

# Thread Safety

# Sean Parent

Principal Scientist  
Software Technology Lab

# September 11<sup>th</sup>, 2008



# Outline of Talk

- Why Concurrent Code?
- What is Thread Safe Code?
  - Classes without synchronization
- Synchronization:
  - Memory Hierarchy, Compiler Reordering, Execution Reordering and Memory Barriers
  - Memory, Atomics, and Locks in Multi-Processor World
- Problems with Locks:
  - Deadlocks, Race Conditions, Starvation
- Common Patterns of Sharing:
  - Singletons
  - Reference Counting and Copy-On-Write

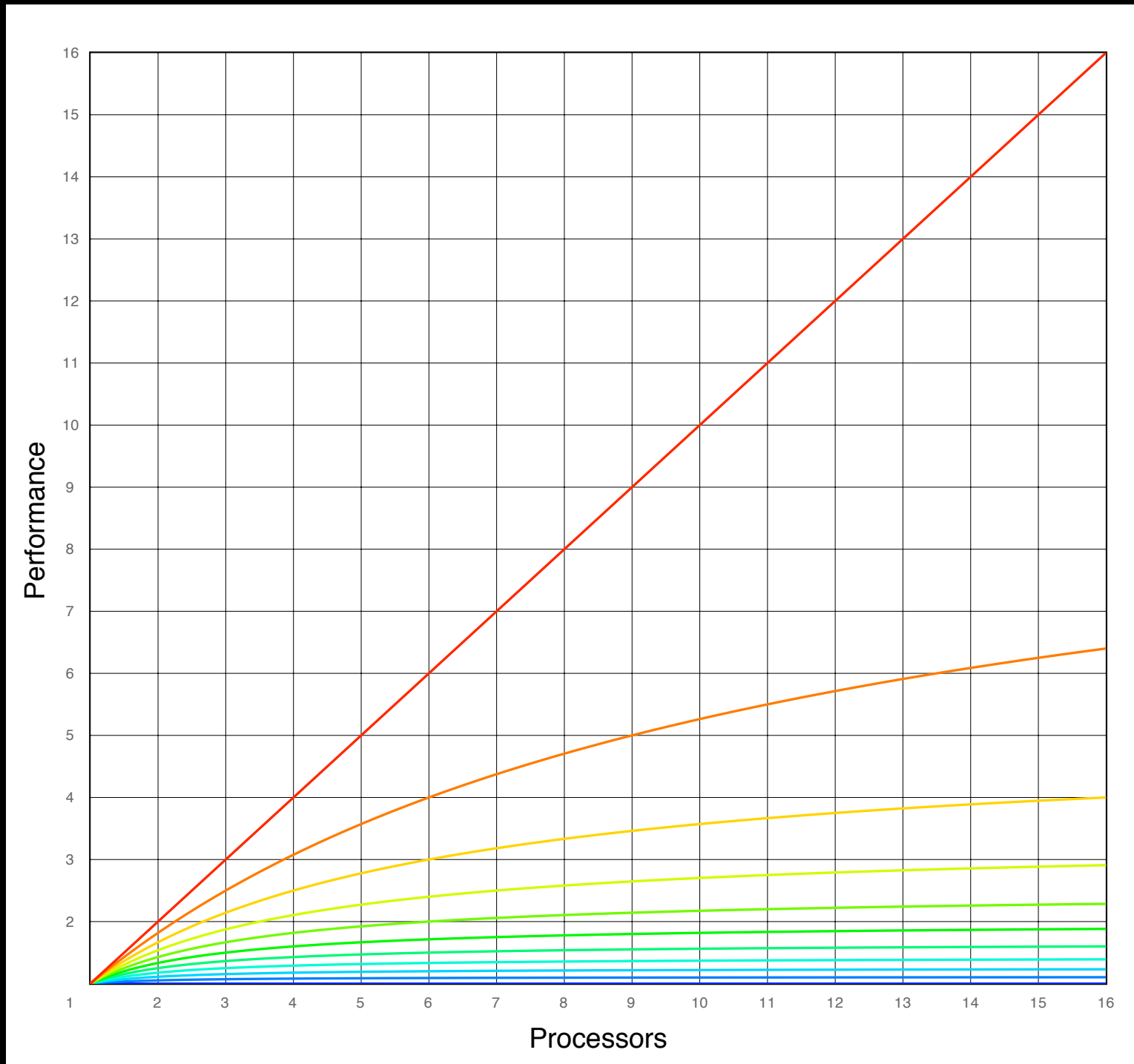
# Why Concurrent Code?

- Clock speeds getting faster at a slower rate
- Number of transistors continuing to double every 18-24 months
- Result is *rapid* growth in number of cores
- Except:
  - Few applications scale well beyond 2-4 cores (most don't scale beyond 1)
  - Very little system gain beyond 8 cores
  - Amdahl's Law\* still hurts
  - Memory has not sped up as fast as cores

\* Amdahl's law states that if  $P$  is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and  $(1 - P)$  is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using  $N$  processors is:

$$\frac{1}{(1-P) + \frac{P}{N}}$$

# Amdahl's Law



# Why Concurrent Code?

- There is no concurrent *silver bullet*
- For Performance:
  - Optimize algorithm complexity
  - Optimize memory access and utilization
  - Optimize I/O (including asynchronous read ahead and write behind)
  - Consider parallel algorithms - the closer P is to 1, the higher the gains
- For Interactivity:
  - Use work queues and execute them on your event loop
  - Consider multiple processes (see Google Chrome)
  - Consider concurrent tasks
- Profile and Measure Response Times
  - Let hard data guide your optimization efforts
  - Concurrency is one tool at your disposal

# What is Thread Safe Code?

- Thread safe code is code which remains correct when executed concurrently
  - But that isn't the whole story...
- A function is thread safe if it remains correct when executed concurrently *assuming any modified objects are not shared or access is synchronized*
- A type is thread safe if the basis operations on the type are thread safe functions
  - Thread safety does not imply a single instance can be shared by multiple threads without synchronization
  - `int` is a thread safe type, `std::vector<int>` is a thread safe type
  - `shared_ptr<int>` is not thread safe
  - `shared_ptr<const int>` is thread safe
- A type is sharable if the basis operations on the type are thread safe functions when operating on a shared instance

# Key to Thread Safety

- All code should be written thread safe
  - The once task based threading is introduced to an application, it becomes difficult to control what code is and is not execute on a thread
- Reduce sharing of mutable data to reduce synchronization - *any* synchronization will fall victim to Amdahl's law given enough processors
  - This is why functional programming and Haskell continue to gain ground - eliminating mutable data is one way to eliminate the sharing of mutable data
- There is often a cost to eliminating sharing and a tradeoff in the performance of code when executed linearly against the ability to scale to many processors
  - Look for opportunities to leverage concurrency to offset the costs - typically, just getting to leverage two processors will more than offset these costs

# Classes Without Synchronization

- OOP has indirectly led to very thread unsafe code -
- The requirement of a polymorphic type, by definition, comes from it's use -
  - That is, there is no such thing as a polymorphic type, only a polymorphic use of *similar* types
- By using inheritance to capture polymorphic use, we shift the burden of use to the type, tightly coupling components
- Inheritance implies variable size, which implies heap allocation
- Heap allocation forces a further burden on use to manage the object lifetime
- Object lifetime management leads to garbage collection or reference counting
- This encourages *shared* ownership and the proliferation of *incidental data-structures*
- Shared mutable objects are not thread safe without synchronization



# Disclaimer

- In the following code, the proper use of header files, inline functions, and namespaces are ignored for clarity

client

library

cout

guidelines

defects

client

library

```
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

cout

**Hello World!**

client

library

cout

guidelines

defects

```
typedef int object_t;

void print(ostream& out, const object_t& x) { out << x << endl; }

typedef vector<object_t> document_t;

void print(ostream& out, const document_t& x)
{
    out << "<document>" << endl;
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {
        print(out, *f);
    }
    out << "</document>" << endl;
}
```

client

library

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects



client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

**<document>**

**0**

**1**

**2**

**3**

**</document>**

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

## guidelines

- Write all code as a library.
  - Reuse increases your productivity.
  - Writing unit tests is simplified.

client

library

```
typedef int object_t;
```

```
void print(ostream& out, const object_t& x) { out << x << endl; }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)
```

```
{
```

```
    out << "<document>" << endl;
```

```
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);
```

```
    }
```

```
    out << "</document>" << endl;
```

```
}
```

cout

guidelines

defects

client

library

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)  
{
```

```
    out << "<document>" << endl;
```

```
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);
```

```
    }
```



cout

guidelines

defects

```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const int& x) : object_m(x)  
    { }  
  
    friend void print(ostream& out, const object_t& x);  
private:  
    int object_m;  
};
```

```
void print(ostream& out, const object_t& x)  
{ print(out, x.object_m); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)  
{  
    out << "<document>" << endl;  
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);  
    }  
}
```



client

library

```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const int& x) : object_m(x)  
    { }  
  
    friend void print(ostream& out, const object_t& x);  
private:  
    int object_m;  
};
```

```
void print(ostream& out, const object_t& x)  
{ print(out, x.object_m); }
```

```
typedef vector<object_t> document_t;
```

guidelines

```
void print(ostream& out, const document_t& doc)
```

- The compiler will supply member-wise copy and assignment operators.
  - Let the compiler do the work where appropriate.

