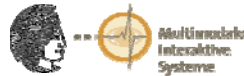


## Advanced Programming Techniques in Computer Vision

### Lecture 4 Temporaries, Inlining, STL 1 May 10, 2007

Gyuri Dorko  
Multimodal Interactive Systems

TU Darmstadt  
dorko@mis.tu-darmstadt.de



## Summary of last week



- Memory areas:
  - › Const data
  - › Stack
  - › Dynamic area (free store) for new/delete
  - › Dynamic area for malloc/free
  - › Global static
- Custom new and delete operators
- Special member functions
  - › constructors / destructors
  - › copy constructor / copy assignment
- Return Value Optimization
- In-place construction
- Smart pointers (e.g. `std::auto_ptr`)
- Memory leaks, memory related bugs, debugging, profiling (valgrind)

## This week



- Temporaries
- Inlining
- Standard Template Library (part 1)
- Debugging, profiling (part 2)
- Source code documentation
- Quiz 3 & 4
- Exercise sheet 4

## Temporaries



## Temporaries Introduction

Temporaries in C++ are invisible

May occur e.g.:

- Object definition
- Type mismatch (implicit conversion)
- Pass-by-value
- Return-by-value

T1: temporary to construct X(2)

T2: temporary to hold the result of f(X(2))

T3: temporary for the argument

T4: temporary for the return value

- T1 and T2 can be avoided by an implementation (compiler)
- Either T3 or T4 required a temporary to avoid aliasing of a

```
class X {
public:
    // ...
    X(int);
    X(const X&);
    ~X();
};
X f(X);
void g(){
    X a(1);
    X b = f( X(2) );
    a = f( a );
}
```

T2  
└───┬───┘  
T1  
└───┬───┘  
X(2)  
└───┬───┘  
T4  
└───┬───┘  
T3  
└───┬───┘  
a

## Temporaries

- Temporaries of class type are created when
  - binding an rvalue to a const reference
  - returning rvalue (see previous slide)
  - conversion that creates an rvalue
    - sizeof operator
    - casts
  - exception handling (e.g. throwing)
  - in initialization (e.g. constructor in an expression)

```
const double& rcd2 = 2; // temporary
const volatile int cvi = i;
const int& r = cvi; // error
```

## Return Value Optimization (RVO) - example (remember from last class)

```
struct X {
public:
    X();
    ~X();
    X(const X&);
    void fun();
};
X f(){
    X t;
    t.fun();
    return t;
}
X y = f();
```

There are two copy constructor calls here.

- Copying local automatic object t into a temporary object for the return value of function f
- Copy that temporary into object y

Don't forget: semantic restrictions must always be respected! (e.g. a copy constructor must be defined).

## Temporaries Initialization & Lifetime

- Temporaries (with non trivial constructors) will be initialized by calling their constructor. Similarly for destructors.
- Temporary objects are destroyed as the last step in evaluating the full-expression that contains the point they were created. Exceptions:
  - the temporary (or its copy) is used in initialization of an object,
  - the reference is bound to a temporary

Example

- T1 and T2 destroyed at the end of operator+
- T3 destroyed at end of cr's lifetime between obj2 and obj1

```
class C {
public:
    C(int);
    friend const C operator+(const C&, const C&);
    ~C();
};
C obj1;
const C& cr = C(16) + C(23);
C obj2;
```

T3  
└───┬───┘  
T1    T2  
└───┬───┘

## Prefer `op=` to stand alone `op`

```
class Number { ... };  
  
Number a, b, c, d, result;  
...  
result = a + b + c + d; // probably uses 3 temporary objects  
...  
result = a; // no temporary  
result += b; // no temporary  
result += c; // no temporary  
result += d; // no temporary
```

- This of course assumes that operator+ and operator+= are related.
- There are implementation that can avoid temporaries in the above case. (We discuss this in a later lecture.)

## Share initialization for constructors (do not create a temporary, it is wrong!)

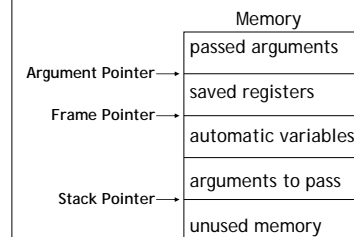
```
class Image{  
public:  
    // ...  
    Image(unsigned int w, unsigned int h);  
    Image(const char* s);  
};  
Image::Image(const char* s){  
    // ...  
    Image(wi dth, hei ght); // (1) do not EVER do this!  
    // ...  
};
```

- The line (1) above creates a temporary Image initializes with the Image(unsigned int, unsigned int) constructor and it immediately destroys it. Note, it is not possible with placement new operator either.
- To share code between constructors, e.g., use
  - default arguments
  - write a separate function

