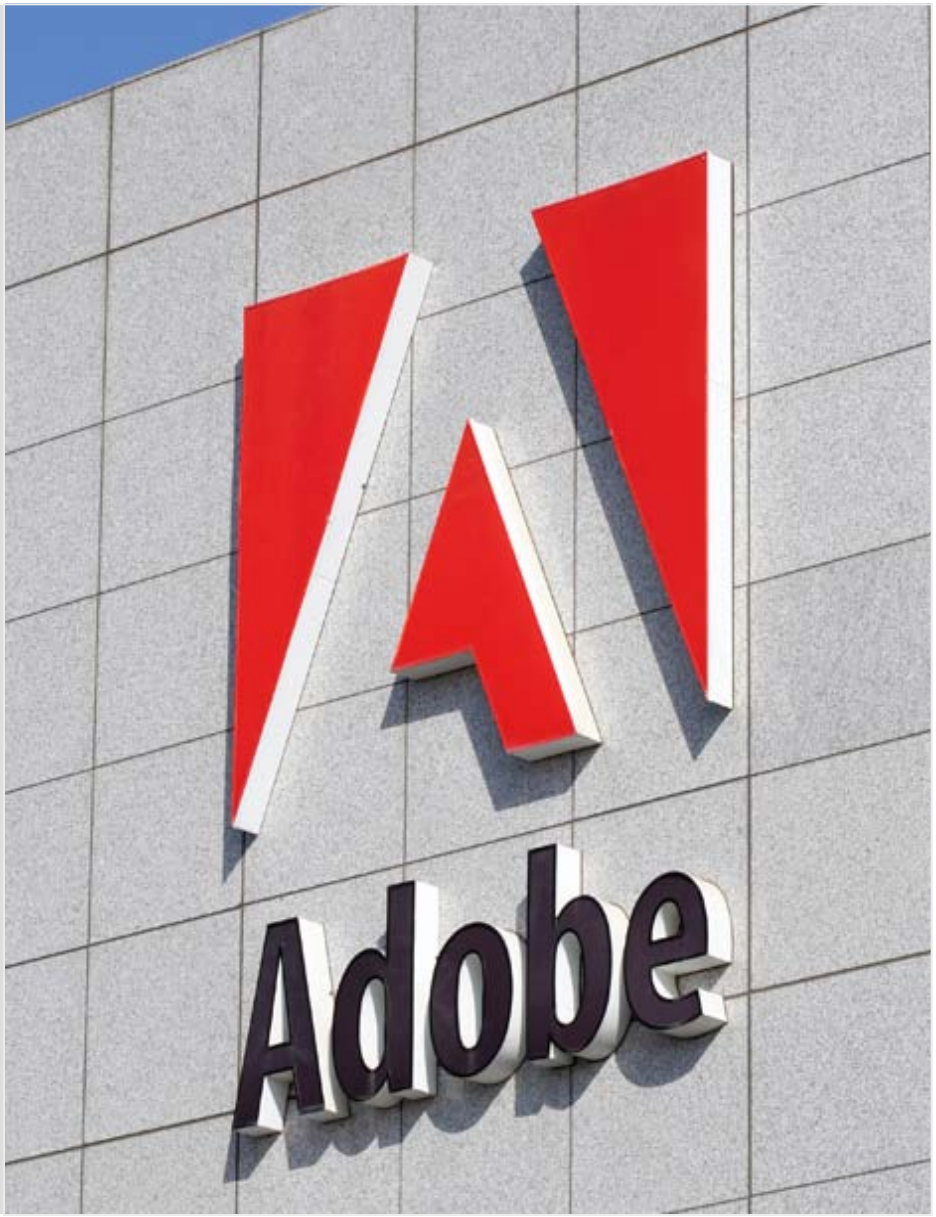
Introduction to GIL, Boost and Generic Programming



Hailin Jin

Advanced Technology Labs
Adobe Systems Incorporated

http://www.adobe.com/technology/people/sanjose/jin.html

Adobe

Outline

- GIL
- Boost
- Generic programming
- STL
- Summary

What is GIL?