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

```
<document>
0
1
2
3
</document>
```



client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

guidelines

- Write classes that behave like *regular* objects to increase reuse.

client

library

```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:
```

```
    explicit object_t(const int& x) : object_m(x)  
    { }
```

```
    friend void print(ostream& out, const object_t& x);
```

```
private:
```

```
    int object_m;
```

```
};
```

```
void print(ostream& out, const object_t& x)  
{ print(out, x.object_m); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)  
{
```

```
    out << "<document>" << endl;
```

```
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);
```

```
    }
```



cout

guidelines

defects

client

library

+

```
class object_t {  
public:  
    explicit object_t(const int& x) : object_m(x)  
    { }  
  
    friend void print(ostream& out, const object_t& x);  
private:  
    int object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ print(out, x.object_m); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)  
{  
    cout << "<document>" << endl;  
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);  
    }  
    cout << "</document>" << endl;
```

+

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(x)  
        { }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        int object_m;  
};
```

```
void print(ostream& out, const object_t& x)  
{ print(out, x.object_m); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)  
{  
    cout << "<document>" << endl;  
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, *f);  
    }  
    cout << "</document>" << endl;  
}
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:
```

+

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)
```

+

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }
```

```
friend void print(ostream& out, const object_t& x);  
private:  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    int_model_t* object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }  
  
typedef vector<object_t> document_t;
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }  
  
typedef vector<object_t> document_t;
```

cout

guidelines

defects



client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

guidelines

- Do your own memory management - don't create garbage for your client to clean-up.

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }
```



```
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }
```



cout

guidelines

defects

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }
```

```
    friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }
```

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }
```

```
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }
```

```
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };
```

guidelines

- The semantics of copy are to create a new object which is equal to, and logically disjoint from, the original.
- Copy constructor must copy your object. The compiler is free to elide copies so if your copy constructor does something else your code is incorrect.
- When a type manages *remote parts* it is necessary to supply your own copy constructor.
  - If you can, use an existing class (such as a vector) to manage remote parts.

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }
```

```
friend void print(ostream& out, const object_t& x);  
private:  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    int_model_t* object_m;  
};
```

cout

guidelines

defects

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { delete object_m; object_m = new int_model_t(*x.object_m); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { delete object_m; object_m = new int_model_t(*x.object_m); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        }  
};
```

guidelines

- Assignment is consistent with copy. Generally:  
 $T\ x; x = y;$  is equivalent to  $T\ x = y;$

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects



client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

**<document>**

**0**

**1**

**2**

**3**

**</document>**

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

## guidelines

- The Private Implementation (Pimpl), or Handle-Body, idiom is good for separating the implementation and reducing compile times.

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { delete object_m; object_m = new int_model_t(*x.object_m); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { delete object_m; object_m = new int_model_t(*x.object_m); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        }.  
};
```

defects

- If new throws an exception, the object will be left in an invalid state.
- If we assign an object to itself this will crash.

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)
```

```
friend void print(ostream& out, const object_t& x);  
private:  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    int_model_t* object_m;  
};
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        }  
};
```

## guidelines

- Assignment should satisfy the *strong exception guarantee*.
  - Either complete successfully or throw an exception, leaving the object unchanged.
- Don't optimize for rare cases which impact common cases.
  - Don't test for self-assignment to avoid the copy.

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```



```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

public:

```
explicit object_t(const int& x) : object_m(new int_model_t(x))
```

```
{ }
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
```

```
{ }
```

```
object_t& operator=(const object_t& x)
```

```
{ object_t tmp(x); swap(*this, tmp); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

private:

```
struct int_model_t {
```

```
    explicit int_model_t(const int& x) : data_m(x) { }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;
```

```
};
```

```
int_model_t* object_m;
```

```
};
```

```
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void print(ostream& out, const object_t& x)
```

```
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void print(ostream& out, const object_t& x)
{ x.object_m->print_(ostream& out); }
```

```
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void print(ostream& out, const object_t& x)
{ x.object_m->print_(ostream& out); }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void print(ostream& out, const object_t& x)
{ x.object_m->print_(ostream& out); }

typedef vector<object_t> document_t;
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void print(ostream& out, const object_t& x)
{ x.object_m->print_(ostream& out); }

typedef vector<object_t> document_t;
```

client

library

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};
```

cout

guidelines

defects



client

library

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};
```

```
void swap(object_t& x, object_t& y)
{
    swap(x.object_m, y.object_m);
    cout << "swap" << endl;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));
```

```
print(cout, document);
return 0;
```

```
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    reverse(document.begin(), document.end());

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    reverse(document.begin(), document.end());

    print(cout, document);
    return 0;
}
```

cout

```
swap
swap
<document>
3
2
1
0
</document>
```

## client

## library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    reverse(document.begin(), document.end());

    print(cout, document);
    return 0;
}
```

## guidelines

- Provide a non-throwing swap function in the same namespace as the class.
- Call swap unqualified allowing argument dependent lookup (ADL) to find the best swap.
- The standard library treats swap as an operator and calls it unqualified.\*

*\*Microsoft Visual C++ does not currently make unqualified calls to swap.*

client

library

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
{
    swap(x.object_m, y.object_m);
    cout << "swap" << endl;
}
```

cout

guidelines

defects

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
{
    swap(x.object_m, y.object_m);
}
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
{
    swap(x.object_m, y.object_m);
}
```



```
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
{
    swap(x.object_m, y.object_m);
}
```

```
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
{
```

```
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(const object_t& x)
{ object_t tmp(x); swap(*this, tmp); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
};

int_model_t* object_m;
};

void swap(object_t& x, object_t& y)
```



public:

```
explicit object_t(const int& x) : object_m(new int_model_t(x))  
{ }
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }
```

```
object_t& operator=(const object_t& x)  
{ object_t tmp(x); swap(*this, tmp); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

private:

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
int_model_t* object_m;
```

```
};
```



```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
```

```
object_t& operator=(const object_t& x)  
{ object_t tmp(x); swap(*this, tmp); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
int_model_t* object_m;  
};
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { cout << "ctor" << endl; }  
        ~object_t() { delete object_m; }
```

```
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { cout << "copy" << endl; }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }
```

```
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }
```

```
        int data_m;
```

```
    };  
  
    int_model_t* object_m;
```

```
};
```

cout

guidelines

defects

client

library

cout

guidelines

defects



client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
       Quiz: What will this print?
    */

    object_t x = function();

    return 0;
}
```

cout

guidelines

defects

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
     * Quiz: What will this print?
     */

    object_t x = function();

    return 0;
}
```

cout

ctor

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
```

{

/\*

## Quiz: What will this print?

**\* /**

```
return 0;
```

}

cout

## guidelines

# defects

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
     * Quiz: What will this print?
     */
```

```
    object_t x(0);
```

```
    x = function();
```

```
    return 0;
```

```
}
```

cout

guidelines

defects

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
     * Quiz: What will this print?
     */

    object_t x(0);

    x = function();

    return 0;
}
```

cout

**ctor**  
**ctor**  
**copy**

client


library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { cout << "ctor" << endl; }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { cout << "copy" << endl; }  
        object_t& operator=(const object_t& x)  
        { object_t tmp(x); swap(*this, tmp); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct int_model_t {  
            explicit int_model_t(const int& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            int data_m;  
        };  
  
        int_model_t* object_m;  
};
```

cout

guidelines

defects

```
class object_t {  
public:  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { cout << "ctor" << endl; }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { cout << "copy" << endl; }  
  
    
```

```
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    int_model_t* object_m;  
};
```



```
class object_t {
public:
    explicit object_t(const int& x) : object_m(new int_model_t(x))
    { cout << "ctor" << endl; }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
    { cout << "copy" << endl; }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        int data_m;
    };

    int_model_t* object_m;
};
```



client

library

```
class object_t {
public:
    explicit object_t(const int& x) : object_m(new int_model_t(x))
    { cout << "ctor" << endl; }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
    { cout << "copy" << endl; }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct int_model_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_ostream(ostream& out) const { print(out, data_m); }
    };
};
```

guidelines

- Pass *sink* arguments by value and swap or move into place.
- A sink argument is any argument consumed or returned by the function.
  - The argument to assignment is a sink argument.

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
     * Quiz: What will this print?
     */

    object_t x(0);

    x = function();

    return 0;
}
```

cout

guidelines

defects

client

library

```
object_t function() { return object_t(5); }
```

```
int main()
{
    /*
     * Quiz: What will this print?
     */

    object_t x(0);

    x = function();

    return 0;
}
```

cout

ctor  
ctor

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { cout << "ctor" << endl; }  
        ~object_t() { delete object_m; }
```

```
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { cout << "copy" << endl; }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }
```

```
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }
```

```
        int data_m;
```

```
    };  
  
    int_model_t* object_m;
```

```
};
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
int_model_t* object_m;  
};
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
    ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }
```

