Effective Analysis Programming - Part III

Hartmut Stadie, Christoph-Erdmann Pfeiler

GridKa school
September, 7th 2011

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

Literature

Literature:

- Stroustrup: “The C++ Programming Language”, 3rd edition
- Sutter, Alexandrescu: “C++ Coding Standards”
- Press et al.: “Numerical Recipes 3rd edition”
- Meyers: “Effective C++” etc.
- ...

Read a book on programming!
You spend a lot of time writing code and should know how to do this!

Coding Guidelines

There are many ways to write C++ code. Use the right one!

Disclaimer

The following is heavily influenced by the book “C++ Coding Standards”.

Main idea:

Minimize the chance of bugs appearing in your code and find them quickly:

- use the compiler to find bugs
- write simple code, use clear designs
- always assume that the code will last long and be used by someone else

- Introduction
- STL
  - Introduction
  - Strings
  - Container
    - Operations
  - Algorithms
  - Traps
- Rules for STL

Standard Library

Die C++ Standardbibliothek:

- unterstützt Spracheigenschaften wie Speicherverwaltung und Laufzeittypinformation
- gibt Informationen über Aspekte der Sprachimplementation, max(float)
- Funktionen, die nicht direkt für jedes System in der Sprache implementiert werden können: sqrt
- Komplexe Werkzeuge, die dem Programmer auf jedem System zur Verfügung stehen: Listen, Maps, Sortierfunktionen, I/O streams
- erlaubt diese Werkzeuge zu erweitern
- Fundament für weitere Bibliotheken

Alle Funktionen, Klassen im Namensraum *std*.

Strings

Text schwierig in C++
Lösung: string
Konstruktoren:

```
string s = "Hartmut";
string s2(s,0,4);//Hart
string s3(5,'h');//hhhhh
```

Zuweisung:

```
string s = "Hartmut";
string t;
t=s;
s[2] = 'u';
```

C-style Strings:

```
const char* text = s.c_str();
```

Speicher gehört string!
einzelne Buchstaben:

```
char c3 = s[3];
char c4 = s.at(3);
```

Strings

Vergleich:

```
if(s == "Hans") { }
```

Verketten:

```
s2 = "Welt";
s += ' ';
s.append(s2);
s.insert(7," ");
```

Finden, Löschen und Ersetzen:

```
string s("Hartmut");
string::size_type pos_a = s.find('a');
string::size_type pos_mut = s.find("mut");
string::size_type pos_t2 = s.rfind('t');
string::size_type npos = s.length();
if(pos_t2 < npos) ...
s.erase(pos_t2,1);
s.replace(pos_a,1,'u');
```

Beispiel: string.cxx

Container

Probleme mit Feldern in C++

- feste Größe
- Fehlerquelle: new/delete

Lösung: STL Container
Dokumentation: http://www.sgi.com/tech/stl/
Beispiel: Wörter einlesen, Buchstaben zählen (stl.cxx)

Operations

Iteratoren:

```
for(vector::iterator i = v.begin(); v != v.end() ; ++i) {
  i->print();//oder (*i).print();
}
```

Elementzugriff:

```
front();
back();
[];
at();
```

Stapel- und Queueoperationen:

```
push_back();
pop_back();
push_front();
pop_front();
```

Operations II

Listenoperationen:

```
insert(p,x);
erase(p);
clear();
```

andere Operationen:

```
size();
empty();
capacity();
reserve();
resize();
swap();
```

Container implemenations

different container: vector; list; map
runtime behavior:

	[]	insert/remove	front	back	iterator
vector	const	O(n)+		const+	random
list		const	const	const	Bi
queue		const	const		
map	O(log(n)	O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	random

Algorithms

in *<algorithm>*:

```
for_each;
find;
count;
copy;
swap;
replace;
fill;
remove;
sort;
binary_search;
partition;
min;
max;
min_element;
```

Einfache Beispiele

```
using std;
list<string> s;
s.push_back("Birne");
s.push_back("Apfel");
s.push_back("Kirsche");
list<string>::iterator i = find(s.begin(),s.end(),"Apfel");
if(i != s.end()) { }
s.sort(s.begin(),s.end());
list<string>::iterator i = min_element(s.begin(),s.end());
int i = 5, j = 9;
int m = max(i,j);
```

Komplexeres Beispiel

Sortiere Liste nach eigenem Kriterium, Länge des Strings: (stl.cxx)

```
bool compareLength(const std::string& s1, const std::string& s2) {
  return s1.length() < s2.length();
}
...
sort(v.begin(),v.end(),compareLength);
min_element(v.begin(),v.end(),compareLength);
```

Fallen

Fallen:

- Container enthalten Kopien
  - Ort im Speicher kann sich ändern(slicing)
    Benutze keine Referenzen oder Zeiger auf Elemente im Container!
  - Kopie kann teuer sein
    Benutze Zeiger und lösche jedes Element vor dem Entfernen
- Iteratoren sind zerbrechlich!!!