- Generic Image Library
- C++ image library
- Open-source
- http://opensource.adobe.com/gil
- Well documented
- The core library only contains header files; No need to link anything
- Thread-safe
- Compatible with most C++ compilers (Visual Studio, gcc, etc)

GIL overview

- Image containers
- 1-D iterators for traversing rows and columns
- 2-D iterators (locator) for arbitrary 2D traversal
- Image views
- Image processing algorithms
- Extensions
  - io extension for reading and writing images of popular formats (PNG, JPEG, TIFF, …)

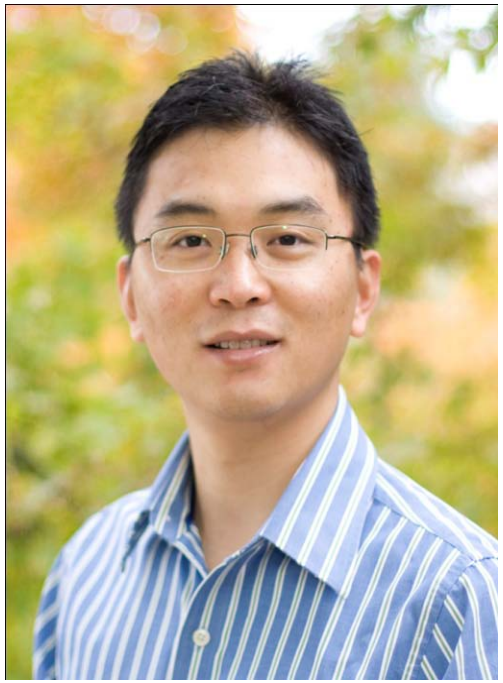
```
rgb8_image_t im;
read_png_image("test.png", im);
```

  - numeric extension for convolutions against fixed and variable-size kernels

```
kernel_1d_fixed<float, 7> kernel;
gray8_image_t im_in, im_out;
convolve_rows_fixed<gray32f_pixel_t>(const_view(im_in),
                                     kernel,
                                     view(im_out));
```

People

Lubomir Bourdev
Adobe



Hailin Jin
Adobe

Christian Henning
Independent

GIL history

- Started as an Adobe internal project in 2005
- Version 1.0 in June 2006
- Accepted into Boost in November 2006
- Version 2.0 in March 2007
- Constantly updated and patched with Boost releases

License

- Initially: MIT license
- Currently: Boost Software License (similar to the MIT license)
- Free to use, reproduce, display, distribute, execute, and transmit, including derivative works
- We retain the copyright

Image processing libraries

- Your own image libraries
- OpenCV
- CImg
- Vigra
- GIL works third-party algorithms, libraries, etc

- http://www.boost.org
- **High-quality** C++ libraries
- Open-source
- Started in 1998
- Latest version: 1.43.0 on May 6, 2010
- Strict review system
- 10 of the Boost libraries are included in the C++ Standard Library TR
- "It is usually a really dumb idea to go and reinvent a wheel that Boost already offers. "
  — Bjarne Stroustrup, inventor of C++

Boost usage

- http://www.boost.org/users/uses.html
- Adobe
- Google
- SAP
- McAFee
- Real Networks
- …

Boost components for computer vision

- Image (GIL)
- Graph (BGL)
- Thread
- Filesystem
- uBLAS
- Program options
- Math related libraries

Boost Graph Library

- Shortest paths
  - Dijkstra's shortest paths
  - Bellman-Ford shortest paths
  - Johnson's all-Pairs shortest paths
- Minimum spanning trees
  - Kruskal's minimum spanning tree
  - Prim's minimum spanning tree
- Connected components
  - Connected components
  - Strongly connected components
- Maximum flow
  - Edmonds Karp
  - Push-relabel
  - **Kolmogorov**
- Topological sort, sparse matrix ordering, and many more

Outline

- GIL
- Boost
- Generic programming
- STL
- Summary

GIL design principles

- Followed the paradigm of generic programming
- Clean separation between algorithms and containers
- Work with third-party algorithms, data structures, and libraries
- Toward optimal performance

