

STANDARD TEMPLATE LIBRARY C++ UNTUK MENGAJARKAN STRUKTUR DATA

Lucky E. Santoso

<http://www.lesantoso.com>

Lisensi Dokumen:

Copyright © 2003-2006 IlmuKomputer.Com

Seluruh dokumen di IlmuKomputer.Com dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari IlmuKomputer.Com.

ABSTRAK

Secara tradisional, matakuliah Struktur Data difokuskan pada detail implementasi dari sejumlah struktur data sederhana. Sebagai alternatif, diusulkan penggunaan Standard Template Library (STL) yang merupakan bagian dari pustaka baku C++. Dengan penggunaan STL, fokus matakuliah dapat digeser kepada pemilihan dan penggunaan struktur data sederhana secara tepat, serta perluasan ke arah struktur data yang lebih kompleks dan lebih dekat ke masalah nyata. Sebagai ilustrasi, ditunjukkan bahwa implementasi *general tree*, yang tingkat kesulitannya membuatnya kerap dihindari dalam pengajaran Struktur Data tradisional, dapat dengan mudah dilakukan dengan bantuan STL.

Kata kunci: *Standard Template Library, C++, struktur data*

1 PENDAHULUAN

Secara tradisional, matakuliah Struktur Data difokuskan pada detail implementasi dari sejumlah struktur data sederhana; mahasiswa membuat dan mempelajari kode untuk *linked list* dan struktur data lain dari nol atau berdasarkan kode buatan dosen atau

pengarang buku teks. Sebagai alternatif, diusulkan penggunaan Standard Template Library C++ (STL) yang telah teruji dan tersedia luas.

STL yang termasuk dalam pustaka baku C++ ini telah menyediakan sejumlah besar struktur data dan algoritma untuk mengolah struktur data tersebut, sehingga pemrogram tidak lagi terbebani

oleh detail implementasi. Dengan penggunaan STL, fokus matakuliah dapat digeser ke jenjang yang lebih tinggi, yaitu kepada pemilihan dan penggunaan struktur data secara tepat. Hal ini tepat dalam kurikulum yang memberi penekanan ke *software engineering*.

Walaupun STL hingga saat ini belum mencakup semua struktur data yang mungkin ada, namun dengan penggunaan STL, perluasan juga dapat lebih mudah dilakukan ke struktur data yang lebih kompleks dan lebih dekat ke masalah nyata. Misalnya implementasi *general tree*, yang biasanya dihindari dalam pengajaran Struktur Data tradisional karena tingkat kesulitannya yang tinggi, dapat dengan mudah dibuat dengan bantuan STL.

STL juga dapat digunakan oleh mahasiswa dalam matakuliah-matakuliah selanjutnya, dan, lebih penting lagi, dalam industri setelah nantinya mereka lulus.

Dikembangkan oleh Alexander Stepanov dan Meng Lee dari Hewlett-Packard, STL telah dimasukkan sebagai bagian dari pustaka baku C++ oleh ANSI/ISO pada tahun 1994. Stepanov (1995) menyatakan bahwa STL telah dikembangkan dengan sangat memperhatikan sifat umum dan efisiensi. Dua aspek ini memiliki arti penting dalam industri.

Dengan perhatian pada aspek pertama, yaitu sifat umum, STL memiliki *reusability* yang tinggi sehingga penggunaannya dapat menghemat waktu dan sumber daya dalam pengembangan sistem. Semua struktur data dalam STL memiliki sifat umum, yaitu dapat menampung elemen bertipe data apa saja sesuai dengan yang ditentukan oleh pemrogram. Hal ini dimungkinkan oleh penggunaan fasilitas *template* (*class* dan *function* yang memiliki parameter berupa tipe data) yang tersedia pada

C++.

