Building Data Structures that can be Used without Stress

Local Variables

Local variables are created on the stack, and have independant storage space. In this way, value semantics is maintained.

```
{
    int i = 4;
    struct { int x, double y } ff;
    double g[100];
    ...

}
```

When a local variable is created, the compiler knows how much space it needs (4, 12, and 800 bytes in the example). It is not our problem.

Value Semantics

I would like to repeat once more that C^{++} is based on value semantics.

Variables don't share anything unless you declare them to be of a special type that implies sharing.

A reference shares with a local variable, or a container element.

An iterator shares an element with its container.

In both cases, sharer and shared are not on the same level. It is clear who is the owner, and who has secondary access.

Garbage collection is only needed with equal level sharing, which does not fit with the proper way of using C^{++}.

Local Variables with Unpredictable Size

Unfortunately, many frequent data structures have unpredictable size: strings, vectors, matrices, search trees, logical formulas, etc.

The *C* solution is to declare a big array, and hope that the value will fit:

```
#define MAXNAMELENGTH 100
   // In one of the header files.

{
   char name[ MAXNAMELENGTH ];
   printf( "what is your name?" );
   scanf( "%s", name );
}
```

This **(1)** wastes a lot of memory, and **(2)** is still not guaranteed to fit.

Using the Heap

It seems better to create (most of) the object on the heap:

```
struct string
{
    size_t len;
    char *p;
};
```

```
{
   string s;

   printf( "what is your name?" );
   readstring( s );     // Allocates s.p big enough.
   printf( "hello, " );
   printstring( s );
   clearstring( s );    // Frees s.p.

   printf( "where are you from?" );
   readstring( s );
   printstring( s );
   clearstring( s );
}
```

Allocating Variables on the Heap

Now the size problem is solved, but it has become very easy to make mistakes when using **string**.

Forgetting `clearstring( )` causes memory leaks.

Using system assignment (=) will cause sharing between the `p`-fields. This results in loss of value semantics. It may cause memory leaks, and/or segmentation faults.

Also, **string** cannot be used in expressions, because we won't be able to call `clearstring()` on temporary variables.

An expression of form
`concatstring( readstring( ), readstring( ) )` will cause memory leaks.

The Essential Operators

C^{++} is designed to enable building of such objects (as **string, bignum**) such that they can be used without stress:

- constructors can allocate additional resources, after the local part of the variable has been created, and before the variable is used. If the user declares a variable without initializer, a default constructor is inserted.
- assignment overwrites an existing value. It is possible to write an assignment operator that frees the old string and allocates a new one. If a class definition contains an assignment operator, it will be automatically inserted everywhere, where the user overwrites an existing value with `=`.
- A destructor is called just before a local variable goes out of scope. If you create a destructor for a given type, the compiler will insert it everwhere where it is needed.

The Essential Operators

For **string**, the default constructor should assign
`len = 0; p = new char[0];`

In the case an initializer is given (from `const char* c`), the constructor should assign `len = strln(c)`, and allocate
`p = new char[ len ];`.

Assignment should deallocate the current `p`, and after that reassign `len`, and allocate `p = new char[ len ]`. After that, it can copy the value.

The destructor should deallocate p with `delete[] p;`.

Hidden Essential Operators

```
{
   string s;   // Default constructor.

   s = "c";
      // Constructor from const char*,
      // followed by assignment.

   string s2 = s; // Copy constructor.

   s = concatstring( s2, "+" );
      // Constructor from const char*,
      // an assignemt, a destructor call.

   s2 = s;   // Assignment.
} // Destructor calls for s and s2.
```

Moral

If you manage to get the essential methods and operators right (a few constructors, an assignment operator, a destructor), you can build data structurs that allocate on the heap, but have the same easyness of use as built-in types.

If you manage to get these three methods right, there will be no observable distinction between built-in and built-by-the-user.

Constructors

A constructor has the same name as the class it constructs:

```
string( const char* s )  // s is argument.
{
   len = strlen(s);
   p = new char[ len ];
   strcpy( p, s );
}
```

We saw before that C^{++} makes a strict distinction between initialization and assignment.

The two assignments in the constructor are indeed assignments.

When are the fields initialized?

Initializers

C^{++} has special syntax for initialization in constructors:

```
string( const char* s )
   len{ strlen(s) },
   p{ new char[ len ] }
{
   strcpy( p, s );
}
```

Initialization takes place in the order in which the fields are declared in the class. Not in the order in which they appear in the constructor!

Reversing the initializers in `string` would still work.

Initializers (2)

- Fields without initializer are initialized by the default constructor (0-argument constructor) of their type. In many cases, this is the right value.
- Avoiding initializers (out of lazyness or because of a Java background) is a bad habit. It causes inefficiency, and in addition, the class types of the fields are required to have default constructors.
- Java doesn't have initializers because it has `(null)`.

Destructors

You do not call a destructor by yourself!

The compiler makes sure that the destructor is called whenever it is needed.

- When a local variable goes out of scope. (This extends to arrays or structs of which the class variable is a field.)
- When an intermediate result in a functional expression goes out of scope.

The task of the programmer is to write the destructor correctly. You don't have to worry about when it is called. The compiler will do that for you.

No discipline or long term memory is needed.

If the object does not claim any resources, don't write a destructor!

