# Building Objects 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 functional 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**) in 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 assignment, two destructor calls.

   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 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 does not need field initializers because it has `(null)`.

## Destructor

The task of the destructor is return resources that are held by a variable that goes out of scope.

1. In 90% of the cases, the destructor returns heap memory (that was obtained by `new`).
2. Sometimes it closes files that were owned by the variable.
3. It may return locks.

## Destructor (2)

Resources that are occupied by a field are not returned by the destructor. The following class does not need a destructor.

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

Rule is: If you are user of a class, you don't have to know if it has a destructor.

## When to Call a Destructor

Never. 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, if the class holds resources. You don't have to worry about when it is called. The compiler decides that for you.

No discipline or long term memory is needed.

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

## There is no Way Around the Destructor

If you follow the rules on the previous slides, no user of your class needs to know that it secretly allocates something:

```
int main( int argc, char* args[] )
{
   try
   {
      string s = "hello";
      f( );
   }
   catch( const std::runtime_error& err )
   {
      std::cout << err. what( ) << "\n";
   }
   return 0;
}
```

# No Way Around the Destructor (2)

```
void f( )
{
   string s = "this is a string";

   string array[] = { "these", "are", "also", "strings" };
      // C-array. Always prefer vector.

   std::vector< string > vect =
      { "these", "are", "even", "more", "strings" };
      // vector is a built-in, dynamic array.

   fail_often( );
}
```

## No Way Around Destructor (3)

```
void fail_often( )
{
   if( rand( ) % 1 )
      throw std::runtime_error( "I failed" );
}
```

## Default Constructor

There can be 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 exists a meaningful default. This means that there must exist one value that is very special among the other values, so special that everyone will agree that it is special.

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 are 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 means 'overwriting an existing value with a new value'.

If the variable has a destructor, one needs to decide what to do with the resources: Either reuse current resources, or clean up, and get new resources.

Often one can say:

> Assignment = Destruction ; Construction.

As with destruction, one should never worry about what is inside fields, because the compiler deals with this question.

## Assignment (2)

In case of **string**, the string variable holds memory:

Assignment s1 = s2:

1. If s1,s2 have different length, then deallocate s1, and reallocate s1 with the length of s2.

2. Copy the characters from s2 to s1.

# Assignment (3)

```
void operator = ( const string& s )
{
   if( len != s. len )
   {
      delete[] p;
      len = s. len;
      p = new char[ len ];
   }

   for( size_t i = 0; i < len; ++ i )
      p[i] = s.p[i];
}
```

## Assignment (4)

There is danger of self-assignment `s = s;`

This operator crashes in case of self assignment:

```
void operator = ( const string& s )
{
   delete[] p;
   len = s. len;
   p = new char[ len ];
   for( size_t i = 0; i < len; ++ i )
      p[i] = s.p[i];
}
```

Because `s` is identical with `*this`, the first `delete[] p`, also deletes `s.p`. Later, the assignment reads from `s.p`.

Moral: Always check what happens in case of self-assignment.

## Assignment (5)

In case of tree-like structure, there is similar problem with subtree assignment. One sided lists are a kind of linear trees.

```
struct list
{
   double val;   // Implementation is simplified.
   list* next;   // In reality, additional class is needed.

   void operator = ( const list& l )
   {
      // Delete *this, and all its nexts.
      // Make a copy of l.
   }
};

l = *( l. next ); // Causes crash
```

## Assignment (6)

If assignment has form `operator = ( string s )`, there are no problems, but it is inefficient. (An unnecessary copy is made).

This problem can be solved with rvalue references. We come back to this later.

## Assignment (6)

As with CC, one can define any assignment operator that one likes:

```
void operator = ( const char* s );
void operator = ( int x );
   // Convert int to string.
```

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

The language allows that an assignment operator returns something, (so that you can write `a = b = c;`), but I think that this is not a good idea.

## Don't Worry About the Fields

This class needs no special assignment operator:

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

In general, no destructor ⇒ no need for assignment, no need for CC.

# Nullptr vs Pointer to Zero Length Segment

```
char* p1 = nullptr;
char* p2 = new char[0];
```

What is difference? You cannot use `*p` on both of them. `p1==p2` has undefined behaviour. `delete[] p` does the same.

There is no need to distinguish between **nullptr** and pointer to zero length fragment.

This can be used in default constructor of **string**.

## Other Implementations of 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.

## Final Remarks

Don't use **malloc** and **free**. These are $C$-relicts.

Don't overuse **new**, **delete**. People who know Java tend to do this. In good $C^{++}$, there is no distinction (for the user) between primitive and defined types.