Ini berbeda misalnya dengan pustaka baku Java untuk struktur data, yaitu Java Collections Framework (JCF), yang struktur datanya dapat memiliki sifat umum berkat penggunaan *inheritance*. Namun atas desakan para pemrogram Java, fasilitas yang serupa dengan *template* juga telah ditambahkan pada versi Java yang terbaru, Java 2 Platform Standard Edition 5.0. Ini berarti bahwa pengalaman dalam menggunakan STL akan dapat bermanfaat juga saat bekerja dengan bahasa pemrograman lain seperti Java.

STL lebih jauh lagi memisahkan struktur data dengan algoritma, sehingga algoritma STL memiliki sifat umum, yaitu dapat diterapkan pada berbagai macam struktur data. Hal ini dimungkinkan oleh penggunaan fasilitas *template* dan karena algoritma STL tidak mengakses elemen secara langsung namun melalui sejenis *pointer*.

Ada pandangan bahwa pemisahan ini merupakan kelemahan STL karena membuatnya tidak murni berorientasi objek dan karenanya memperbesar peluang terjadinya kesalahan pemrograman (Keffer 1995). Namun keuntungan dari pemisahan ini adalah bahwa pemrogram dapat menerapkan algoritma baru buatannya sendiri pada struktur data STL, dan dapat menerapkan algoritma STL pada struktur data baru buatannya sendiri. Fleksibilitas ini tentunya meningkatkan *reusability* STL.

Dengan perhatian pada aspek kedua, yaitu efisiensi, STL dapat digunakan tanpa perlu mengorbankan efisiensi yang selama ini telah menjadi ciri khas dari C and C++. Walaupun memiliki sifat umum, tiap komponen dalam STL memiliki implementasi yang efisiensinya tidak jauh berbeda dengan kode ekuivalennya yang dibuat secara manual. Bahkan ada argumentasi bahwa

pemanggilan algoritma STL lebih baik daripada pembuatan algoritma secara manual (yang biasanya dilakukan pemrogram dengan menggunakan *loop*) karena algoritma STL seringkali lebih efisien. Selain alasan efisiensi ini ada alasan-alasan lain, yaitu pembuatan *loop* lebih rentan terhadap kesalahan, dan pemanggilan algoritma STL seringkali menghasilkan kode yang lebih ringkas dan jelas sehingga lebih mudah dipelihara (Meyers 2001).

Berikutnya akan diberikan suatu pengantar singkat mengenai STL, dan kemudian akan dibicarakan hal-hal yang perlu diperhatikan dalam penggunaan STL untuk mengajarkan Struktur Data.

2 STL

Penjelasan dasar mengenai STL dapat diperoleh antara lain dari buku teks karya Hubbard (2000), Savitch (2002), dan Deitel dan Deitel (2003), sedangkan penjelasan lebih lanjut dapat diperoleh antara lain dari artikel karya Stepanov dan Lee (1995) dan buku teks karya Austern (1999). Berikut hanya akan diberikan pengantar singkat mengenai STL. STL terdiri atas tiga komponen, yaitu *container*, *iterator*, dan *algorithm*. Pada prinsipnya dapat dikatakan bahwa *algorithm* melakukan manipulasi terhadap *container* dengan bantuan *iterator*.

2.1 Container

Container adalah struktur data yang dapat menampung elemen bertipe data apa saja. *Container* diimplementasikan sebagai *container class* yang merupakan suatu *template class*, yaitu class yang memiliki parameter berupa tipe data. Beberapa di antaranya, yaitu *vector*, *list*, *deque*, *stack*, *queue*, *priority_queue*, *set*, dan *map* akan dibicarakan di sini. Tiap *class* tersebut didefinisikan pada *header*

dengan nama yang sama dengan nama *class* yang dimilikinya, kecuali *priority_queue* didefinisikan pada *header* <queue>.