## Inlining

## Method invocation overhead

- Invocation
  - parameter marshalling
  - return address to the call register
  - invoked method saves SP, AP, and FP
  - invoked method saves all (approx. 3-4) registers that the function will use
- Return
  - usually two registers are used for the return parameter (easier to clean up)
  - restore registers
  - FP, AP are restored
  - SP adjusted (before the first passed argument)
  - return address placed in IP



## inline keyword & implicit inline



- inlining replaces method calls with macro-like expansions
- two mechanism to inline
  - › prefix the method definition with inline
  - › define the method within the declaration header
- both cases the method definition is in the header file (it must be available in every translation unit)
- calling of inline methods is like calling any other methods
- the keyword inline is only a suggestion to the compiler
- inline methods have no linkage requirements
- warnings
  - › set up your dependencies correctly in the makefile

## Advantages



- Inlining is one of the most efficient way to improve your program. It allows to remove function calls from the program fast-path
- may reduce code size
- Large part of the improvement is because “cross-call optimization” (compilers are good to optimize small code windows). Compare:

```
MyClass: g(){  
    // ...  
    int k = 0;  
    k = 10;  
    // ...  
}
```

```
MyClass: f(int i &){  
    // ...  
    i = 10;  
    // ...  
}  
  
MyClass: g(){  
    // ...  
    int k = 0;  
    ret = this->f(k);  
    // ...  
}
```

## Disadvantages



- increased compile time (overhead in development)
- may increase code size
- more cache misses
- inline methods must be in the headers (they must be defined in each translation unit)
- harder to debug & profile the program
- inlining constructors & destructors of base classes may cause weird things

## Inlining and Compilers



- Compiler options
  - › to turn inlining on, use: gcc -O3
  - › warn if a function cannot be inlined (but declared inline): -Winline
  - › controlling inline heuristics: -finline-limits=*n*
  - › see gcc documentation for more information

## When can/shall a function be inlined?

[Bulka & Mayhew, 1998]

Yes:

- the function has a small dynamic size and large invocation rate, especially on the fast path
- trivials (methods that are so small, they could never cause code expansion), very frequent in C++
- the function is called at one place (may or may not have invocation rate)

Dynamic Frequency	Static size		
	Large	Medium	Small
Low	Do not inline	Do not inline	Inline, if you have the time
Medium	Do not inline	Consider rewriting the method to expose its fast path and then inline	Always inline
High	Consider rewetting the method to expose its fast path and then inline	Selectively inline the high frequency static invocations	Always inline

## inline & virtual functions

- is possible when
  - an object (not a pointer) is used,
  - for pointers, even polymorphic, when the creation of the object, associated with the pointer is visible (can be determined compile time),
  - you have a sophisticated compiler.

```
struct C {  
    virtual int f() {  
        return data_;  
    }  
    int data_;  
};  
int main(){  
    C c;  
    C* pc = new C;  
    int i = c.f();  
    int j = pc->f();  
    //...  
}
```

## Standard Template Library - Part 1

- [S. Meyers, Effective STL, 2001]
- STL reference: <http://www.sgi.com/tech/stl>

## Recall: Asymptotic complexity

- sum-up all elements in a vector:  $O(N)$
- build a vector of size  $N$  inserting elements at the beginning:
  - overall shift of elements:  $(1+2+3+...N) = (N/2)(N+1) = N^2/2 + N/2$
  - $O(N^2)$
- the complexity of  $N^2$  and  $7*N^2$  is the same ( $O(N^2)$ ).
- terms:
  - constant time: complexity unaffected by  $N$  (e.g. accessing a vector element)
  - logarithmic time:  $O(\log N)$  (e.g., `set::find`)
  - linear time:  $O(N)$  (e.g. `count`)
- good approximation (good start)
- however, we do care about constant multipliers
- STL specifies complexity guarantees

## STL Basic terms



- Containers
  - standard sequence containers: vector, string, list, deque,
  - standard associative containers: set, multiset, map, multimap
  - non-standard containers: slist, rope, hash\_XXX, bitset
- Iterators (behave closely like pointers)
  - input iterators: read-only
  - output iterators: write-only
  - forward iterators: input and output, but can only move forward
  - bidirectional iterators: just like forward, but they can move also backward
  - random access iterators: bidirectional + offer iterator arithmetic
- Functors: classes that overload operator() (see also Lecture 1)

## STL Containers



- Contiguous-memory containers: vector, string, deque
- Node-based containers: list, slist, all associative containers
- How to decide
  - Do you need to insert a new element at arbitrary position? (often on fast path) YES: use sequence containers
  - Do you care how elements are ordered? NO: hashed containers
  - Must the container be part of the standard?
  - What category of iterators do you need?
  - Is it important to avoid movement of existing container elements when insertion or erasures take place?
  - Do you need layout compatibility with C?
  - Is lookup speed critical (i.e. on the fast path)?
  - ...
- If you are unsure think about using vector.

