Optimization

Motivation

- Examine some other optimization patterns
- Several C++ specific techniques

Containers

- Nearly all programs must manipulate data
- That means we need to store it in a data structure
- Choice of container can have significant impact on overall program time

Containers

- Basic container: vector<>
  - Indexing with [] or at() : O(1)
  - Append with push_back: *amortized constant time*: O(1)
  - Insert in middle/remove from middle: O(n)

Containers

- Linked list: list<>
  - No indexing possible
  - Append/insert/remove: O(1), all cases

Containers

- Double ended queue: deque
- How it’s stored internally: Like vector<vector<>>
  - Outer vector contains more vectors
  - Inner vectors contain the data items
- This leads to unique time characteristics...

Deque

- Prepending or appending is O(1)
- Access (with []) is O(1)
  - But larger hidden constant than vector<>
- Insert is O(n/2) = O(n)

Example

- Suppose we are getting collection of data items (maybe from network)
- We want to store them in reversed order
- For benchmarking, we'll use block of memory with data items and just time the container insertion operation
- Code: reverse.cpp

Results

```
$ ./a.out 10
Times: vector = 2 usec, deque = 3 usec
$ ./a.out 100
Times: vector = 4 usec, deque = 1 usec
$ ./a.out 1000
Times: vector = 159 usec, deque = 19 usec
$ ./a.out 10000
Times: vector = 4946 usec, deque = 52 usec
$ ./a.out 100000
Times: vector = 840208 usec, deque = 255 usec
```

Question

- What about appending?
- append.cpp

Results

```
$ ./a.out 10
Times: vector = 1 usec, deque = 4 usec
$ ./a.out 100
Times: vector = 1 usec, deque = 3 usec
$ ./a.out 1000
Times: vector = 13 usec, deque = 23 usec
$ ./a.out 10000
Times: vector = 40 usec, deque = 49 usec
$ ./a.out 100000
Times: vector = 361 usec, deque = 404 usec
$ ./a.out 1000000
Times: vector = 3605 usec, deque = 4254 usec
```

Insert

- What about inserting in the middle?
- middle.cpp

Results

```
$ ./a.out 10
Times: vector = 3 usec, deque = 2 usec
$ ./a.out 100
Times: vector = 4 usec, deque = 10 usec
$ ./a.out 1000
Times: vector = 102 usec, deque = 376 usec
$ ./a.out 10000
Times: vector = 1536 usec, deque = 4856 usec
$ ./a.out 100000
Times: vector = 366951 usec, deque = 533487 usec
```

Parameters

- Acccidental copying of function parameters can consume considerable time

```
void foo( BigThing b ){
    ...
}
```

Solution?

- We can avoid overhead of copiesby using references:

```
void foo( BigThing& b ){
    ...
}
```

- But what if someone accidentally tries to copy object with =?

C++ 11

- C++ 11 provides *deleted* functions

```
class Foo{
    public:
    void operator=(const Foo& x) = delete;
    Foo(const Foo&) = delete;
};
```

Benefit

- This catches several problems:
  - Pass by value
    ```
    void foo(BigThing b){ ... }
    ```
  - Use of =
    ```
    BigThing b2 = b;
    ```
  - Use of copy constructor
    ```
    BigThing b2(b);
    ```
  - Return by value
    ```
    BigThing foo(){ ... }
    ```

Loop Tests

- Consider this code:

```
char x[512];
...put data in x...
for(auto i=0;i<strlen(x);i++){
    ...examine x[i]...
}
```

- What’s the problem?

Problem

- strlen is O(n)
- Called on every loop iteration
- If length of x not changed in loop, this is wasteful

Better

- Compute length once, cache it

```
char x[512];
...put data in x...
auto len = strlen(x);
for(suto i=0;i<len;i++){
    ...examine x[i]...
}
```

Or

- Do the count the other way:

```
char x[512];
...put data in x...
for(auto i=strlen(x)-1;i>=0;i--){
    ...examine x[i]...
}
```

- This code is broken. Do you know why?

Problem

- Consider: for(auto i=strlen(x);i>=0;i--)
- What type is i?

Type

- strlen defined to return size_t
- size_t is typically either uint32_t or uint64_t
- So test is like: for(unsigned i=strlen(x)-1;i>=0;i--)
- When does the termination condition get triggered?