2.1.1 vector

Container *vector* diimplementasikan sebagai *array* dengan realokasi memori otomatis saat kapasitasnya dalam menampung elemen perlu ditambah. *Iterator*-nya berjenis *random-access*. *Member function* yang dimiliki antara lain:

- *vector()* merupakan *default constructor* yang membentuk *vector* kosong.
- *vector(n)* merupakan *constructor* yang membentuk *vector* berisi *n* buah elemen.
- *empty()* memberikan *true* jika dan hanya jika *vector* kosong.
- *size()* memberikan jumlah elemen.
- *capacity()* memberikan jumlah maksimum elemen yang dapat ditampung tanpa realokasi memori.
- *reserve(n)* melakukan alokasi atau realokasi memori sebesar *n*.
- operator [*i*] memberikan elemen ke-*i*.
- *front()* memberikan elemen paling awal.
- *back()* memberikan elemen terakhir.
- *begin()* memberikan *iterator* yang menunjuk ke elemen paling awal.
- *end()* memberikan *iterator* yang menunjuk ke posisi setelah elemen terakhir.
- *push_back(x)* menyisipkan elemen *x* pada bagian akhir.
- *pop_back()* menghapus elemen terakhir.
- *insert(p, x)* menyisipkan elemen *x* pada lokasi sebelum yang ditunjuk oleh *p*.

- `erase(p)` menghapus elemen pada lokasi yang ditunjuk oleh `p`.
- `clear()` menghapus semua elemen.

2.1.2 list

Container `list` diimplementasikan sebagai *doubly-linked list*. *Iterator*-nya berjenis *bidirectional*. *Member function* yang dimiliki oleh `vector` juga dimiliki oleh `list`, kecuali `capacity()`, `reserve()`, dan operator `[]`. *Class* `list` memiliki *member function* yang tidak dimiliki oleh `vector`, antara lain:

- `push_front(x)` menyisipkan elemen `x` pada bagian awal.
- `pop_front()` menghapus elemen paling awal.
- `splice(p, l, p1, p2)` memindahkan elemen pada posisi `p1` hingga `p2-1` dari `list l` ke posisi `p` pada `list` ini.
- `remove(x)` menghapus semua elemen `x`.
- `unique()` menghapus semua elemen yang memiliki duplikat.
- `merge(l)` menggabungkan semua elemen pada `list l` ke `list` ini. Kedua `list` harus sudah terurut.
- `reverse()` memutarbalikkan susunan elemen.
- `sort()` mengurutkan elemen.

2.1.3 deque

Diimplementasikan sebagai sederetan *pointer* yang menunjuk ke sejumlah *array*, elemen dapat secara efisien disisipkan ke dan dihapus dari bagian awal dan bagian akhir *container* `deque`. *Iterator*-nya berjenis *random-access*. *Member function* yang dimiliki oleh `vector` juga dimiliki oleh `deque`, kecuali `capacity()` dan `reserve()`. *Class* `deque` memiliki dua buah *member function* yang tidak

dimiliki oleh `vector`, yaitu `push_front(x)` dan `pop_front()`.

2.1.4 stack

Container `stack` dibuat berdasarkan `deque` dan tidak memiliki *iterator*. *Container* ini hanya mengizinkan penyisipan dan penghapusan elemen pada bagian atas. *Member function* yang dimiliki antara lain:

- `empty()` memberikan `true` jika dan hanya jika `stack` kosong.
- `size()` memberikan jumlah elemen.
- `top()` memberikan elemen paling atas.
- `push(x)` menyisipkan elemen `x`.
- `pop()` menghapus elemen.

2.1.5 queue

Container `queue` dibuat berdasarkan `deque` dan tidak memiliki *iterator*. *Container* ini hanya mengizinkan penyisipan elemen pada bagian belakang dan penghapusan elemen pada bagian depan. Selain `empty()`, `size()`, `push(x)`, dan `pop()`, *member function* yang dimiliki antara lain:

- `front()` memberikan elemen paling depan.
- `back()` memberikan elemen paling belakang.