## Example: std::vector (dynamic array)



Note: this does not meant to be a meaningful code, just demonstrates some functionalities of std::vector

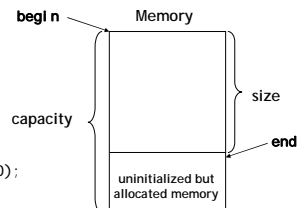
```
vector<int> v;
v.push_back(10);
v.push_back(20);
int j = v[1]; // j==20

vector<int> v2(20, 5); // 20 elements initialized to 5
unsigned int s = v2.size();
bool e = v2.empty(); // will be false

v2.reserve(100); // set capacity (make place for new elements)
swap(v, v2); // very fast
vector<int>(v).swap(v); // trim excess capacity

int sum = 0;
vector<int>::const_iterator i = v2.begin();
while(i != v2.end()){
    sum += *i;
    ++i;
}
int sum2 = accumulate(v2.begin(), v2.end(), 0);

myCFuncti on(&v2[0], v2.size());
```

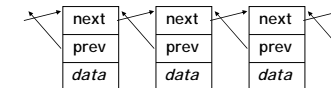


## Example: std::list (double-linked list)



Note: this does not meant to be a meaningful code, just demonstrates some functionalities of std::list

```
list<int> l;
l.push_back(20);
l.push_front(10);
list<int>::const_iterator i = l.begin();
++i;
int j = *i; // j==20
```



```
list<int> l2(20, 5); // 20 elements initialized to 5
unsigned int s = l2.size(); // can be 0(N)
bool e = l2.empty(); // will be false, it is always 0(1)

swap(l, l2); // very fast
l1.reverse(); // 0(N), all iterators stay valid

int prod = accumulate(l2.begin(), l2.end(), 1, multiplies<int>());

// move all nodes in l2 from the first occurrence of 5
// thought the last occurrence of 10 to the end of l
l.splice(l.end(), l2,
         find(l2.begin(), l2.end(), 5),
         find(l2.rbegin(), l2.rend(), 10).base());
```

## Example: std::map (balanced binary tree)

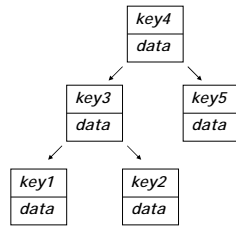
```
map<string, int> m;
m["january"] = 31;
m["february"] = 28;
m.insert(make_pair(string("march"), 20));
m["march"] = 31;
```

```
...
string s = "june"
cout << s << " has " << m[s] << " days\n";
...
```

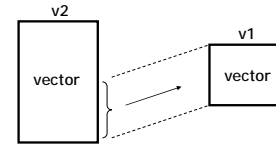
```
unsigned int s = m.size();
bool e = m.empty();
```

```
struct lstr : std::binary_function<const char*, const char*, bool> {
    bool operator() (const char* s1, const char* s2) const {
        return strcmp(s1, s2) < 0;
    }
}; // this should define strict weak ordering as std::less<Key> does
```

```
map<const char*, int, lstr> m;
```



## Range functions - Example 1



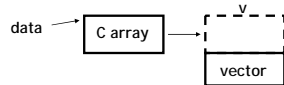
```
Solution 1
v1.clear();
for(vector<int>::const_iterator i=v2.begin()+v2.size()/2; i!=v2.end(); ++i)
    v1.push_back(*i);
```

```
Solution 2
v1.clear();
copy(v2.begin() + v2.size()/2, v2.end(), back_inserter(v1));
```

```
Solution 3
v1.clear();
v1.insert(v1.end(), v2.begin() + v2.size()/2, v2.end());
```

```
Solution 4
v1.assign(v2.begin()+v2.size()/2, v2.end()); // BEST
```

## Range function - Example 2



```
Solution 1
int data[N];
vector<int> v;
...
vector<int>::iterator loc = v.begin();
for(int i=0; i<N; ++i){
    loc = v.insert(loc, data[i]);
    ++loc;
}
```

```
Solution 2
int data[N];
vector<int> v;
...
copy(data, data+N, inserter(v, v.begin()));
```

```
Solution 3
int data[N];
vector<int> v;
...
v.insert(v.begin(), data, data+N); // BEST
```

## Range Functions

- Range Constructors

```
int data[N];
...
vector<int> v(data, data+N);
```

- Range inserts (see previous slide)

- Range erase

- sequences: returns the element following the one that was erased

- Range assignment (2 slides before)