Generic programming

- Programming paradigm for developing efficient, **reusable** software code
- Pioneered by Alexander Stepanov and David Musser
- First successful example is the Standard Template Library (STL) which is now part of the C++ standard
- Programming based on **parameterization**
  - Parameterize a data structure with different types (example: a **`std::vector<>`** with its element types)
  - Parameterize an algorithm with different types and algorithms (example: **`std::sort()`** function for **`int*`** and **`float*`** and with a comparison function)
- Generalize an algorithm or data structure to its most general and useful form
  - Better code
  - Fewer bugs

Polymorphism

- Allows values of different data types to be handled using a uniform interface
- Object-oriented programming (OOP)
  - Ad-hoc polymorphism
  - Class hierarchies: specified inheritance
  - Function/method overloading: different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations
  - Dynamic dispatch
- Generic programming
  - Parametric polymorphism (data structure and algorithms)
  - Code is written without mention of any specific type and thus can be used transparently with any number of new types
  - Static (compile-time) dispatch

Generic programming: a case study

- Compute the sum of a set of data

Object-oriented style

```
class int_vector {
private:
    int* array;
    int n;
    ...
public:
    ...
    int accumulate() {
        int result = 0;
            for (int i = 0; i < n; ++i)
                result = result + array[i];
        return result;
    }
    ...
};
```

Separate the algorithm from the container

```
int accumulate(int* array, int n) {
     int result = 0;
     for (int i = 0; i < n; ++i)
         result = result + array[i];
     return result;
}

class int_vector {
private:
     int* array;
     int n;
     ...
public:
     ...
     int size() {return n;}
     int* begin() {return array;}
     ...
};

int_vector a_vec;
int result = accumulate(a_vec.begin(), a_vec.size());
```

Same algorithm with different types

```
int accumulate(int* array, int n) {
     int result = 0;
     for (int i = 0; i < n; ++i)
         result = result + array[i];
     return result;
}

float accumulate(float* array, int n) {
     float result = 0;
     for (int i = 0; i < n; ++i)
         result = result + array[i];
     return result;
}
```

C++ templates

```
template <typename T>
T accumulate(T* array, int n) {
     T result = 0;
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}
```

String concatenation

```
template <typename T>
T accumulate(T* array, int n) {
     T result = 0;
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}

std::string concatenate(std::string* array, int n) {
     std::string result = "";
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}

template <typename T>
T accumulate(T* array, int n) {
     T result = T();                    ≠  T result;
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}
```

Generalize the interface and type requirements

```
template <typename T>
T accumulate(T* array, int n, T init) {
     for (int i = 0; i < n; ++i)
         init = init + array[i];
     return init;
}
```

Type requirements on **T**

- **T** must have a copy constructor **`T(const T&)`**
- **T** must have an assignment **`operator=()`**
- **T** must have a binary **`operator+(,)`**

Linked list

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
T accumulate(list_node<T>* first,
             T init) {
    while (first!=0) {
        init = init + first->value;
        first = first->next;
    }
    return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
T accumulate(list_node<T>* first,
             T init) {
    while (first!=0) {
        init = init + first->value;
        first = first->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* array, int n, T init) {
     for (int i = 0; i < n; ++i)
          init = init + array[i];
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
T accumulate(list_node<T>* first,
             T init) {
    while (first!=0) {
        init = init + first->value;
        first = first->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, int n, T init) {
     T* last = first + n;
     for (;first!=last;++first) {
          init = init + *first;
     }
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
T accumulate(list_node<T>* first,
             T init) {
    while (first!=0) {
        init = init + first->value;
        first = first->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, int n, T init) {
     T* last = first + n;
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
T accumulate(list_node<T>* first,
             T init) {
    while (first!=0) {
        init = init + first->value;
        first = first->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
struct list_iterator {
    list_node<T>* node;
};

template <typename T>
T accumulate(list_iterator<T> first,
             T init) {
    while (first.node!=0) {
        init = init + first.node->value;
        first.node = first.node->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
struct list_iterator {
    list_node<T>* node;
};

template <typename T>
T accumulate(list_iterator<T> first,
             list_iterator<T> last,
             T init) {
    while (first!=last) {
        init = init + first.node->value;
        first.node = first.node->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
struct list_iterator {
    list_node<T>* node;
};

template <typename T>
T accumulate(list_iterator<T> first,
             list_iterator<T> last,
             T init) {
    while (first!=last) {
        init = init + first.node->value;
        first.node = first.node->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}
```