```
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }
```

```
        int data_m;  
    };  
  
    int_model_t* object_m;
```

```
};
```

cout

guidelines

defects

client

library

```
class object_t {  
public:
```

```
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:
```

cout

guidelines

defects

client

library

```
class object_t {  
    public:  
        explicit object_t(const string& x) : object_m(new string_model_t(x))  
        { }  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct string_model_t {  
            explicit string_model_t(const string& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            string data_m;  
        };  
};
```

cout

guidelines

defects



```
class object_t {  
    public:  
        explicit object_t(const string& x) : object_m(new string_model_t(x))  
        { }  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct string_model_t {  
            explicit string_model_t(const string& x) : data_m(x) { }  
            void print_(ostream& out) const { print(out, data_m); }  
  
            string data_m;  
        };  
};
```



public:

```
explicit object_t(const string& x) : object_m(new string_model_t(x))  
{ }
```

```
explicit object_t(const int& x) : object_m(new int_model_t(x))  
{ }
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

private:

```
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;  
};
```

```
struct int_model_t {
```



```
explicit object_t(const string& x) : object_m(new string_model_t(x))
{ }
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct string_model_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
```



```
{ }  
explicit object_t(const int& x) : object_m(new int_model_t(x))  
{ }  
~object_t() { delete object_m; }  
  
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
};
```





```
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct string_model_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```

```
struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }
```





```
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct string_model_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
```





```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

private:

```
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;  
};
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```



```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct string_model_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
};
```



```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct string_model_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
};

int_model_t* object_m;
```

client

library

```
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct string_model_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };  
  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    int_model_t* object_m;  
};
```

cout

guidelines

defects

client

library

+

```
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;  
};
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
int_model_t* object_m;
```

```
};
```

+

cout

guidelines

defects

client

library

+

```
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;  
};
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
};
```

+

cout

guidelines

defects

client

library

+

```
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct string_model_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;  
};
```

```
struct int_model_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;  
};
```

```
concept_t* object_m;
```

```
};
```

+

cout

guidelines

defects

client

library



```
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct string_model_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };  
  
    struct int_model_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    concept_t* object_m;  
};
```



cout

guidelines

defects

client

library

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:
```

```
explicit string_model_t(const string& x) : data_m(x) { }  
void print_(ostream& out) const { print(out, data_m); }
```

```
string data_m;  
};
```

```
explicit int_model_t(const int& x) : data_m(x) { }  
void print_(ostream& out) const { print(out, data_m); }
```

```
int data_m;  
};
```

```
concept_t* object_m;  
};
```

cout

guidelines

defects

client

library



```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:
```

```
struct concept_t {  
    virtual ~concept_t() { }  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```

```
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};
```

```
concept_t* object_m;  
};
```



cout

guidelines

defects



```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};  
  
concept_t* object_m;
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };  
  
    struct string_model_t : concept_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };  
  
    struct int_model_t : concept_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    concept_t* object_m;
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};  
  
concept_t* object_m;
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
};
```



```
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct concept_t {
```

```
    virtual ~concept_t() { }
```

```
    virtual void print_(ostream& out) const = 0;
```

```
};
```

```
struct string_model_t : concept_t {
```

```
    explicit string_model_t(const string& x) : data_m(x) { }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;
```

```
};
```

```
struct int_model_t : concept_t {
```

```
    explicit int_model_t(const int& x) : data_m(x) { }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;
```

```
};
```





```
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```

```
struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
```





```
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
};
```



```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
        explicit string_model_t(const string& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        string data_m;
    };

    struct int_model_t : concept_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }
```



```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
        explicit string_model_t(const string& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        string data_m;
    };

    struct int_model_t : concept_t {
        explicit int_model_t(const int& x) : data_m(x) { }
```



```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```

```
struct int_model_t : concept_t {
```





```
{ }  
~object_t() { delete object_m; }  
  
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };  
  
    struct string_model_t : concept_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };
```





```
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```





```
{ }
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
```





```
explicit object_t(const string& x) : object_m(new string_model_t(x))
{ }
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }
```





public:

```
explicit object_t(const string& x) : object_m(new string_model_t(x))  
{ }
```

```
explicit object_t(const int& x) : object_m(new int_model_t(x))  
{ }
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```



```
class object_t {  
    public:  
        explicit object_t(const string& x) : object_m(new string_model_t(x))  
        { }  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
        { }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct concept_t {  
            virtual ~concept_t() { }  
            virtual void print_(ostream& out) const = 0;  
        };  
  
        struct string_model_t : concept_t {  
            explicit string_model_t(const string& x) : data_m(x) { }
```



```
class object_t {
public:
    explicit object_t(const string& x) : object_m(new string_model_t(x))
    { }
    explicit object_t(const int& x) : object_m(new int_model_t(x))
    { }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
```

```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```



```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };  
};
```



```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```



```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(new int_model_t(*x.object_m))  
    { }
```

```
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }
```

```
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```



```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }
```

```
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```



```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(x.object_m->copy_())  
    { }
```

```
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }
```

```
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```



```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(x.object_m->copy_())  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```





```
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:  
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(x.object_m->copy_())  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };
```

```
class object_t {
public:
    explicit object_t(const string& x) : object_m(new string_model_t(x))
    { }
    explicit object_t(const int& x) : object_m(new int_model_t(x))
    { }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
```

```
class object_t {  
    public:  
        explicit object_t(const string& x) : object_m(new string_model_t(x))  
        { }  
        explicit object_t(const int& x) : object_m(new int_model_t(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(x.object_m->copy_())  
        { }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct concept_t {  
            virtual ~concept_t() { }  
            virtual void print_(ostream& out) const = 0;  
        };  
  
        struct string_model_t : concept_t {  
            explicit string_model_t(const string& x) : data_m(x) { }
```



public:

```
explicit object_t(const string& x) : object_m(new string_model_t(x))  
{ }
```

```
explicit object_t(const int& x) : object_m(new int_model_t(x))  
{ }
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }
```





```
explicit object_t(const string& x) : object_m(new string_model_t(x))
{ }
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }
```





```
{ }
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
```





```
explicit object_t(const int& x) : object_m(new int_model_t(x))
{ }
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```





```
{ }  
~object_t() { delete object_m; }  
  
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };  
  
    struct string_model_t : concept_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };
```







```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```

```
struct int_model_t : concept_t {
```



```
object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
        explicit string_model_t(const string& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        string data_m;
    };

    struct int_model_t : concept_t {
        explicit int_model_t(const int& x) : data_m(x) { }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
        explicit string_model_t(const string& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }

        string data_m;
    };

    struct int_model_t : concept_t {
        explicit int_model_t(const int& x) : data_m(x) { }
        void print_(ostream& out) const { print(out, data_m); }
```

```
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
};
```



```
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```

```
struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
```





```
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct concept_t {
```

```
    virtual ~concept_t() { }
```

```
    virtual void print_(ostream& out) const = 0;
```

```
};
```

```
struct string_model_t : concept_t {
```

```
    explicit string_model_t(const string& x) : data_m(x) { }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;
```

```
};
```

```
struct int_model_t : concept_t {
```

```
    explicit int_model_t(const int& x) : data_m(x) { }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    int data_m;
```

```
};
```



```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
};
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};  
  
concept_t* object_m;
```



```
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual void print_(ostream& out) const = 0;  
    };  
  
    struct string_model_t : concept_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };  
  
    struct int_model_t : concept_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };  
  
    concept_t* object_m;  
};
```



private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```

```
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};
```

```
concept_t* object_m;  
};
```



client

library

+

private:

```
struct concept_t {  
    virtual ~concept_t() { }
```

```
virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }
```

```
void print_(ostream& out) const { print(out, data_m); }
```

```
string data_m;  
};
```

```
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }
```

```
void print_(ostream& out) const { print(out, data_m); }
```

```
int data_m;  
};
```

+

cout

guidelines

defects

client

library

+

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```

```
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    concept_t* copy_() const { return new int_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};
```

+

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(1));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

cout

```
<document>
0
Hello
2
3
</document>
```



client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(2));
    document.push_back(object_t(3));

    print(cout, document);
    return 0;
}
```

guidelines

- Don't allow polymorphism to complicate the client code.
  - Polymorphism is an implementation detail.