Example of Allocating Types (Vector)

```
// This is more or less the implementation of
// std::vector:

template< typename X > struct vector
{
   size_t size;
      // Between 0 .. size is initialized.
   size_t capacity;
      // Between size and capacity is allocated but
      // not initialized.
   X* p;
};
```

Vector

Constructors: If we have an initializer, then reserve the smallest power of 2 that can hold the initializing value or 8 if the length of the initializer is less than 8.

If no initializer is present, then start with size = 0 and capacity = 8.

Assignment: If the new value fits with capacity, then just copy the new value. If not, deallocate our current p, and allocate a new p that is large enough.

```
void push_back( const X& x )
{
   if( capacity < size )
   { p [ size ++ ] = x;
       // This requires some low level trickery
       // because p[size] was not initialized.
   }
   else
   { allocate p1 =  new X[ 2 * capacity ];
      copy p[ 0 .. size ] to p1[ 0 .. size ].
      destroy p [ 0 .. size ]; delete p;
        // This requires some low level trickery.
      p1 [ size ++ ] = x;
        // Some more low level trickery.
   }
}
```

Note: You normally don't write code as on the previous page. It is just an illustration to show how the things work.

Default Constructor

There is at most one constructor without arguments. If it is exists, it is called the default constructor.

It is inserted when you declare a variable without initializer.

```
{
   X x;
      // Equivalent to X x = X( );
}
```

You should provide a default constructor only when there is a meaningful default. This means that there must exist one value that is very special among the other values.

The need for default constructors is often created by bad coding, by not using initializers, or declaring a variable too early:

```
{ X x;

  for( unsigned int i = 0; i < 5; ++ i )
  {
     x = X(i);
     std::cout << x << "\n";
  }
}
```

Only create a default constructor when a natural candidate exists! Empty string, zero length vector, number zero, empty search tree are OK.

Identity matrix is dubious. Why not zero matrix?

Single Argument Constructors

Examples are:

```
string( const char* s );
string( const string& s );
string( size_t i );
   // Artificial example. A string of i spaces.
```

Single argument constructors can be automatically inserted in initializations and assignments:

```
string s = "C#";
s = "C++"; // constructor + assignment.
```

Is this good? That depends on what the constructor does.

Explicit Constructors

If a one argument constructor constructs something that can be considered a different representation of its argument, then implicit conversion is fine:

```
string( const char* s );
   // String is just another representation of s.
```

But what if it is not?

```
string s = 4; // Assigns 4 spaces.
s = 5; // Assigns 5 spaces.
```

Use `explicit` keyword to forbid implicit conversion:

```
explicit string( size_t i );
```

Copy Constructor

A constructor with one argument, which is a reference to the same class, is called copy constructor:

```
string( const string& s );
```

In the example below, the second initialization uses a copy constructor:

```
string s1 = "abcdefgh";
   // string( const char* );

string s2 = s1;
   // string( const string& s )
```

Copy Constructor (2)

Copy construtors are also used for parameter passing, when arguments are passed by value:

```
size_t length( string s )
{
   ...
}  // Implicit destructor call for s.


string s = "hi!";
size_t i = length(s);
   // Uses copy constructor.
```

Assignment

Assignment has form

```
      void operator = ( const string& s );
      void operator = ( string s );
      void operator = ( const char* s );
```

You can write any assignment operator that you like.

Assignment is usually a member function, but it doesn't have to be.

The language allows that an assignment operator returns something, but I think that using this is not a good idea.

Self and Subtree Assignment

Consider the following assignment operator:

```
void operator = ( const string& s )
{
   delete[] p;
   p = new char[ s. capacity ];
   // copy contents of s.p into p.
}
```

What happens if somebody assigns `s = s`?

Whenever you write an assignment operator, think about self assignment. If you are dealing with tree like structures, think about subtree assignment.

Always test self and subtree assignment.

Self and Subtree Assignment

All following versions are safe:

```
void operator = ( const string& s ) {
   if( p != s.p )
   {
      ...
   }
}

void operator = ( const string& s ) {
   if( capacity < s. capacity )
   {
      // *this must be distinct from s.
   }
}
```

```
void operator = ( string s )
{
   delete p;
      // Since s is a fresh copy, it
      // is certainly distinct from *this.
}
```

Destructors

A destructor has name `~C`, where C is the name of the class.

Its task is to release resources that were claimed by the object.

Note that the fields of the object are destroyed separately after the destructor of the class is finished.

The following class does not need a destructor:

```
struct twostrings
{
   string first;
   string second;
};
```

Destructors (2)

Then why are destructors needed at all?

Mostly because pointers do not destroy the object they point to.
They cannot do this by default, because pointers possibly share.

There exist pointers that do destroy when they go out of scope.
There also exist sharing pointers with built-in garbage collection.
Such pointers are called smart pointers.

One still needs ordinary pointers to build them.

Other Implementations of Vector and String

- The current implementation in the STL library `std::string` keeps the first part of the string locally on the stack, and the rest on the heap. In this way, short strings can be used without using the heap. Using the heap is inefficient because it may cause cash misses.

- It is tempting to share on the heap. It is possible to do this as long as there is no assignment to a shared object. One can use reference counting, and make a unique copy when an assignment is made, and the reference counter is greater than one. This technique is acceptable because it can be hidden, so that value semantics is preserved.

  Earlier versions of `std::string` used this. I often use this for logical formulas.