valstat for callable constructors

Document Number:	**PXXXX**
Date	2021-APR-??
Potential Audience	WG21
Author	Dusan B. Jovanovic (dbj@dbj.org)

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." -- C.A.R. Hoare

1. Abstract

This paper describes very simple C++callable constructors proposal, by imposing a core language change that is fundamental but in the same time non breaking.

For the actual implementation the VALSTAT protocol definition is deployed.

2. Motivation

Standard C++ constructors only way to signal the outcome, is to throw an exception. Projects where exception are mandated to be non existent are using factory methods to create class instances. That has effectively created another C++ dialect and fragmented the community.

Valstat structure, sometimes called "valstat carrier", is a record made of two fields: value and status. Field can be empty or occupied.

Using valstat carrier as a return type we can navigate a lightweight and simple route around tha inability of constructors to return values.

Cost of this mechanism is almost zero.

3. Synopsis

Vocabulary

Term	Meaning
C Ctor	
callable constructor	Callable Constructor
CC Class	Class having one or more Callable Constructors
CC Struct	Struct having one or more Callable Constructors

CC Class can have a mixture of callable and non-callable (aka "normal") constructors.

Specimen CC Class:

```
// CC Struct
struct person
{
struct valstat { person * value; const char * status; };

std::string name ; // data

~person () { if (! name.empty() ) name = ""; }

person person ( string new_name_) : name (new_name_)
{
 return valstat{ *this, "person constructed" };
}

person () noexcept : name("")
{
    return valstat{ *this, "default person constructed" };
}

person ( person const & ) noexcept = delete ;
person & operator = ( person const & ) noexcept = delete ;

person & operator = ( person && other) {
    using namespace std;
    std::swap(this->name, other.name) ;
    return *this;
}
person ( person && another_ ) noexcept : name ( another_.name ) {
   another_.name = "" ;
}

} ; // eof person
```

Comments, rules and explanations

```
struct person
{
```

3.1. Rule: callable constructor return type must be nested valstat type of the same class

value field type must be the pointer to the type being constructed.

```
  // T::valstat declaration
  // C Ctor has to return instance of this struct
  struct valstat { person * value; const char * status; };
```

Value of the T::valstat field must not be freed. T::valstat field can point to non existent T instance.

3.2. Rule: there can be one or more nested types following the rule 1

Callable *person* constructors must return *person::valstat*.

3.3. Rule: callable constructors have the same signature as any other non callable constructors.

All the other language rules for constructors do apply.

If constructor has no return type in its implementation it can not be called; it is a "normal" constructor.

3.4. Rule: Callable constructors can not be declared explicit.

Callable constructors:

```
    std::string name ; // data

   person () noexcept : name("")
   {
/* person::valstat in an OK state */
       return valstat{ *this, "default person constructed" };
   }

   person person ( string new_name_) : name (new_name_)
   {
/* person::valstat in an OK state */
```

```
        return valstat{ *this, "person constructed" };
    }
```

Constructor return type rules are the same rules as for any other function except that return type is not declared.

```
// destructors were always callable
~person () { if (! empty() ) name = ""; }
```

3.5. Rule: compiler generated constructors and assignments are not callable.

```
person & operator = ( person const & ) noexcept = default ;
person & operator = ( person && ) noexcept = default ;
```

3.6. Rule: Assignments can not be callable in any case

Compiler deleted constructors and assignments are obviously not callable.

Copy or move constructor, signature is unchanged. Declarations of constructors and assignments are same as ever before.

```
person ( person const & another_ ) noexcept = delete ;
```

3.7. Side effect: noexcept ctor is certain at last

Until now noexcept constructors have been a best guess. In this scenario noexcept might be finally a true mark of no exceptions thrown. At least when callable constructors are concerned.

3.8. Callable Constructors and Unfinished instances

In case of returning a valstat prematurely i.e.from an unfinished object ctor, `this` will be a nullptr. Compiler should be able to catch that as an error.

4. The usage

4.1. Legacy

One is free to construct T on heap and do traditional constructions as ever before. Thus the legacy code is unaffected.

```
// legacy instantiation syntax
// using a constructor as ever before
// valstat is not returned
person p() ;
```

```
// ignoring the valstat returned
return person() ;
```

Only explicit assignment to T::valstat will provoke callable constructors return to be passed out.

```
// argument type is a person instance
// not person::valstat
void login ( person p);

login( person() );

// argument type is a person::valstat instance
void check ( person::valstat const & pv ) {
  // value might point to a temporary object
  if ( pv.value )
   logging() << pv.value->name << " checks OK";
}
// constructor return passed as argument value
check( person() ) ;
```

Canonical valstat return from a callable constructor

```
person::valstat pv = person();
```

Or

```
auto [ value, status ]  = person();
```

4.2. Rule: If T is created on the heap, compiler ignores returns from Callable Constructors .

```
person * pp = new person();
```

Compiler should be able to resolve the above easily.

4.3. valstat two step decoding

valstat structure carries information. Information = state + data. One can naturally decode the full valstat information returned as a result of a constructor call.

```
// using the callable constructor
// return type is: person::valstat
// pp is a person instance pointer
// status type is "const char *"
auto [ pp , status ] = person() ;

// the standard valstat information two steps decoding
if ( pp ) {
    std::cout << "new person is created: " << pp->name ;
} else {
  std::cout << "person default constructor has failed." ;
}

if ( status ) {
    std::cout << "status is: " << status ;
} else
    std::cout << "status is empty" ;
}
```

4.4. Rule: If constructor does not contain return statements, it can not be called.

```
// compilation error -- non callable constructor
auto [ p , status ] = person("Mr Person") ;
```

5. Conclusion

Can this mechanism be abused? Anything in C++ can be abused. Standard constructor paradigm can be abused. Callable constructors can be abused too.

This language extension would not break any existing code.

6. Appendix: The valstat nano course

T::valstat is a mandated type whose instance is returned from a callable constructor of a type T.

Instances of that type are used to carry the full information. One of the four states as described in VALSTAT and the data, if any.

Combination of value *and* status occupancies is giving four possible states.

State Tag	Value	op	Status
Info	Occupied	AND	Occupied
OK	Occupied	AND	Empty
Error	Empty	AND	Occupied
Empty	Empty	AND	Empty

> Valstat type is the information carrier
>
> Information = state + data

The link to the full VALSTAT document.

EOF