```
void print(ostream& out, const string& x) { out << x << endl; }  
void print(ostream& out, const int& x) { out << x << endl; }
```

```
class object_t {  
public:
```

```
    explicit object_t(const string& x) : object_m(new string_model_t(x))  
    { }  
    explicit object_t(const int& x) : object_m(new int_model_t(x))  
    { }  
    ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(x.object_m->copy_())  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }
```

```
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual concept_t* copy_() const = 0;  
        virtual void print_(ostream& out) const = 0;
```



client

library

```
class object_t {  
public:
```

```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```



cout

guidelines

defects

```
template <typename T>
void print(ostream& out, const T& x) { out << x << endl; }
```

```
class object_t {
public:
```

```
    template <typename T>
    explicit object_t(const T& x) : object_m(new model<T>(x))
    { }
```

```
    ~object_t() { delete object_m; }
```

```
    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
```

```
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }
```

```
    friend void print(ostream& out, const object_t& x);
```

```
    friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };
```



```
template <typename T>
void print(ostream& out, const T& x) { out << x << endl; }

class object_t {
public:
    template <typename T>
    explicit object_t(const T& x) : object_m(new model<T>(x))
    { }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };
};
```



```
void print(ostream& out, const T& x) { out << x << endl; }
```

```
class object_t {  
public:  
    template <typename T>  
    explicit object_t(const T& x) : object_m(new model<T>(x))  
    { }  
    ~object_t() { delete object_m; }  
  
    object_t(const object_t& x) : object_m(x.object_m->copy_())  
    { }  
    object_t& operator=(object_t x)  
    { swap(*this, x); return *this; }  
  
    friend void print(ostream& out, const object_t& x);  
    friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual concept_t* copy_() const = 0;  
        virtual void print_(ostream& out) const = 0;  
    };  
};
```

```
class object_t {
public:
    template <typename T>
    explicit object_t(const T& x) : object_m(new model<T>(x))
    { }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
```

```
class object_t {  
    public:  
        template <typename T>  
        explicit object_t(const T& x) : object_m(new model<T>(x))  
        { }  
        ~object_t() { delete object_m; }  
  
        object_t(const object_t& x) : object_m(x.object_m->copy_())  
        { }  
        object_t& operator=(object_t x)  
        { swap(*this, x); return *this; }  
  
        friend void print(ostream& out, const object_t& x);  
        friend void swap(object_t& x, object_t& y);  
    private:  
        struct concept_t {  
            virtual ~concept_t() { }  
            virtual concept_t* copy_() const = 0;  
            virtual void print_(ostream& out) const = 0;  
        };  
  
        struct string_model_t : concept_t {  
            explicit string_model_t(const string& x) : data_m(x) { }
```



public:

```
template <typename T>
explicit object_t(const T& x) : object_m(new model<T>(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy_() const = 0;
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy_() const { return new string_model_t(data_m); }
```



```
template <typename T>
explicit object_t(const T& x) : object_m(new model<T>(x))
{ }
~object_t() { delete object_m; }

object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy_() const = 0;
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy_() const { return new string_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }
```





```
explicit object_t(const T& x) : object_m(new model<T>(x))  
{ }  
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }
```





```
{ }  
~object_t() { delete object_m; }  
  
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;
```





```
~object_t() { delete object_m; }
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())  
{ }
```

```
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};
```



```
object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy_() const = 0;
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy_() const { return new string_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```

```
object_t(const object_t& x) : object_m(x.object_m->copy_())
{ }
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }

friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };

    struct string_model_t : concept_t {
        explicit string_model_t(const string& x) : data_m(x) { }
        concept_t* copy_() const { return new string_model_t(data_m); }
        void print_(ostream& out) const { print(out, data_m); }

        string data_m;
    };

    struct int_model_t : concept_t {
```



```
{ }  
object_t& operator=(object_t x)  
{ swap(*this, x); return *this; }  
  
friend void print(ostream& out, const object_t& x);  
friend void swap(object_t& x, object_t& y);  
private:  
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }
```







```
object_t& operator=(object_t x)
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
```

private:

```
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy() const = 0;
    virtual void print_(ostream& out) const = 0;
};
```

```
struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy() const { return new string_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};
```

```
struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    concept_t* copy() const { return new int_model_t(data_m); }
```





```
{ swap(*this, x); return *this; }
```

```
friend void print(ostream& out, const object_t& x);
```

```
friend void swap(object_t& x, object_t& y);
```

```
private:
```

```
struct concept_t {
```

```
    virtual ~concept_t() { }
```

```
    virtual concept_t* copy() const = 0;
```

```
    virtual void print_(ostream& out) const = 0;
```

```
};
```

```
struct string_model_t : concept_t {
```

```
    explicit string_model_t(const string& x) : data_m(x) { }
```

```
    concept_t* copy() const { return new string_model_t(data_m); }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```

```
    string data_m;
```

```
};
```

```
struct int_model_t : concept_t {
```

```
    explicit int_model_t(const int& x) : data_m(x) { }
```

```
    concept_t* copy() const { return new int_model_t(data_m); }
```

```
    void print_(ostream& out) const { print(out, data_m); }
```



```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy_() const = 0;
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy_() const { return new string_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    concept_t* copy_() const { return new int_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }
```

```
friend void print(ostream& out, const object_t& x);
friend void swap(object_t& x, object_t& y);
private:
struct concept_t {
    virtual ~concept_t() { }
    virtual concept_t* copy() const = 0;
    virtual void print_(ostream& out) const = 0;
};

struct string_model_t : concept_t {
    explicit string_model_t(const string& x) : data_m(x) { }
    concept_t* copy() const { return new string_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }

    string data_m;
};

struct int_model_t : concept_t {
    explicit int_model_t(const int& x) : data_m(x) { }
    concept_t* copy() const { return new int_model_t(data_m); }
    void print_(ostream& out) const { print(out, data_m); }

    int data_m;
```

```
friend void swap(object_t& x, object_t& y);  
private:  
    struct concept_t {  
        virtual ~concept_t() { }  
        virtual concept_t* copy() const = 0;  
        virtual void print_(ostream& out) const = 0;  
    };  
  
    struct string_model_t : concept_t {  
        explicit string_model_t(const string& x) : data_m(x) { }  
        concept_t* copy() const { return new string_model_t(data_m); }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        string data_m;  
    };  
  
    struct int_model_t : concept_t {  
        explicit int_model_t(const int& x) : data_m(x) { }  
        concept_t* copy() const { return new int_model_t(data_m); }  
        void print_(ostream& out) const { print(out, data_m); }  
  
        int data_m;  
    };
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};  
  
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    concept_t* copy_() const { return new int_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};
```

private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```

```
struct string_model_t : concept_t {  
    explicit string_model_t(const string& x) : data_m(x) { }  
    concept_t* copy_() const { return new string_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    string data_m;  
};  
  
struct int_model_t : concept_t {  
    explicit int_model_t(const int& x) : data_m(x) { }  
    concept_t* copy_() const { return new int_model_t(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    int data_m;  
};
```

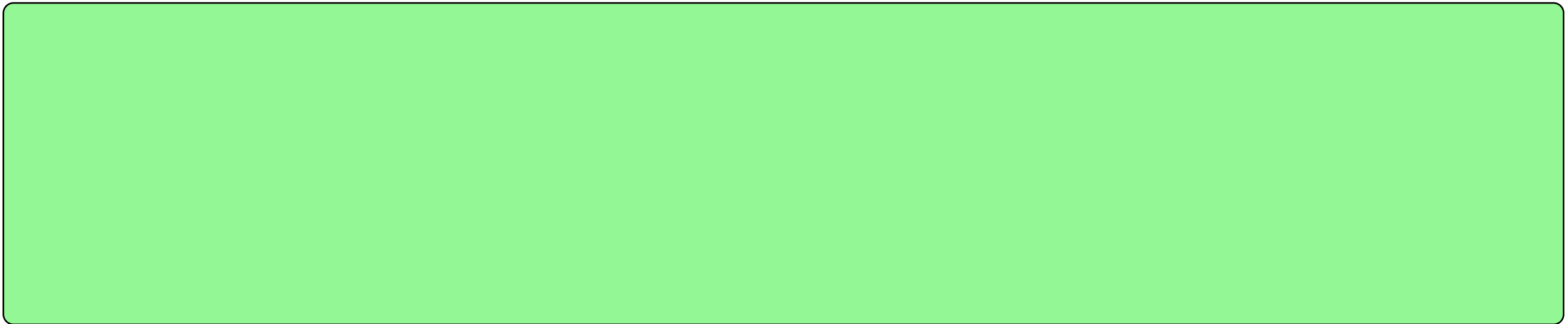
client

library



private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```



```
concept_t* object_m;  
};  
  
void swap(object_t& x, object_t& y)  
{  
    swap(x.object_m, y.object_m);  
}
```



cout

guidelines

defects





private:

```
struct concept_t {  
    virtual ~concept_t() { }  
    virtual concept_t* copy_() const = 0;  
    virtual void print_(ostream& out) const = 0;  
};
```

```
template <typename T>  
struct model : concept_t {  
    explicit model(const T& x) : data_m(x) { }  
    concept_t* copy_() const { return new model(data_m); }  
    void print_(ostream& out) const { print(out, data_m); }  
  
    T data_m;  
};
```

```
concept_t* object_m;  
};  
  
void swap(object_t& x, object_t& y)  
{  
    swap(x.object_m, y.object_m);  
}
```



client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(2));
```

```
    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
class my_class_t {  
    /* ... */  
};  
  
void print(ostream& out, const my_class_t& x)  
{ out << "my_class_t" << endl; }
```

```
int main()  
{  
    document_t document;  
  
    document.push_back(object_t(0));  
    document.push_back(object_t(string("Hello")));  
    document.push_back(object_t(2));  
    document.push_back(object_t(my_class_t()));  
  
    print(cout, document);  
    return 0;  
}
```

cout

guidelines

defects

client

library