- Introduction
- STL
- Rules for STL
  - Do not write namespace **using**s in a header file or before an **#include**
  - Use **vector** by default
  - Use **vector** and **string** instead of arrays
  - Use **vector** and **string::c_str** to exchange data with non C++ APIs
  - Store only values and smart pointers in containers
  - Prefer **push_back** to other ways of expanding a sequnce
  - Prefer range operations to single element operations
  - Use the accepted idioms to really shrink capacity and really erase elements
  - Prefer algorithms to handwritten loops
  - Use the right STL search algorithm
  - Use the right STL sort algorithm

Do not write namespace **usings** in a header file or before an **#include**

Summary:

"Namespace **usings** are for your convenience, not for you to inflict on others: Never write a **using** declaration or a **using** directive before an **#include** directive."

Use **vector** by default

Summary:

"Using the *right* container is great: If you have a good reason to use a specific container type, use that container type knowing that you did the right thing.
So is using **vector**: Otherwise write **vector** and keep going without breaking stride, also knowing you did the right thing."

Use **vector** and **string** instead of arrays

Summary:

"Why juggle Ming vases? Avoid implementing array abstractions with C-style arrays, pointer arithmetic, and memory management primitives. Using **vector** or **string** only makes your life easier, but also helps you write safer and more scalable software."

Use **vector** and **string::c_str** to exchange data with non-C++ APIs (e.g. ROOT)

Summary:

"**vector** is not lost in translation: **vector** and **string::c_str** are your gateway to communicate with non-C++-APIs. But do not assume iterators are pointers; to get the address of an element referred to by **vector<T>::iterator iter** user **&*iter**."

Store only values and smart pointers in containers

Summary:

"Store objects of value in containers: Containers assume they contain value-like types, including value types(held directly), smart pointers, and iterators."

Prefer **push_back** to other ways of expanding a sequence

Summary:

"**push_back** all you can: If you do not need to care about the insert position, prefer using **push_back** to add an element to a sequence. Other means can be both vastly slower or less clear."

Prefer range operations to single element operations

Summary:

"Do not use oars when the wind is fair: When adding elements to sequence containers, prefer to use range operations (e.g. the form of **insert** that takes a pair of iterators) instead of a series of calls to the single-element form of the operation. Calling the range operation is generally easier to write, easier to read, and more efficient than an explicit loop."

Use the accepted idioms to really shrink capacity and really erase elements

Summary:

"Use a diet that works:...Use the erase-remove idiom."

Prefer algorithms to handwritten loops

Summary:

“Use function objects judiciously: For very simple loops. handwritten loops can be the simplest and most efficient solution. But writing algorithm calls instead of handwritten loops can be more expressive and maintainable, less error-prone, and as efficient.
When calling algorithms, consider writing your own custom function object that encapsulates the logic you need...”

Use the right STL search algorithm

Summary:

"To search an unsorted range, use **find**/**find_if** or **count**/**count_if**. To search a sorted range use **lower_bound**, **upper_bound**, **equal_range**, or (rarely) **binary_search**."

Use the right STL sort algorithm

Summary:

"Understand what each sorting algorithms does, and use the cheapest algorithm that does what you need."

Make predicates pure functions

Summary:

A predicate is a function object that returns a yes/no answer, typically as a **bool** value. A function is pure if its result depends only on its arguments. Do not allow predicates to hold or access state that affects the result of their **operator()**. Prefer to make **operator()** a **const** member function for predicates."

Prefer function objects over functions as algorithm and comparer arguments

Summary:

"Function objects are adaptable and, counterintuitively they typically produce faster code than functions."

Type Safety

avoid type switching, use polymorphism!

rely on types, not on representations!

avoid using **reinterpret_cast**!

avoid using **static_cast**!

avoid casting away **const**!

don't use C-style casts!

Type Safety

- don't **memcpy** or **memcmp** non-PODs!
- don't use varargs (ellipsis)!
- don't use invalid objects or unsafe functions!
- don't treat arrays polymorphically!