2.1.6 priority_queue

Container `priority_queue` dibuat berdasarkan `vector` dan tidak memiliki *iterator*. *Container* ini hanya mengizinkan penghapusan elemen berprioritas paling tinggi. Harus sudah didefinisikan operator `<` untuk tipe data elemen. Jika `x < y` memberikan `true`, ini berarti bahwa prioritas elemen `y` lebih tinggi daripada elemen `x`. *Member function* yang dimiliki serupa dengan yang dimiliki oleh `stack`.

2.1.7 set

Diimplementasikan sebagai *red-black tree*, *container* ini menampung elemen secara terurut dan tanpa duplikat. Harus sudah didefinisikan operator `<` untuk tipe data elemen. *Iterator*-nya berjenis *bidirectional*. Selain `empty()`, `size()`, `begin()`, `end()`, `erase(p)`, dan `clear()`, yang serupa dengan yang dimiliki oleh *vector*, *member function* yang dimiliki antara lain:

- `insert(x)` menyisipkan elemen `x`
- `erase(x)` menghapus elemen `x`
- `find(x)` memberikan *iterator* yang menunjuk ke elemen `x`, atau memberikan `end()` jika elemen `x` tidak ada.

2.1.8 map

Container `map` serupa dengan `set`, namun elemennya tidak tunggal melainkan pasangan indeks-nilai. *Member function* yang dimiliki serupa dengan yang dimiliki oleh `set`, dengan tambahan operator `[key]` yang memberikan nilai dengan indeks `key`, jika ada, dan menyisipkan elemen dengan indeks tersebut, jika belum ada.

2.2 Iterator

Berperilaku seperti *pointer*, *iterator* digunakan untuk menelusuri dan memanipulasi elemen-elemen yang tersimpan dalam suatu *container*. *Iterator* diimplementasikan sebagai *embedded class* dalam *container class* yang bersangkutan. Operator yang dapat diterapkan ke semua jenis *iterator* antara lain:

- `++` membuat *iterator* menunjuk ke elemen selanjutnya.
- `--` membuat *iterator* menunjuk ke elemen sebelumnya.
- `==` dan `!=` untuk mengetahui apakah dua *iterator* menunjuk ke elemen yang

sama.

- `*` memberikan akses ke elemen yang ditunjuk oleh *iterator*.

Sedangkan operator yang hanya dapat diterapkan ke *iterator random-access* antara lain `[i]` yang memberikan akses ke elemen berjarak `i` dari elemen yang ditunjuk oleh *iterator*.

2.3 Algorithm

Diimplementasikan sebagai *template function*, yaitu *function* yang memiliki parameter berupa tipe data, *algorithm* adalah algoritma yang dapat diterapkan ke rentang elemen tertentu dalam *container* yang ditunjukkan oleh pasangan *iterator*. *Template function* yang didefinisikan pada *header* `<algorithm>` antara lain:

- `binary_search(p1, p2, x)` memberikan `true` jika elemen `x` terdapat pada rentang `p1` hingga `p2-1`. Rentang harus sudah terurut secara *ascending*.
- `copy(p1, p2, p)` menduplikasi elemen pada rentang `p1` hingga `p2-1` ke rentang berukuran sama yang dimulai dari `p`.
- `count(p1, p2, x)` memberikan cacah kemunculan elemen `x` pada rentang `p1` hingga `p2-1`.
- `equal(p1, p2, p)` memberikan `true` jika rentang `p1` hingga `p2-1` serta rentang berukuran sama yang dimulai dari `p` memiliki elemen dengan nilai dan urutan yang sama.
- `find(p1, p2, x)` memberikan posisi kemunculan pertama elemen `x` pada rentang `p1` hingga `p2-1`.
- `merge(p1, p2, q1, q2, r)` menggabungkan elemen pada rentang `p1` hingga `p2-1` dan rentang `q1` hingga `q2-1` ke rentang yang dimulai dari `r`.
- `random_shuffle(p1, p2)` mengacak susunan elemen pada rentang `p1` hingga `p2-1` (hanya untuk

iterator random-access).