```
    template <typename T>
    bool operator!=(list_iterator<T> x,
                    list_iterator<T> y) {
        return x.node!=y.node;
    }
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
struct list_iterator {
    list_node<T>* node;
};

template <typename T>
T accumulate(list_iterator<T> first,
             list_iterator<T> last,
             T init) {
    while (first!=last) {
        init = init + first.node->value;
        first.node = first.node->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}

    template <typename T>
    bool operator!=(list_iterator<T> x,
                    list_iterator<T> y) {
        return x.node!=y.node;
    }

    template <typename T>
    T operator*(list_iterator<T> x) {
        return x.node->value;
    }

    template <typename T>
    list_iterator<T>&
    operator++(list_iterator<T>& x) {
        x.node = x.node->next;
        return x;
    }
```

Unifying linked lists and arrays

```
template <typename T>
struct list_node {
    list_node<T>* next;
    T value;
};

template <typename T>
struct list_iterator {
    list_node<T>* node;
};

template <typename T>
T accumulate(list_iterator<T> first,
             list_iterator<T> last,
             T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* first, T* last, T init) {
     while (first!=last) {
          init = init + *first;
          ++first;
     }
     return init;
}

    template <typename T>
    bool operator!=(list_iterator<T> x,
                    list_iterator<T> y) {
        return x.node!=y.node;
    }

    template <typename T>
    T operator*(list_iterator<T> x) {
        return x.node->value;
    }

    template <typename T>
    list_iterator<T>&
    operator++(list_iterator<T>& x) {
        x.node = x.node->next;
        return x;
    }
```

Unified version

```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Type requirements

```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

- **I** must have a copy constructor **`I(const I&)`**
- **I** must have a binary **`operator!=(,)`**
- **I** must have an **`operator*()`**
- **I** must have an **`operator++()`**
- **T** must have a copy constructor **`T(const T&)`**
- **T** must have an assignment **`operator=()`**
- **T** must have a binary **`operator+(,)`**

Further generalization with function objects

```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}

template <typename I,
          typename T,
          typename B>
T accumulate(I first, I last, T init, B op) {
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

Type requirements

```
template <typename I,
          typename T,
          typename B>
T accumulate(I first, I last, T init, B op) {
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

- **I** must have a copy constructor **I(const I&)**
- **I** must have a binary **operator!=(,)**
- **I** must have an **operator*()**
- **I** must have an **operator++()**
- **T** must have a copy constructor **T(const T&)**
- **T** must have an assignment **operator=()**
- **B** must have a copy constructor **B(const B&)**
- **B** must have a binary application **operator()(,)**

Function object example

```
template <typename T>
struct moment_2 {
     T m1, m2;
     moment_2(T m1_, T m2_) : m1(m1_), m2(m2_) {}
};
template <typename T>
moment_2<T>
compute_moment_2(T* first, T* last, moment_2<T> init) {
     while (first!=last) {
          init.m1 += *first;
          init.m2 += *first**first;
          ++first;
     }
     return init;
}
```

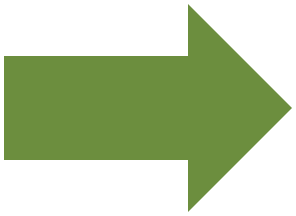
Function object example

```
template <typename T>
struct compute_moment_2_s {
     moment_2<T> operator()(moment_2<T> m, T x) const {
          m.m1 += x; m.m2 += x*x;
          return m;
     }
};
template <typename T>
moment_2<T>
compute_moment_2(T* first, T* last, moment_2<T> init) {
     return accumulate(first, last, init,
                       compute_moment_2_s<T>());
}

template <typename T>
struct max_s {
    T operator()(T x, T y) const {return x>y?x:y;}
};
template <typename T>
T range_max(T* first, T* last, T init) {
    return accumulate(first, last, init, max_s());
}
```

Lessons learned: #1

- Separate between an algorithms and a data structure

