle

Rectangle != Rectangle

++Rectangle
(prefix)

Rectangle + Rectangle

Rectangle++
(postfix)

Example

```
class Rectangle {
public:
  ...
  Rectangle operator+(const Rectangle &to_be_added);
  void operator=(const Rectangle &to_be_assigned);
  const Rectangle& operator++(); // prefix
  const Rectangle operator++( int ); // postfix
  bool operator==(const Rectangle &to_be_compared);
  bool operator!=(const Rectangle &to_be_compared);
  ...
}
```

You customize these operators for your object

We are overloading the operators as they already exists

The parameter is the operand on the right hand side of the operator

Example: Assignment

```
void Rectangle::operator =
  (const Rectangle &to_be_assigned) {
   this->m_width = to_be_assigned.m_width;
   this->m_length = to_be_assigned.m_length;
}

// Usage
Rectangle b(10, 12), c;
Rectangle a = b;
c = b;
```

Example: Arithmetic

```
Rectangle Rectangle::operator+(const Rectangle &to_add) {
  Rectangle output;
  output.m_width = this->m_width + to_add.m_width;
  output.m_length = this->m_length + to_add.m_length;
  return output;
}

// Usage
Rectangle a(10, 12), b(2, 3), c, d;
c = a + b;
/* is equivalent to */
d = a.operator+(b);
```

Example: Relational

```
bool Rectangle::operator==(const Rectangle &to_be_compared) {
  return ((m_width == to_be_compared.m_width) &&
          (m_length == to_be_compared.m_length));
}

bool Rectangle::operator!=(const Rectangle &to_be_compared) {
  // notice the re-usage of operator==
  return !(*this == to_be_compared);
}

// Usage
Rectangle a(10, 12), b(2, 3);
a == b;
/* is equivalent to */
operator==(a, b);
```

We can reuse previously defined operators

Increment/Decrement Operators

- Overloaded pre-(post-) increment(decrement) operators
  - Are distinguished by an (unused) int argument

```
C& operator++();
C& operator--();
```

    - Overloads the pre-increment (decrement) operator
    - Usually modifies the object and then returns *this

```
C operator++(int);
C operator--(int);
```

    - Overloads the post-increment (decrement) operator
    - Usually copies the object before modifying it and then returns the unmodified copy

Increment/Decrement Operators

```
const Rectangle& Rectangle::operator++() {
   m_width++;
   m_length++;
   return *this;
}
```

Pre-fix
First we increment, then we use the value
Return a const reference to this object

Post-increment
First we return the value, then we increment
Return a copy by value

```
const Rectangle Rectangle::operator++( int ) {
   Rectangle R(*this);
   ++(*this);
   return R;
}
```

It creates a copy of the object

Subscript operator

- Classes that behave like containers or pointers usually override the subscript operator []
  - **a[b]** is equivalent to **a.operator[](b)**
  - Type of b can be anything
    - For array-like containers, b is usually of type **size_t**
- Two types
  - const (for reading) and
  - non-const (for writing)

Example

```
class FooContainer {
    Foo* fooArray;


  public:
    Foo& operator[](size_t n) {
      return fooArray[n];
    }


    const Foo& operator[](size_t n) const {
      return fooArray[n];
    }
};
```

Private data

Need exceptions to check for the value of n

The compiler will verify const and non-const use of the methods

Conversion operator

```
type ()
```

- A class C can use converting constructors to convert values of other types to type C
- Similarly, conversion operators can be used to convert objects of type C to other types

Conversion operator

- Syntax
  - Conversion operators have the implicit return type *type*
  - They are usually declared as const
  - The **explicit** keyword can be used to prevent implicit conversions
  - Explicit conversions are done with static_cast

Example

```
class Float {
  float f;
  explicit operator float() const {
    return f;
  }
};

Float b{1.0};
float y = b; // ERROR, implicit conversion float
y = static_cast<float>(b); // OK
```

Deference operator

- Classes that behave like pointers usually override the operators * (dereference) and -> (member of pointer)
  - operator*() usually returns a reference
  - operator->() should return a pointer or an object that itself has an overloaded -> operator
- Also in this case, two types are available
  - const and
  - non-const

Example

The compiler will verify const and non-const use of the methods

```
class FooPtr {
  Foo* ptr;

  public:
    Foo& operator*() { return *ptr; }

    const Foo& operator*() const {
      return *ptr;
    }

    Foo* operator->() { return ptr; }

    const Foo* operator->() const {
      return ptr;
    }
};
```

Friend operators

- Output and input streams use the <<, >> operators for standard types
- It is possible to overload these operators for other (personal) classes
  - The overloaded operator function must then be declared as a friend of your class so it can access the private data within your class
    - Note that << and >> are also bitwise operators
    - Check carefully the parameters and return types to distinguish among them
  - Bitwise operators (shift) do not need to be friend

Example

```
class Rectangle {
  public:
    /* Overload the << and >> operators */
    friend istream& operator>>(istream& is, Rectangle& rect);

    friend ostream& operator<<(ostream& os,
      const Rectangle& rect);
};
```

Const object

Example

```
ostream &operator<<(ostream &os, const Rectangle &rect) {
  os << rect.m_width << "x" << rect.m_length;
  return os;
}

istream &operator>>(istream &is, Rectangle &rect) {
  is >> rect.m_width >> rect.m_length;
  return is;
}
```

Definition

I receive, modify, and return the stream

Usage

```
Rectangle a;
cin >> a; // get two numbers into m_width and m_length
cout << a; // print "<m_width> x <m_length>";
```

Rules for operator overloading

- Overloaded operators must satisfy specific rules
  - You can define only existing operators
    - + - * / % [] () ^ ! & < <= > >=
  - You can define operators only with their conventional number of operands
    - no unary <= (less than or equal)
    - no binary ! (not)
  - An overloaded operator must have at least one user-defined type as operand
    - int operator+(int,int); // error: you can't overload built-in +
    - Vector operator+(const Vector&, const Vector &); // ok

Rules for operator overloading

- Advice (not language rule)
  - Overload operators only with their conventional meaning
    - + should be addition
    - * be multiplication
    - [] be access
    - () be call
    - etc.
- Advice (not language rule)
  - Don’t overload unless you really have to