```
class my_class_t {  
    /* ... */  
};  
  
void print(ostream& out, const my_class_t& x)  
{ out << "my_class_t" << endl; }
```

```
int main()  
{  
    document_t document;  
  
    document.push_back(object_t(0));  
    document.push_back(object_t(string("Hello")));  
    document.push_back(object_t(2));  
    document.push_back(object_t(my_class_t()));  
}
```

cout

ment);

<document>

0

Hello

2

my\_class\_t

</document>

guidelines

client

library

```
class my_class_t {  
    /* ... */  
};  
  
void print(ostream& out, const my_class_t& x)  
{ out << "my_class_t" << endl; }
```

```
int main()  
{  
    document_t document;  
  
    document.push_back(object_t(0));  
    document.push_back(object_t(string("Hello")));  
    document.push_back(object_t(2));  
    document.push_back(object_t(my_class_t()));  
  
    print(cout, doc  
    return 0;
```

guidelines

- The runtime-concept idiom allows polymorphism when needed without inheritance.
  - Client isn't burdened with inheritance, factories, class registration, and memory management.
  - Penalty of runtime polymorphism is only paid when needed.
  - Polymorphic types are used like any other types, including built-in types.

client

library

```
class my_class_t {
    /* ... */
};

void print(ostream& out, const my_class_t& x)
{ out << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(my_class_t()));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

client

library

```
class my_class_t {  
    /* ... */  
};  
  
void print(ostream& out, const my_class_t& x)  
{ out << "my_class_t" << endl; }  
  
int main()  
{  
    document_t document;  
  
    document.push_back(object_t(0));  
    document.push_back(object_t(string("Hello")));  
    document.push_back(object_t(document));  
    document.push_back(object_t(my_class_t()));  
  
    print(cout, document);  
    return 0;  
}
```

cout

guidelines

defects

client

library

```
class my_class_t {  
    /* ... */  
};  
  
void print(ostream& out, const my_class_t& x)  
{ out << "my_class_t" << endl; }  
  
int main()  
{  
    document_t document;  
  
    document.push_back(object_t(0));  
    document.push_back(object_t(string("Hello")));  
    document.push_back(object_t(document));  
}
```

cout

```
<document>  
0  
Hello  
<document>  
0  
Hello  
</document>  
my_class_t  
</document>
```



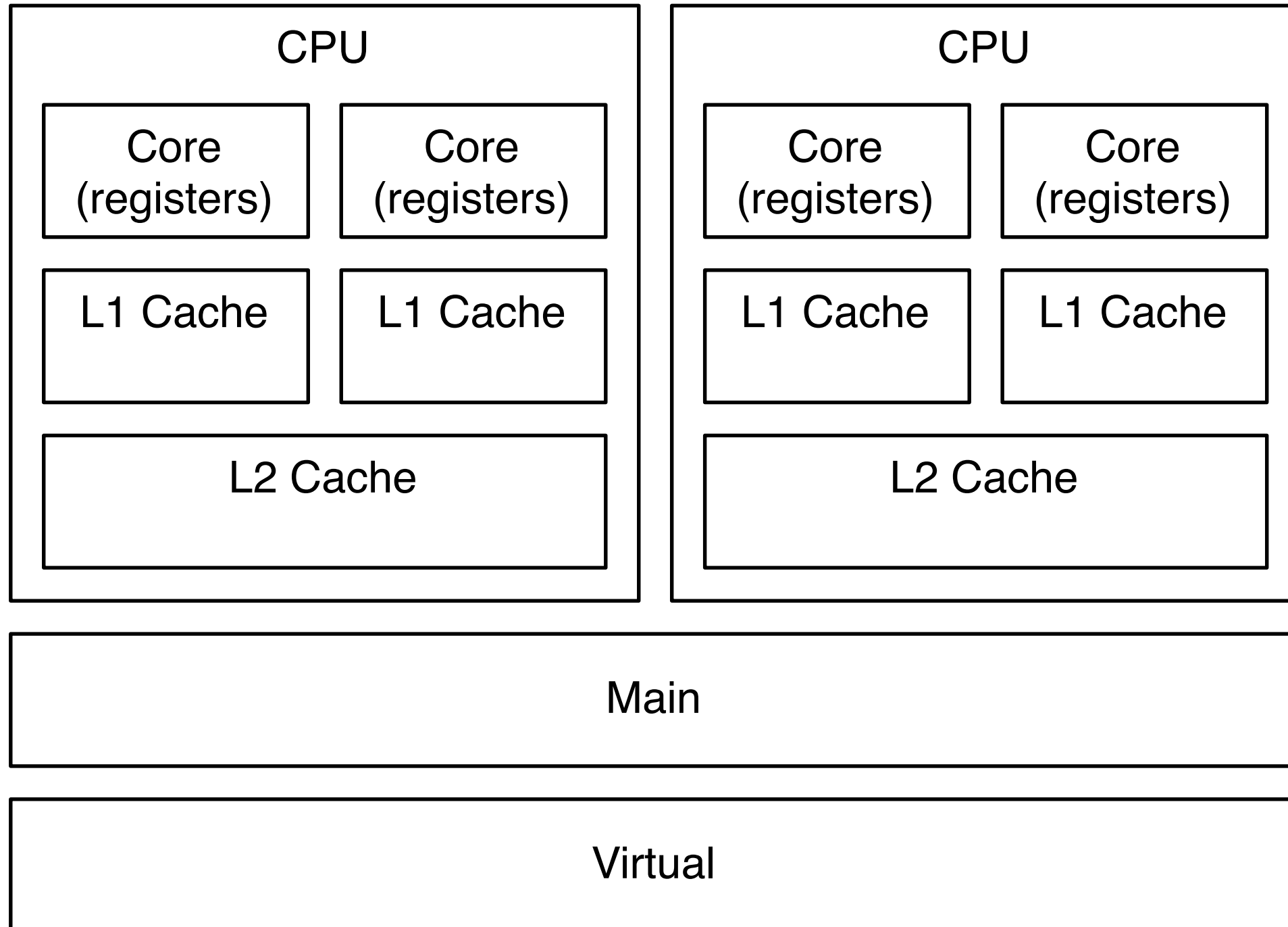
# Classes Without Synchronization

- Using regular semantics for the common basis operations *copy*, *assignment*, and *move*, help to reduce shared objects and improve thread safety
- Regular types promote interoperability of software components, increases productivity as well as quality, security, and performance
- There is no necessary performance penalty to using regular semantics, and often times there are performance benefits from a decreased use of the heap (currently a synchronization point on Mac and Windows).
- When constructs such as `shared_ptr` are used, or already in place, great care must be taken

# Synchronization

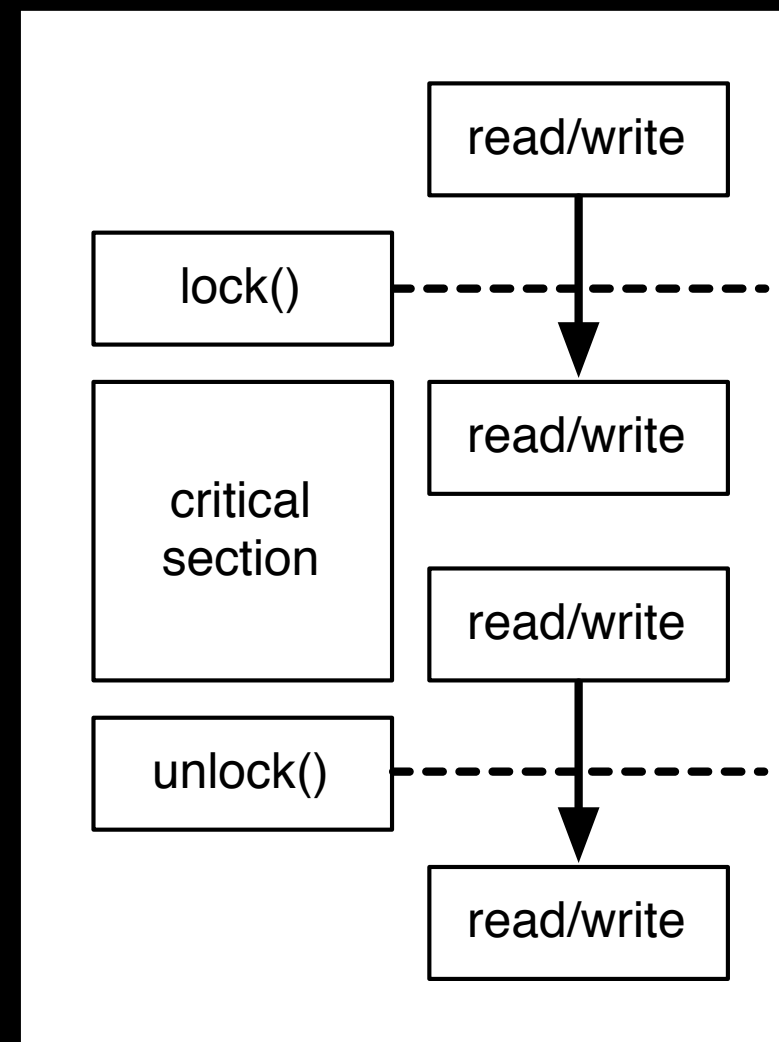
- Sometimes sharing is unavoidable, or the cost of avoidance is unacceptable
  - The cost of avoidance is sometimes amortized across the tasks
  - It may be better to write code that takes many times longer on one processor but scales linearly
- There are two basic synchronization primitives:
  - Locks
  - Atomics
- For modifying a single word of memory on a Core 2 Duo:
  - An Atomic takes 5x as long as direct memory access
  - A lock/unlock takes 10x as long as a direct memory access