- `remove(p1, p2, x)` menggeser ke belakang semua elemen `x` pada rentang `p1` hingga `p2-1`, lalu memberikan `p` di mana `p1` hingga `p-1` adalah rentang tanpa elemen `x`.
- `reverse(p1, p2)` memutar balik susunan elemen pada rentang `p1` hingga `p2-1`.
- `search(p1, p2, q1, q2)` memberi posisi kemunculan pertama isi rentang `q1` hingga `q2-1` pada rentang `p1` hingga `p2-1`.
- `sort(p1, p2)` mengurutkan elemen pada rentang `p1` hingga `p2-1` (hanya untuk *iterator random-access*).

3 STL UNTUK MENGAJARKAN STRUKTUR DATA

Dalam penggunaan STL untuk mengajarkan Struktur Data, dianggap mahasiswa telah memiliki pengetahuan dasar mengenai C atau C++ dari matakuliah sebelumnya. Akan sangat membantu jika mahasiswa telah atau sedang menempuh matakuliah Pemrograman Berorientasi Objek atau ekuivalennya dalam C++. Namun jika tidak demikian, dalam beberapa kali pertemuan awal mahasiswa dapat diperkenalkan kepada *class*, *operator overloading*, dan *template*; sedangkan konsep-konsep C++ lain seperti *inheritance* dan *polymorphism* maupun detail-detail seperti *destructor*, *copy constructor*, dan *initialization section* untuk sementara dapat dengan aman diabaikan.

Mahasiswa dapat diperkenalkan kepada STL antara lain melalui kode. Sebagai contoh, kode-kode berikut menyisipkan sejumlah elemen ke dalam *container*, menggunakan *iterator* untuk melakukan penelusuran sambil mencetak nilai tiap elemen, dan kemudian menggunakan *algorithm find* untuk mencari suatu

elemen dengan nilai tertentu. Kode berikut menggunakan *vector*:

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

void main() {
    vector<int> c;
    vector<int>::iterator i;
    c.push_back(1);
    c.push_back(3);
    c.push_back(2);
    for(i=c.begin(); i!=c.end(); i++)
        cout << *i;
    i = find(c.begin(), c.end(), 2);
    if(i != c.end()) cout << "found";
    else cout << "not found";
}
```

Sedangkan kode berikut menggunakan *list*:

```
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

void main() {
    list<char> c;
    list<char>::iterator i;
    c.push_front('C');
    c.push_front('A');
    c.push_front('B');
    for(i=c.begin(); i!=c.end(); i++)
        cout << *i;
    i = find(c.begin(), c.end(), 'B');
    if(i != c.end()) cout << "found";
    else cout << "not found";
}
```

Dalam kode-kode di atas, *vector* dibuat menampung elemen bertipe `int` dan *list* dibuat menampung elemen bertipe `char`. Ini menunjukkan sifat umum *container*. Kode lain untuk kasus lain perlu juga diadakan, misalnya kasus *container* menampung *pointer* ke *object*

merupakan kasus penggunaan STL yang sering terjadi (Eckel 1996).

Kode-kode di atas juga menggambarkan bahwa cara penggunaan *iterator* adalah seragam untuk *container* yang berbeda-beda, dan bahwa *algorithm* dapat diterapkan pada berbagai macam *container*.

Dengan penggunaan STL, tidak berarti semua pendekatan dalam pengajaran tradisional dapat ditinggalkan. Yang perlu dipertahankan adalah antara lain pengantar ke analisis kompleksitas (notasi *big-O*).

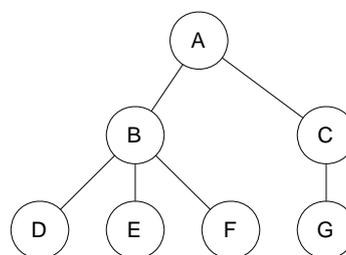
Pemilihan struktur data secara tepat kini mendapat perhatian khusus. STL menyediakan berbagai macam *container* dan mahasiswa perlu dianjurkan untuk memilih *container* yang paling efisien sesuai permasalahan. Sebagai contoh, biasanya *vector* merupakan *container* pilihan, namun jika akan sering dilakukan operasi penyisipan atau penghapusan pada posisi sembarang dalam struktur data, maka *list* lebih tepat dipilih karena operasi tersebut kompleksitasnya linear untuk *vector* namun konstan untuk *list*. Pemilihan struktur data ini dapat diperjelas dengan menunjukkan contoh-contoh permasalahan nyata.

Penggunaan struktur data secara tepat juga kini mendapat perhatian khusus. Meskipun *algorithm* memiliki sifat umum, yaitu dapat diterapkan pada berbagai macam *container*, namun mahasiswa perlu dianjurkan untuk memilih kombinasi yang efisien dan menghindari kombinasi yang tidak efisien. Sebagai contoh, dalam memilih *algorithm* pencarian, perlu dilihat apakah *iterator* menunjuk ke rentang terurut. Jika ya, sebaiknya dipilih *algorithm* yang kompleksitasnya logaritmik seperti *binary_search*. Jika tidak, terpaksa dipilih *algorithm* yang kompleksitasnya linear seperti *find*.

Sebelum menerapkan suatu *algorithm*

pada suatu *container*, perlu dipastikan bahwa tidak ada *member function* dalam *container* yang dapat mengerjakan tugas tersebut secara lebih efisien. Sebagai contoh, *set* dan *map* selalu terurut dan memiliki *member function* *find* yang kompleksitasnya logaritmik, sehingga tentunya *algorithm* *find* sama sekali tidak tepat diterapkan pada mereka.

Dengan penggunaan STL, perluasan juga dapat lebih mudah dilakukan ke struktur data yang lebih kompleks, seperti misalnya *general tree*. *General tree*, bentuk umum dari *tree*, merupakan struktur data yang sering digunakan, misalnya dalam aplikasi-aplikasi kecerdasan buatan. Gambar 1 memberikan suatu contoh *general tree*. Setiap *node* di dalam *general tree* dapat memiliki *child* dalam jumlah tak terbatas, berbeda dengan *node* di dalam *binary tree* yang hanya dapat memiliki paling banyak dua *child*; akibatnya implementasi untuk *general tree* lebih kompleks daripada untuk *binary tree*.



Gambar 1. Contoh *general tree*

Salah satu cara untuk membuat implementasi *general tree* adalah dengan menempatkan suatu *linked list* di setiap *node* untuk menampung sejumlah pointer yang masing-masing menunjuk ke tiap *child* dari *node* tersebut. Ini dapat dengan mudah dilakukan dengan menggunakan *list* seperti ditunjukkan oleh kode pada Gambar 2.

Dalam program tersebut, nilai dari *node-node* dimasukkan secara *level order* dan setelah itu dilakukan penelusuran secara

inorder dan *postorder* sambil mencetak nilai dari setiap *node*. Demi kesederhanaan, implementasi ini memang terbatas fasilitasnya. Namun tidak sulit untuk melengkapinya dan menjadikannya sebagai komponen buatan sendiri yang kompatibel dengan komponen STL lain.

Berikut contoh dialog program yang berkaitan dengan *general tree* pada Gambar 1:

```
Enter root: A
Enter children of A: BC
Enter children of B: DEF
Enter children of C: G
Enter children of D:
Enter children of E:
Enter children of F:
Enter children of G:
Preorder: ABDEF CG
Postorder: DEF B G CA
```

4 KESIMPULAN