```
class int_vector {
private:
    int* array;
    int n;
    ...
public:
    ...
    int accumulate() {
        int result = 0;
            for (int i = 0; i < n; ++i)
                result = result + array[i];

        return result;
    }
    ...
};
```

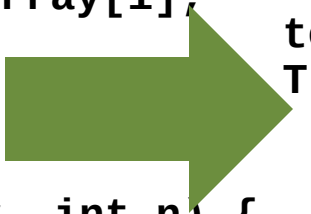


```
int accumulate(int* array, int n) {
     int result = 0;
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}

class int_vector {
private:
     int* array;
     int n;
     ...
public:
     ...
     int size() {return n;}
     int* begin() {return array;}
     ...
};

int_vector a_vec;
int result = accumulate(a_vec.begin()
```

Lessons learned: #2

- Generalize an algorithm to a general form

```
int accumulate(int* array, int n) {
     int result = 0;
     for (int i = 0; i < n; ++i)
          result = result + array[i];

     return result;
}

float accumulate(float* array, int n) {
     float result = 0;
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}
```

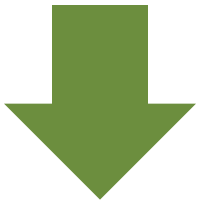


```
template <typename T>
T accumulate(T* array, int n) {
     T result = T();
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}
```

Lessons learned: #3

- Generalize the interface of an algorithm

```
template <typename T>
T accumulate(T* array, int n) {
     T result = T();
     for (int i = 0; i < n; ++i)
          result = result + array[i];
     return result;
}
```



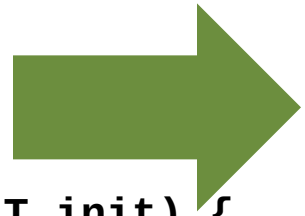
```
template <typename T>
T accumulate(T* array, int n, T init) {
     for (int i = 0; i < n; ++i)
          init = init + array[i];
     return init;
}
```

Lessons learned: #4

- Generalize through iterators

```
template <typename T>
T accumulate(list_iterator<T> first,
             T init) {
    while (first!=0) {
        init = init + first.node->value;
        first.node = first.node->next;
    }
    return init;
}
```

```
template <typename T>
T accumulate(T* array, int n, T init) {
     for (int i = 0; i < n; ++i)
         init = init + array[i];
     return init;
}
```



```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Lessons learned: #5

- Generalize through function objects

```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

```
template <typename I,
          typename T,
          typename B>
T accumulate(I first, I last, T init, B op) {
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

Concepts/Models

- Type requirements
- Concept: description of requirements on one or more types stated in terms of the existence and properties of procedures, type attributes, and type functions defined on the types
- Model: satisfies all the requirements of a concept
- Example

```
template <typename T>
T accumulate(T* array, int n, T init) {
      for (int i = 0; i < n; ++i)
           init = init + array[i];
      return init;
}
```

Concept

- **T** must have a copy constructor **`T(const T&)`**
- **T** must have an assignment **`operator=()`**
- **T** must have a binary **`operator+(,)`**

Model

- **`int`**, **`float`**, **`std::string`**, …

- Concepts are not designed or invented. Instead they are discovered

Iterators

- An interface between **sequential** data structures and algorithms
- Separates containers from algorithms
- Models: raw points and linked list iterators

```
template <typename I,
          typename T>
T accumulate(I first, I last, T init) {
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

- **`I`** must have a copy constructor **`I(const I&)`**
- **`I`** must have a binary **`operator!=(,)`**
- **`I`** must have **`operator*()`**
- **`I`** must have **`operator++()`**

More models of iterators

```
template <typename T>
class step_pointer {
privte:
    T* x;
    int step;
public:
    ...
    T operator*() {return *x;}
    step_pointer& operator++() {
        x += step; return *this;
    }
    ...
};
```

```
template <typename T>
class indirect_pointer {
privte:
    T* x;
    int* y;
public:
    ...
    T operator*() {return x[*y];}
    step_pointer& operator++() {
        ++y; return *this;
    }
    ...
};
```