# Memory Hierarchy



# Locks

- Lock state is aquired / released atomically
- Compiler barrier prevents load / stores from moving out of critical section
  - But they can move into the critical section
- Memory barrier provides partial ordering of read / writes to main memory with respect to corresponding locks
- Double-Check-Locking fails because there is no guarantee that the memory written by the critical section occurs before the write to the checked value - *a race condition*



# Atomics

- Atomic operations only guarantee atomicity regarding the value being operated on
- Atomics do not provide a compiler or memory barrier
- You cannot build a lock from an atomic

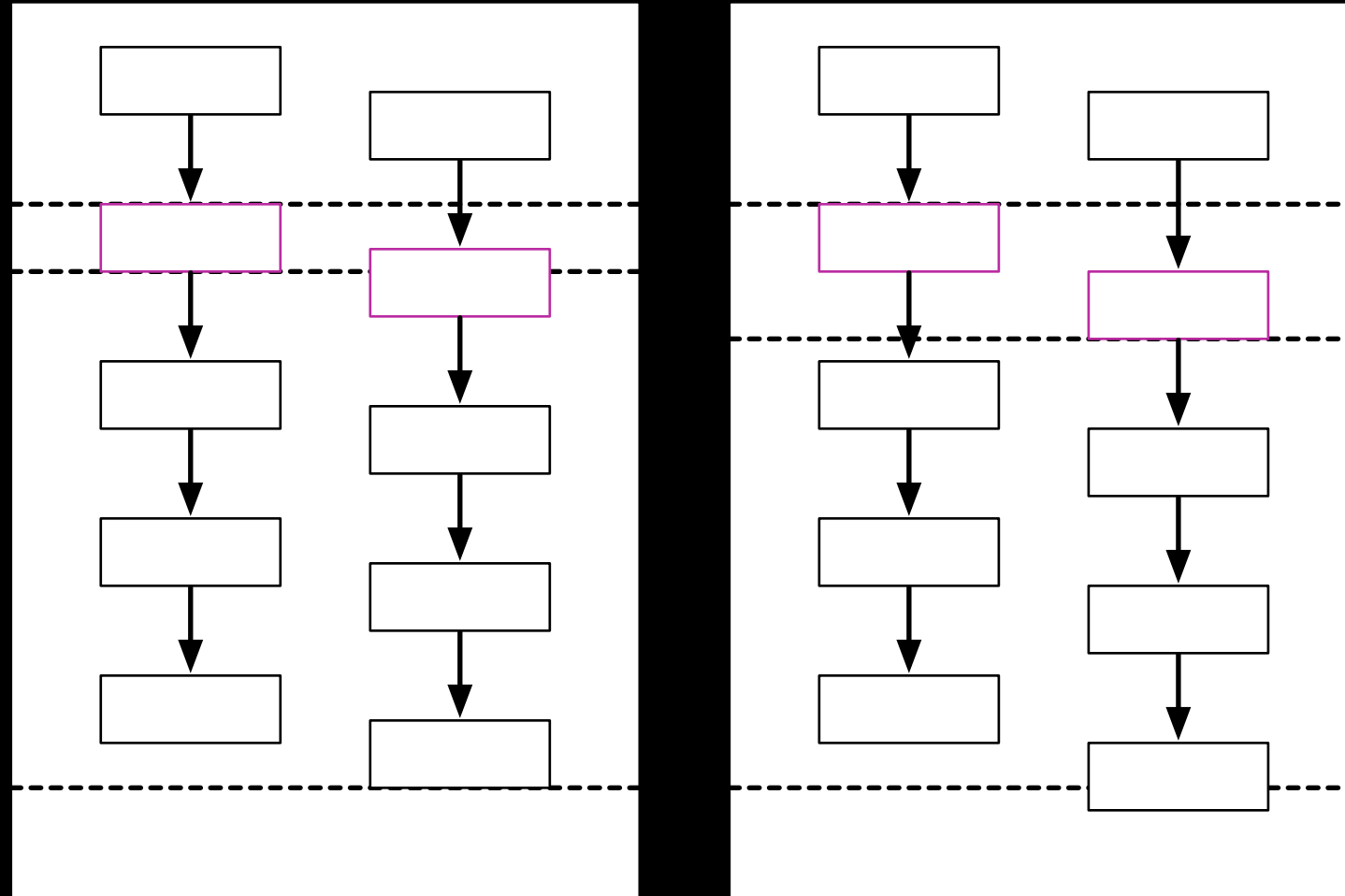
# Problems with Locks

- **Deadlock:** Multiple tasks wait for each other and will not resume until the other task proceeds
  - Avoid using locks, especially multiple locks at the same time. If you must use multiple locks, then always acquire/release them in a strictly scoped manner
- **Race Condition:** Multiple tasks read from and write to the same memory location without proper synchronization
  - Avoid shared data. Manage shared data in a disciplined manner

# Problems with Locks

- **Starvation:** More threads are executing than allowed by the proportion of the critical sections leading to high lock contention (see Amdahl's Law)
  - Avoid using locks. Hold locks as briefly as possible. Use *shared* locks when possible.
- **Livelock:** A result of starvation when some tasks cannot acquire a resource and forward progress is halted
  - Avoid using locks. Use *fair* locks unless you can be certain of light contention even with many processors under heavy load. Use shared locks when possible.
- **Convoying:** When a task is interrupted while holding or waiting for a lock causing other task to reach the lock and wait
  - Avoid using locks. Use *spin* locks if you are certain of light contention. Use shared locks when possible.

# Locks and Performance





# Common Use Cases for Sharing

- **Singletons:** Singletons (or any global) within an application must be sharable or all code accessing the shared instance must be synchronized
  - Do not force client code to directly manage sharing, treat object consistency in the presence of concurrency as any other invariant and protect it behind an API

# Disclaimer

- Some classes in the following code do not provide implementations of that standard basis operations for regular types and for brevity, the compiler supplied operations are not suppressed even if incorrect
- Complete implementations are left as an exercise

client

library

cout

guidelines

defects

```
/*  
    Question: Is singleton thread safe?  
*/  
  
const string singleton("Hello World!");  
  
void print_singleton(ostream& out) { out << singleton << endl; }  
  
int main()  
{  
    thread t1(bind(&print_singleton, ref(cout)));  
    thread t2(bind(&print_singleton, ref(cerr)));  
  
    return 0;  
}
```

client

library

```
/*  
    Question: Is singleton thread safe?  
*/  
  
const string singleton("Hello World!");  
  
void print_singleton(ostream& out) { out << singleton << endl; }  
  
int main()  
{  
    thread t1(bind(&print_singleton, ref(cout)));  
    print_singleton, ref(cerr));  
}
```

cout

----- cout -----

**Hello World!**

----- cerr -----

**Hello World!**

client

library

```
/*  
    Question: Is singleton thread safe?  
*/  
  
const string singleton("Hello World!");  
  
void print_singleton(ostream& out) { out << singleton << endl; }  
  
int main()  
{  
    thread t1(bind(&print_singleton, ref(cout)));  
    thread t2(bind(&print_singleton, ref(cerr)));  
  
    return 0;  
}
```

guidelines

- Globals are initialized in a thread safe manner prior to any call in the unit.
- But globals slow application launch and should be avoided.