```
// Copyright (c) L. E. Santoso 2004. All rights reserved

#include <list>
#include <iostream>
using namespace std;

class N { // node
public:
    char data;
    list<N> *link;
    N(char d, list<N> *l) { data = d; link = l; }
};

void createSubs(list<N> *l) {
    for(list<N>::iterator i = l->begin(); i != l->end(); i++) {
        cout << "Enter children of " << i->data << ": ";
        char c;
        cin.get(c);
        while(c != '\n') {
            if(!i->link) i->link = new list<N>;
            i->link->push_back(N(c, 0));
            cin.get(c);
        }
    }
    for(i = l->begin(); i != l->end(); i++)
        if(i->link) createSubs(i->link);
}

void traverse(list<N> *l, int order) {
    for(list<N>::iterator i = l->begin(); i != l->end(); i++) {
        if(order == 0) cout << i->data; // preorder
        if(i->link) traverse(i->link, order);
        if(order == 1) cout << i->data; // postorder
    }
}

void main() {
    cout << "Enter root: ";
    char c;
    cin.get(c); cin.ignore();
    list<N> *root = new list<N>;
    root->push_back(N(c, 0));
    createSubs(root);
    cout << "Preorder: "; traverse(root, 0); cout << endl;
    cout << "Postorder: "; traverse(root, 1); cout << endl;
}
```

Gambar 2. Kode untuk *general tree*

Diusulkan penggunaan STL oleh staf pengajar yang selama ini masih mengajarkan matakuliah Struktur Data dengan fokus tradisional, yaitu pada detail implementasi struktur data sederhana. Dengan penggunaan STL, fokus matakuliah dapat digeser ke jenjang yang lebih tinggi, yaitu kepada pemilihan dan penggunaan struktur data sederhana tersebut secara tepat. Dengan

penggunaan STL, perluasan juga dapat dilakukan ke struktur data yang lebih kompleks dan lebih dekat ke masalah nyata. Sebagai ilustrasi, ditunjukkan bahwa implementasi *general tree*, yang tingkat kesulitannya membuatnya kerap dihindari dalam pengajaran Struktur Data tradisional, dapat dengan mudah dilakukan dengan bantuan STL.

DAFTAR PUSTAKA

- Austern, M. H. 1999. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Reading, MA: Addison-Wesley.
- Deitel, H. M., and P. J. Deitel. 2003. *C++ how to program*. 4th ed. Upper Saddle River, NJ: Prentice-Hall.
- Eckel, B. 1996. Putting STL to work. *Unix Review*, October.
- Hubbard, J. R. 2000. *Schaum's outline of theory and problems of data structures with C++*. New York: McGraw-Hill.
- Keffer, T. 1995. Programming with the Standard Template Library. *Dr. Dobb's Journal*, Special Issue.
- Meyers, S. 2001. STL algorithms vs. hand-written loops. *C/C++ Users Journal*, October.
- Savitch, W. J. 2002. *Absolute C++*. Boston: Addison-Wesley.
- Stepanov, A. 1995. The Standard Template Library. *Byte*, October.
- Stepanov, A., and M. Lee. 1995. *The Standard Template Library*. Hewlett-Packard. <http://www.cs.rpi.edu/~musser/doc.ps>

BIOGRAFI DAN PROFIL



Lucky E. Santoso, lahir di Semarang, 4 Agustus 1972, adalah dosen di bidang Ilmu Komputer dengan pengalaman lebih dari 10 tahun di sejumlah perguruan tinggi di Jakarta dan Yogyakarta. Penulis ini juga adalah pengembang independen untuk aplikasi berbasis Windows dan Web dengan pengalaman lebih dari 10 tahun. Sarjana Komputer, Sarjana Sains di bidang Fisika, dan Magister Manajemen dengan spesialisasi di bidang Keuangan ini juga merupakan seorang Sun Certified Java Programmer (SCJP), dan telah lulus ujian IBM OOAD with

UML.

Informasi lebih lanjut tentang penulis ini bisa didapatkan di <http://www.lesantoso.com>