```
template <typename T>
class value_iterator {
privte:
    T x;
public:
    ...
    T operator*() {return x;}
    step_pointer& operator++() {
        ++x; return *this;
    }
    ...
};
```

Compilers are working hard for us

- C++ compilers resolves all templates at compile time
- No dynamic (run-time) dispatch
- Function objects can be inlined; No function calls overhead

STL

- Standard Template Library
- http://www.sgi.com/tech/stl
- First widespread application of generic programming
- Part of the C++ standard

STL

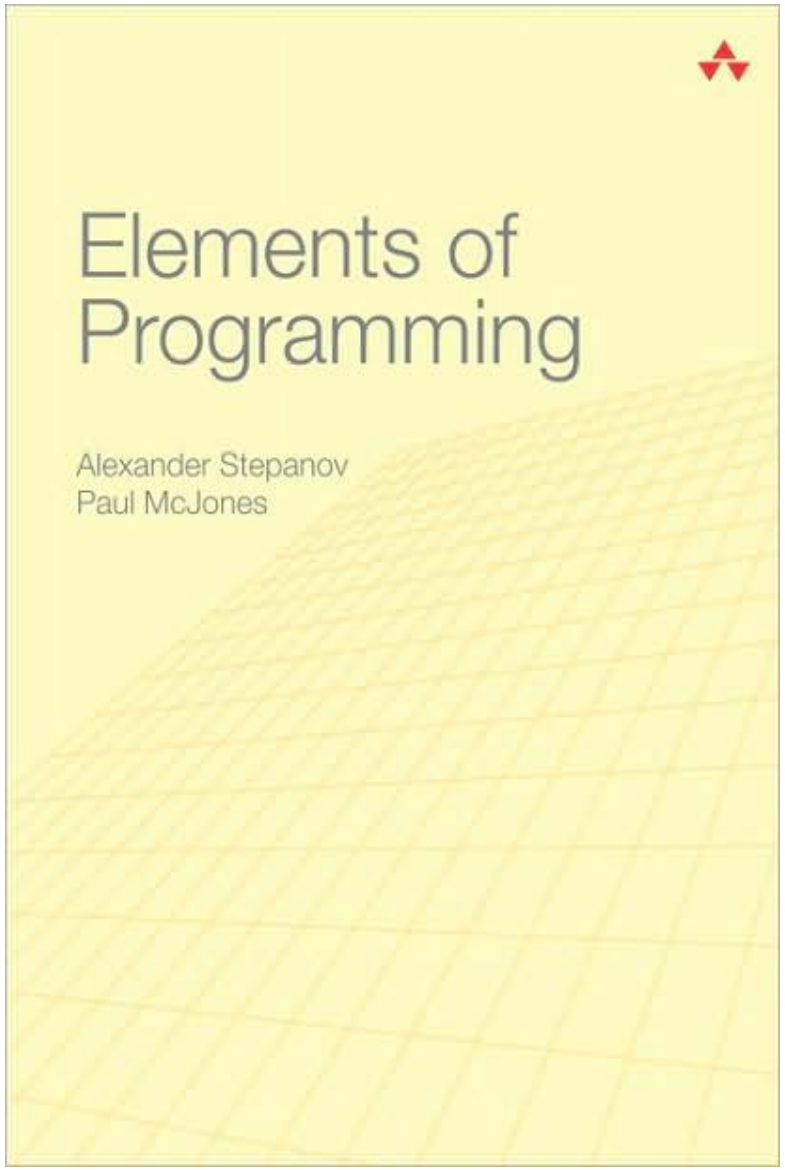
- Containers
  - **vector, list, slist**
  - **set, map, multiset, multimap**
- Algorithms
  - **find(), for_each(), fill(), transform(), copy()**
  - **partition(), stable_partition()**
  - **sort(), lower_bound(), nth_element(), merge()**
  - **set_difference(), set_union(), set_intersection()**
  - **accumulate(), inner_product(), partial_sum()**

Summary

- GIL
- Boost
- Generic programming
- STL
- Code reuse
- Better understanding of algorithms
- Better code

Book recommendation

- For those who desire a deep understanding of programming



Adobe