Problem

- Never!
- On last iteration of loop: i=0
- Then i-- occurs
- i wraps to INT_MAX
- And loop keeps going!

Solution

- Need to use signed type here
- But: If string is very long (2GB+), ’i’ will be initialized to negative value

Note

- Same problem can occur in other contexts
- Ex:

```
vector<Foo> v;
...put stuff in v...
for(auto i=v.size()-1;i>=0;i--){
    ...
}
```

Idea

- Since loops run many times, they are suspect for code hot spots
- One rule of thumb: Avoid function calls in loop
- Some are not obvious

Question

- How many function calls are made?

```
for(int i=0;i<100;++i){
    Foo f;
}
```

Question

- How many function calls are made?

```
for(int i=0;i<100;++i){
    Foo f;
}
```

- 200!
  - Constructor runs at top of loop
  - Destructor runs at bottom

Example

- Example (based on one in Guntheroth):
- This is suboptimal: Lots of constructor/destructor calls:

```
for(...){
    string x("something");
    ...code that changes x...
}
```

Improved

- We can improve this by taking advantage of the string class's abilities:

```
string x;
for(...){
    x.clear()
    x+="...";
    ...code that changes x...
}
```

Loops

- Be aware of overloaded operators
  - +,-,*,/, etc.
  - Operations can be more costly than they look
- Ex:

  ```
  mat4 A = ...;
  mat4 B = ...;
  mat4 C = A*B;
  ```
- The matrix-matrix multiply is much more expensive than a single scalar multiply

Organization

- Instead of calling a function in a loop, it can be better to perform a loop in a function
- Ex: Suboptimal:

```
vector<...> foo;
for(i=0;i<foo.size();i++){
    func(foo[i]);
}
Better:
funcAll(foo);
void funcAll(vector<...>& foo ){
    for(i=0;i<foo.size();i++){
        ...process foo[i]...
    }
}
```

Explanation

- Why is first one potentially slower?
  - We have function call overhead on every loop iteration
  - Second form has only one function call

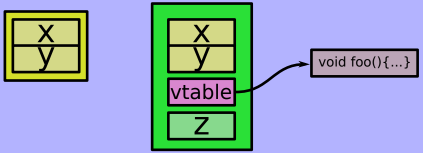
Classes

- Consider inheritance:

```
class Base{
    int x,y;
};
class Der : public Base {
    int z;
    virtual void foo(){
    }
};
```

Vtable

- If class contains virtual functions, it must have a vtable
- Organization of Base and Der:



Example

- Suppose we have:
  Der* d = ...;
  d->foo();
- System must get value in d (the address), add sizeof(Base) to it, then do indirect function call on result
  - This can be slower

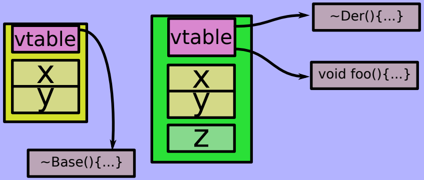
Better

```
class Base{
    int x,y;
    virtual ~Base(){}
};
class Der : public Base {
    int z;
    virtual void foo(){
    }
};
```

- Any function that’s going to be the base of inheritance chain should have virtual destructor anyway

Result

- Organization of memory:



- Now, vtable is at start of object
  - Saves an addition upon function call

Multiple Inheritance

- C++ also allows multiple inheritance:

  ```
  class Foo : public Bar, public Baz {
      ...
  };
  ```

- Problem: If Bar and Baz both have virtual functions, Foo has multiple vtables
  - This results in the same issue with needing to add an offset to pointer location to get function addresses
  - So multiple inheritance can be (a bit) slower to use

Inlining

- In C++, if function body inside class declaration, it's usually inlined
  - Saves function call/return time
  - Might help optimize register usage
  - But: Virtual functions normally cannot be inlined
- So: Virtual functions can have additional cost here too
- But: Series of switch/if-else can be slower than virtual function call

Sources

- Herb Sutter. GotW: Compilation Firewalls. https://herbsutter.com/gotw/_100/
- https://stackoverflow.com/questions/22306949/does-deque-provide-o1-complexity-when-inserting-on-top
- https://stackoverflow.com/questions/7572529/complexity-of-stl-dequeinsert
- https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org