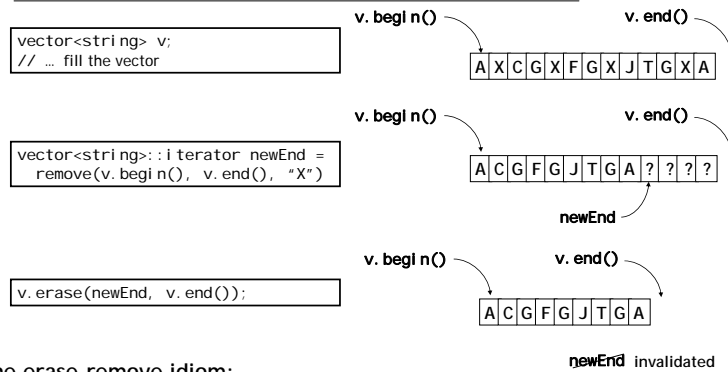
## Always use (look for) member functions when possible



- If a member method is available, it has a reason, a specialized version is implemented because
  - the general implementation does not work on that type of container
  - there is a more efficient way to implement it

```
list<int> l;  
...  
l.sort(); // N log N stable sort written for lists (std::sort relies on random access iterators)  
multimap<string, int> m;  
...  
unsigned int n = m.count("A110"); // more efficient than std::count()
```

## Removing elements from a vector



## Removing elements from a list



- The erase-remove idiom works, but it is not efficient.
- Use list's member function instead

```
l.erase("X");
```

## Removing elements from a map



- Remove element with a given key:

```
map<string, int> m;  
...  
m.erase("X");
```

- Remove elements with a given value:

```
map<int, string> m;  
map<int, string>::iterator i = m.begin()  
while(i != m.end()){  
    if(i->second=="X")  
        m.erase(i++); // postfix increment is a must  
    else  
        ++i;  
}
```

## Sorting ranges



- **stable\_sort**: sort a full range, preserves relative ordering of equivalent elements
- **sort**: sort a full range
- **partial\_sort**: the “best” n elements are in the right order at the beginning of the range
- **nth\_element**: the nth “best” element is the “nth” in the range and the “best” n elements are the first n elements at the beginning (not necessarily in the right order)
- **stable\_partition**: all elements satisfy a condition is in the beginning of the range, preserves relative ordering among those that satisfy or do not satisfy the condition
- **partition**: all elements satisfy a condition is in the beginning of the range

Random Access Iterators  
Bidirectional Iterators

## Example: Sorting

```
bool nameCompare(const Person& lhs, const Person& rhs){
    return lhs.getName() > rhs.getName();
}

vector<Person> v;
... // fill vector v
partial_sort(v.begin(), v.end(), nameCompare);
```

```
struct nameCompare : std::binary_function<Person, Person, bool>{
    bool nameCompare(const Person& lhs, const Person& rhs) const{
        return lhs.getName() > rhs.getName();
    }
}; // this is an adaptable binary predicate
vector<Person> v;
... // fill vector v
sort(v.begin(), v.end(), not2(nameCompare()));
```

## Example: Sorting

```
class IsYoung : public unary_function<Person, bool>{
public:
    IsYoung(int a):oldage(a) {}
    bool operator()(const Person& p) const {
        return p.getAge() < oldage;
    }
private:
    int oldage;
};

vector<Person>::iterator youngEnd =
    partition(person.begin(), person.end(), IsYoung(50));
sort(person.begin(), youngEnd, nameCompare());
...
```

```
vector<A> v;
...
sort(v.begin(), v.end());
typedef vector<A>::iterator VIter;
typedef pair<VIter, VIter> VIPair;
A a(...);
VIPair p = equal_range(v.begin(), v.end(), a);
cout << "There are " << distance(p.first, p.second) << " elements equi v. to a\n";
```

## Searching [Meyers, 2001]

What do you want to know?	Unsorted range	Sorted range	std::set std::map member functions	std::multiset std::multimap member functions
Value exist?	find	binary_search	count	find
Exists & where is the first one?	find	equal_range	find	find (for any) or lower_bound
The value not preceding the desired value.	find_if	lower_bound	lower_bound	lower_bound
The value succeeding the desired value	find_if	upper_bound	upper_bound	upper_bound
How many of a given value?	count	equal_range	count	count
Where are all the desired values?	find (in a loop)	equal_range	equal_range	equal_range



## Code documentation

- Several tools available.
- One of the most popular GPL tool is doxygen (<http://www.doxygen.org>)

```
/**
 * \brief The main Image class.
 * The Image class stores gray scale images. All pixel values treated as float.
 */
class Image {
public:
    /// create an "empty image"
    Image();

    /** \brief Create an image with dimensions wxh
     * the long description
     * \param w the image width
     * \param h the image height
     */
    Image(unsigned int w, unsigned int h);

    /// Returns the width of the image
    unsigned int getWidth() const;

    /// Returns the height of the image
    unsigned int getHeight() const;
    ...
private:
    float* data_ //!< this is where all the pixels are
    ...
}
```