client

library

cout

guidelines

defects

```
struct aggregate_string
{
    char* data_m;
    operator string () const { return data_m; }
};

const aggregate_string singleton = { "Hello World!" };

void print_singleton(ostream& out) { out << string(singleton) << endl; }

int main()
{
    thread(bind(&print_singleton, ref(cout)));
    thread(bind(&print_singleton, ref(cerr)));

    return 0;
}
```



client

library

```
struct aggregate_string
{
    char* data_m;
    operator string () const { return data_m; }
};
```

```
const aggregate_string singleton = { "Hello World!" };
```

```
void print_singleton(ostream& out) { out << string(singleton) << endl; }
```

```
int main()
{
    print_singleton(cout, ref(cout));
}
```

cout

----- cout -----

**Hello World!**

----- cerr -----

**Hello World!**

client

library

```
struct aggregate_string
{
    char* data_m;
    operator string () const { return data_m; }
};
```

```
const aggregate_string singleton = { "Hello World!" };
```

```
void print_singleton(ostream& out) { out << string(singleton) << endl; }
```

```
int main()
{
    thread(bind(&print_singleton, ref(cout)));
    thread(bind(&print_singleton, ref(cerr)));

    return 0;
}
```

guidelines

- Types with no-user defined constructors (including for members) can be used to reduce startup time
- Sometimes this causes a tradeoff with cost of use vs. initialization

client

library

cout

guidelines

defects

```
struct cons
{
    const char* name_m;
    const cons* next_m;
};

const cons x = { "x" };
const cons y = { "y", &x };
const cons z = { "z", &y };
const cons w = { "w", &y };

int main()
{
    for (const cons* f = &z; f != 0; f = f->next_m) cout << f->name_m << "->";
    cout << endl;

    for (const cons* f = &w; f != 0; f = f->next_m) cout << f->name_m << "->";
    cout << endl;

    return 0;
}
```

client

library

```
struct cons
{
    const char* name_m;
    const cons* next_m;
};
```

```
const cons x = { "x" };
const cons y = { "y", &x };
const cons z = { "z", &y };
const cons w = { "w", &y };
```

int

cout

{

```
f = &z; f != 0; f = f->next_m; cout << f->name_m << " ";
```

**z->y->x->**

**w->y->x->**

client

library

```
struct cons
```

```
{
```

```
    const char* name_m;
```

```
    const cons* next_m;
```

```
};
```

```
const cons x = { "x" };
```

```
const cons y = { "y", &x };
```

```
const cons z = { "z", &y };
```

```
const cons w = { "w", &y };
```

```
int main()
```

```
{
```

```
    for (const cons* f = &z; f != 0; f = f->next_m) cout << f->name_m << "->";
```

```
    cout << endl;
```

```
    for (const cons f = f->next_m) cout << f->name_m << "->";
```

guidelines

- It's possible to create compile time structures
- Combining templates with static class members allows for fairly complex structures

client

library

cout

guidelines

defects

client

library

```
int main()
{
    typedef adobe::vector<double[2]> complex_set_t;

    cout << adobe::type_info<complex_set_t>() << endl;

    return 0;
}
```

cout

guidelines

defects



client

library

```
int main()
{
    typedef adobe::vector<double[2]> complex_set_t;

    cout << adobe::type_info<complex_set_t>() << endl;

    return 0;
}
```

cout

**vector:version\_1:adobe<array[0000000002]<double>>**

client

library

```
int main()
{
    typedef adobe::vector<double[2]> complex_set_t;

    cout << adobe::type_info<complex_set_t>() << endl;

    return 0;
}
```

guidelines

- ASL uses const aggregates to simulate RTTI in a form which is ABI safe
- ASL uses similar techniques to simulate v-tables and inheritance

client

library

cout

guidelines

defects

```
/*  
    Question: Is singleton thread safe?  
*/  
  
const string& singleton()  
{  
    static const string result("Hello World!");  
    return result;  
}  
  
void print_singleton(ostream& out) { out << singleton() << endl; }  
  
int main()  
{  
    thread t1(bind(&print_singleton, ref(cout)));  
    thread t2(bind(&print_singleton, ref(cerr)));  
  
    return 0;  
}
```

client

library

```
/*  
    Question: Is singleton thread safe?  
*/  
  
const string& singleton()  
{  
    static const string result("Hello World!");  
    return result;  
}  
  
void print_singleton(ostream& out) { out << singleton() << endl; }
```

cout

in

?????

client

library

```
/*  
    Question: Is singleton thread safe?  
*/
```

```
const string& singleton()  
{  
    static const string result("Hello World!");  
    return result;  
}
```

```
void print_singleton(ostream& out) { out << singleton() << endl; }
```

```
int main()  
{  
    thread t1(bind(&print_singleton, ref(cout)));  
    thread t2(bind(ref(cerr)));  
}
```

guidelines

- With GCC, this code is correct
- With VC, this code is a potential race and may crash
- With C++0x, this code will be correct

/\*

Question: Is singleton thread safe?

\*/

```
void once_singleton(const string*& x)
{
    static const string result("Hello World!");
    x = &result;
}

const string& singleton()
{
    static const string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}
```

```
void print_singleton(ostream& out) { out << singleton() << endl; }
```

```
int main()
{
```



client

library

```
/*
```

```
    Question: Is singleton thread safe?
```

```
*/
```

```
void once_singleton(const string*& x)
{
    static const string result("Hello World!");
    x = &result;
}
```

```
const string& singleton()
{
```

cout

```
    string* result = 0;
```

```
    flag = BOOST_ONCE_INIT.
```

```
----- cout -----
```

```
Hello World!
```

```
----- cerr -----
```

```
Hello World!
```



client

library

/\*

Question: Is singleton thread safe?

\*/

```
void once_singleton(const string*& x)
{
    static const string result("Hello World!");
    x = &result;
}

const string& singleton()
{
    static const string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, once_singleton, ref(result));
    return *result;
}
```

guidelines

- Use call\_once to assure that singletons are correctly initialized
- call\_once does have cost on every invocation (check of once\_flag is atomic)
- I've made a couple of attempts at writing a useful template that will do the correct initialization depending on the compiler - I don't have anything simpler than this yet

client

library

```
/*  
    Question: Is singleton thread safe?  
*/
```

```
void once_singleton(string*& x)  
{  
    static string result("Hello World!");  
    x = &result;  
}  
  
string& singleton()  
{  
    static string* result = 0;  
    static once_flag flag = BOOST_ONCE_INIT;  
  
    call_once(flag, bind(&once_singleton, ref(result)));  
    return *result;  
}
```

```
void print_singleton(ostream& out) { out << singleton() << endl; }
```

```
int main()  
{
```



cout

guidelines

defects

client

library

+

```
void once_singleton(string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }

int main()
{
    thread t1(bind(&print_singleton, ref(cout)));
    thread t2(bind(&print_singleton, ref(cerr)));

    return 0;
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&print_singleton, ref(cout)));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&print_singleton, ref(cout)));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&modify_singleton));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&modify_singleton));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}
```

```
string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}
```

```
void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton("World!"); }
```

defects

- This will create a race condition and could cause a crash



client

library

+

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}
```

```
string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}
```

```
void print_singleton() { cout << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }
```

guidelines

- There is no simple fix for allowing direct read/write access to the client
- We need to figure out how we can allow access to the basis operations and protect the invariants

client

library

+

```
class shared_string
{
    mutable shared_mutex    mutex_m;
    string                  data_m;
public:
    explicit shared_string(const char* x) : data_m(x) { }
    shared_string& operator=(const char* x) {
        lock_guard<shared_mutex> lock(mutex_m);
        data_m = x;
        return *this;
    }
    operator string () const {
        shared_lock<shared_mutex> lock(mutex_m);
        return data_m;
    }
};
```

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(const string*& x)
{
    static string result("Hello World!");
    x = &result;
}

string& singleton()
{
    static string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << singleton() << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&modify_singleton));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

+

```
void once_singleton(shared_string*& x)
{
    static shared_string result("Hello World!");
    x = &result;
}

shared_string& singleton()
{
    static shared_string* result = 0;
    static once_flag flag = BOOST_ONCE_INIT;

    call_once(flag, bind(&once_singleton, ref(result)));
    return *result;
}

void print_singleton(ostream& out) { out << string(singleton()) << endl; }
void modify_singleton() { singleton() = "Goodbye Cruel World!"; }

int main()
{
    thread t1(bind(&modify_singleton));
    thread t2(bind(&print_singleton, ref(cerr)));
}
```

+

cout

guidelines

defects

client

library

cout

guidelines

defects

+

```
const char* table[] = {  
    "width",  
    "height",  
    "orientation"  
};  
  
const char* lookup(const char* x)  
{  
    static bool init;  
    if (!init) { sort(table, str_less_t()); init = true; }  
    return *lower_bound(table, x, str_less_t());  
}
```



client

library

cout

guidelines

defects

+

```
const char* const table[] = {
    "height",
    "orientation",
    "width"
};

const char* lookup(const char* x)
{
    #ifndef NDEBUG
        static bool init;
        if (!init) { init = true; assert(is_sorted(table, str_less_t()) && "FATAL: Table
must be sorted");
        #endif
        return *lower_bound(table, x, str_less_t());
    }
```



# Common Use Cases for Sharing

- Reference Counting: A common use case for atomics is reference counts
- An atomic count only protects the count - not the rest of the object
- `shared_ptr` to a mutable object has all of the same issues as singletons
  - "A `shared_ptr` is as good as a global variable."
- If you have a `shared_ptr` to an immutable object, you can optimize the case where the pointer isn't shared and allow modification - this is copy-on-write
- Since we can implement polymorphism without inheritance, our copy-on-write can operate on values instead of pointers
  - Programming is much less error prone when you eliminate the use of pointers

client

library

cout

guidelines

defects

```
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};
```

```
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};
```

client

library

```
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
    }
};
```

defects

- There is a subtle race condition here:
  - if count != 1 then the bad\_cow could also be owned by another thread(s)
  - if the other thread(s) releases the bad\_cow between these two atomic operations
  - then our count will fall to zero and we will leak the object

```
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            if (0 == --object_m->count_m) delete object_m;
            object_m = tmp;
        }
        return *this;
    }
};
```

# Common Use Cases for Sharing

- One common concern with using regular types vs. shared pointers is that you end up with too many copies
- By leveraging RVO with the ASL move library, and `copy_on_write`, all unnecessary copies can be eliminated

client

library

+

}

```
void print(ostream& out, const object_t& x)
{ x.object_m->print_(out); }
```

```
typedef vector<object_t> document_t;
```

```
void print(ostream& out, const document_t& x)
{
    out << "<document>" << endl;
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {
        print(out, *f);
    }
    out << "</document>" << endl;
}
```

```
typedef adobe::vector<document_t> history_t;
```

```
void commit(history_t& x) { assert(x.size()); x.push_back(x.back()); }
void undo(history_t& x) { assert(x.size() > 1); x.pop_back(); }
document_t& current(history_t& x) { assert(x.size()); return x.back(); }
```

cout

guidelines

defects



client

library

+

```
}  
  
void print(ostream& out, const object_t& x)  
{ x.object_m->print_(out); }
```

```
typedef adobe::vector<copy_on_write<object_t> > document_t;
```

```
void print(ostream& out, const document_t& x)  
{  
    out << "<document>" << endl;  
    for (document_t::const_iterator f = x.begin(), l = x.end(); f != l; ++f) {  
        print(out, **f);  
    }  
    out << "</document>" << endl;  
}
```

```
typedef adobe::vector<document_t> history_t;
```

```
void commit(history_t& x) { assert(x.size()); x.push_back(x.back()); }  
void undo(history_t& x) { assert(x.size() > 1); x.pop_back(); }  
document_t& current(history_t& x) { assert(x.size()); return x.back(); }
```

cout

guidelines

defects

client

library

+

```
class object_t {
public:
    template <typename T>
    explicit object_t(const T& x) : object_m(new model<T>(x))
    { }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };

    template <typename T>
    struct model : concept_t {
```

+

cout

guidelines

defects

client

library

+

```
class object_t {
public:
    template <typename T>
    explicit object_t(const T& x) : object_m(new model<T>(x))
    { cout << "copy-ctor" << endl; }
    ~object_t() { delete object_m; }

    object_t(const object_t& x) : object_m(x.object_m->copy_())
    { }
    object_t& operator=(object_t x)
    { swap(*this, x); return *this; }

    object_t(move_from<object_t> x) : object_m(x.source.object_m)
    { x.source.object_m = 0; }

    friend void print(ostream& out, const object_t& x);
    friend void swap(object_t& x, object_t& y);
private:
    struct concept_t {
        virtual ~concept_t() { }
        virtual concept_t* copy_() const = 0;
        virtual void print_(ostream& out) const = 0;
    };
};
```

+

cout

guidelines

defects

```
class my_class_t
{
    /* ... */
};

void print(ostream& out, const my_class_t& x)
{ out << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(document));
    document.push_back(object_t(my_class_t()));

    print(cout, document);
    return 0;
}
```

client

library

```
int main()
{
    document_t document;

    document.push_back(object_t(0));
    document.push_back(object_t(string("Hello")));
    document.push_back(object_t(document));
    document.push_back(object_t(my_class_t()));

    print(cout, document);
    return 0;
}
```

cout

guidelines

defects

# Concluding Remarks

- Q & A
- References:
- Memory Barriers:
  - <http://lxr.linux.no/linux/Documentation/memory-barriers.txt>
- C++ Memory Model:
  - [http://www.hpl.hp.com/personal/Hans\\_Boehm/](http://www.hpl.hp.com/personal/Hans_Boehm/)
- Regular Types:
  - <http://stepanovpapers.com/>



Adobe

client

library

```
struct shape_t
{
    virtual ~shape_t() { }
    virtual void draw() = 0;
};

typedef vector<shared_ptr<shape_t> > document_t;

struct circle_t : shape_t
{
    void draw() { cout << "circle_t" << endl; }
};
```

cout

guidelines

defects



```
int main()
{
    document_t document;
    document.push_back(shared_ptr<shape_t>(new circle_t));

    for (document_t::const_iterator f = document.begin(), l = document.end();
         f != l; ++f) {
        (**f).draw();
    }

    return 0;
}
```

client

library

```
int main()
{
    document_t document;
    document.push_back(shared_ptr<shape_t>(new circle_t));

    for (document_t::const_iterator f = document.begin(), l = document.end();
         f != l; ++f) {
        cout << *f << endl;
    }
}
```

cout

**circle\_t**

```
struct shape_t
{
    virtual ~shape_t() { }
    virtual void draw() const = 0;
};

typedef std::vector<shared_ptr<shape_t> > document_t;

struct circle_t : shape_t
{
    void draw() const { cout << "circle_t" << endl; }
};

template <typename T>
void draw(const T& x) { x.draw(); }

template <typename T>
struct poly_shape_t : shape_t
{
    explicit poly_shape_t(T x) : data_m(move(x)) { }
    void draw() const { draw(data_m); }

    T data_m;
```

```
};

template <typename T>
void draw(const T& x) { x.draw(); }

template <typename T>
struct poly_shape_t : shape_t
{
    explicit poly_shape_t(T x) : data_m(move(x)) { }
    void draw_() const { draw(data_m); }

    T data_m;
};

template <typename T>
shared_ptr<shape_t> make_poly_shape(T x)
{ return shared_ptr<shape_t>(new poly_shape_t<T>(move(x))); }
}
```

```
struct square_t
{
    void draw() const { cout << "square_t" << endl; }
};

void draw(const int& x) { cout << x << endl; }

int main()
{
    document_t document;
    document.push_back(shared_ptr<shape_t>(new circle_t));
    document.push_back(make_poly_shape(square_t()));
    document.push_back(make_poly_shape(int(10)));

    for (document_t::const_iterator f = document.begin(), l = document.end();
         f != l; ++f) {
        (**f).draw_();
    }

    return 0;
}
```

client

library

```
struct square_t
{
    void draw() const { cout << "square_t" << endl; }
};
```

```
void draw(const int& x) { cout << x << endl; }
```

cout

in

```
circle_t
square_t
10
```

```

struct shape_t
{
    virtual ~shape_t() { }
    virtual void draw() const = 0;
};

template <typename T>
struct disable_if_is_shape_t_pointer :
    disable_if<is_base_and_derived<shape_t, typename remove_pointer<T>::type> >
{ };

struct poly_shape_t
{
    template <typename T>
    explicit poly_shape_t(T x, typename disable_if_is_shape_t_pointer<T>::type* = 0)
        : object_m(new model_t<T>(move(x))) { }
    explicit poly_shape_t(shape_t* p) : object_m(p) { }
    ~poly_shape_t() { delete object_m; }
    poly_shape_t(const poly_shape_t& x)
        : object_m(dynamic_cast<concept_t&>(*x.object_m).copy()) { }
    poly_shape_t(move_from<poly_shape_t> x)
        : object_m(x.source.object_m) { x.source.object_m = 0; }
    poly_shape_t& operator=(poly_shape_t x) { swap(*this, x); return *this; }

```

```
poly_shape_t& operator=(poly_shape_t x) { swap(*this, x); return *this; }
friend inline void swap(poly_shape_t& x, poly_shape_t& y)
    { swap(x.object_m, y.object_m); }
friend inline void draw(const poly_shape_t& x) { x.object_m->draw(); }

struct concept_t : shape_t
{
    virtual concept_t* copy() = 0;
};

template <typename T>
struct model_t : concept_t
{
    explicit model_t(T x) : data_m(move(x)) { }
    void draw() const { draw(data_m); }
    concept_t* copy() { return new model_t(data_m); }

    T data_m;
};

shape_t* object_m;
};
```



```
poly_shape_t& operator=(poly_shape_t x) { swap(*this, x); return *this; }
friend inline void swap(poly_shape_t& x, poly_shape_t& y)
    { swap(x.object_m, y.object_m); }
friend inline void draw(const poly_shape_t& x) { x.object_m->draw(); }

struct concept_t : shape_t
{
    virtual concept_t* copy_() = 0;
};

template <typename T>
struct model_t : concept_t
{
    explicit model_t(T x) : data_m(move(x)) { }
    void draw_() const { draw(data_m); }
    concept_t* copy_() { return new model_t(data_m); }

    T data_m;
};

shape_t* object_m;
};
```

client

library

```
typedef vector<poly_shape_t> document_t;
```

```
struct circle_t : shape_t
```

```
{
```

```
    void draw() const { cout << "circle_t" << endl; }
```

```
};
```

```
template <typename T>
```

```
void draw(const T& x) { x.draw(); }
```

cout

guidelines

defects

```
struct square_t
{
    void draw() const { cout << "square_t" << endl; }
};

void draw(const int& x) { cout << x << endl; }

int main()
{
    document_t document;
    document.push_back(poly_shape_t(new circle_t));
    document.push_back(poly_shape_t(square_t()));
    document.push_back(poly_shape_t(int(10)));

    for_each(document, bind((void (*)(const poly_shape_t&))(&draw), _1));

    return 0;
}
```

client

library

```
struct square_t
{
    void draw() const { cout << "square_t" << endl; }
};
```

```
void draw(const int& x) { cout << x << endl; }
```

cout

in

```
circle_t
square_t
10
```