Null Values in Programming Languages

Christian Heinlein
Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract

Even though many programming languages support a distinguished null *or* nil value *for* pointer *or* reference types *to indicate that a pointer or reference currently does not refer to any object, none of these languages supports null values as a general concept for* all *types of the language. This inability to explicitly express a null or* missing value *raises several questions when defining the semantics of a programming language, e. g., whether it is allowed and well-defined to use uninitialized variables or to omit explicit return statements in functions. Introducing null values as a general concept in programming languages not only solves such problems in a natural and straightforward way, but also leads to some interesting and unexpected consequences with respect to the interpretation of null pointers: Reading the value referenced by a null pointer should return null, while writing to such a value should simply do nothing. Besides developing a conceptual basis, the paper also presents concrete approaches to implement null values in programming languages.*

Keywords: Null values, pointers and references, dereferencing.

1. Introduction

Many programming languages support a distinguished *null* or *nil value* for *pointer* or *reference types* to indicate that a pointer or reference currently does not refer to any object. Examples include `NIL` in Pascal [4], Modula-2 [10], and Oberon(-2) [11], `0` (when used as a pointer value) in C and C++ [8], and `null` in Java [3]. None of these (and many other) languages, however, supports null values for other types (in particular not for basic types such as integers or characters), i. e., "values" indicating the *absence* of any real value. This inability to explicitly express a null or *missing value* raises several questions when defining the semantics of a programming language, e. g., whether variables must be explicitly initialized by the programmer before they are used (as in Java), whether the compiler or run time system provides a well-defined default value for uninitialized variables (as for global variables in C and C++), or whether uninitialized variables will have completely undefined values (as local variables in C and C++). While none of these common alternatives is completely satisfactory (cf. Sec. 2.2), defining an uninitialized variable to simply possess *no value* appears rather natural and straightforward. A similar idea can be found in the database query language SQL [6], where data items might be `NULL` to indicate a missing or unknown value. Based on these observations, the basic proposal of this paper is to support the concept of null or missing values for every data type of a programming language, including basic types such as integers or characters, user-defined record or class types, and pointer or reference types.

After some preliminary definitions (Sec. 2.1), Sec. 2 examines the positive consequences of this approach for the treatment of uninitialized variables (Sec. 2.2) and missing return statements (Sec. 2.3), explains how to detect (Sec. 2.4) and use null values (Sec. 2.5), and finally explains some unexpected consequences for dereferencing null pointers (Sec. 2.6). Following these conceptual considerations, Sec. 3 describes general approaches to implement null values for basic types such as floating point numbers (Sec. 3.1), integer numbers (Sec. 3.2), and characters (Sec. 3.3), as well as pointer types (Sec. 3.4). Based on these general ideas, Sec. 4 describes a user-level implementation of null values in C++, i. e, an implementation that does not require any compiler modifications. Finally, Sec. 5 concludes the paper by summarizing the concept and comparing it with related work.

2. Conceptual Considerations

2.1 Preparations

Mathematically speaking, a *type* of a programming language describes a *set of values*, e. g., the type `int` in C or C++ describes a particular subset of the mathematical set of integral numbers, while a user-defined type such as `Person` might describe the set of all persons used in a program. Usually, a *variable* of some type represents a storage cell that is capable to store exactly one value of its type at a time. Similarly, an *expression* of some type evaluates to exactly one value of its type (unless its evaluation is aborted, e. g., by raising an exception).

To incorporate the concept of null values, these definitions are extended to say that a variable of some type might either contain a particular value of that type *or no value at all*, while an expression of some type evaluates to either a particular value of that type *or to no value at all*. In the latter cases, one might also say slightly inaccurately that the variable or expression *has the value null*, even though "null" is actually not a value of any type.

To give a visual illustration, one might think of a variable as a box containing either a particular value or being empty.

2.2 Uninitialized Variables

The fact that a variable is explicitly capable of containing no value at all, can be exploited in a very natural and straightforward manner to define that variables which are not explicitly initialized by the programmer simply contain *no value*, i. e., null.

Since null is a well-defined "value," using such a variable results in well-defined and reproducible behaviour (cf. Sec. 2.5), in contrast to using uninitialized local variables, e. g., in C or C++. On the other hand, since null is actually not a value at all, it can be distinguished from all real values of the variable's type (cf. Sec. 2.4), in particular from numerical zeros or "null bytes" (which are actually "zero bytes"). Using such a distinguished null value seems to be more appropriate than using some arbitrary real value such as zero

for global variables in C and C++. Furthermore, allowing a programmer to declare and use uninitialized variables and to implicitly initialize them to null, instead of rejecting them by the compiler (as in Java), significantly simplifies both the definition and the implementation of the language. For instance, the Java Language Specification [3] contains a whole non-trivial chapter to precisely define the notion of "definite assignment" needed by the compiler to reject potentially uninitialized variables. In addition to its complexity, this approach suffers from the very practical problem that a compiler can only perform a *conservative* flow analysis of a program, which occasionally leads to situations where a programmer *knows* (and could prove) that a variable *will* be definitely assigned a value before its first use, but the compiler *thinks* that it *might* not. In particular, `switch` statements without a `default` branch are a frequent source of annoyance, since such statements usually do not cover all theoretically possible cases, even if they contain `case` labels for all values occurring in practice.

To summarize, implicitly initializing a variable that is not explicitly initialized by the programmer to *no value* leads to a very simple and natural semantics in practice while avoiding sophisticated yet incomplete flow analyses by a compiler. (Nevertheless, a compiler might still use such analyses to avoid unnecessary implicit initializations of variables which will be definitely assigned a real value before their first use.)

2.3 Missing Return Statements

Similar problems and questions to those discussed above arise with respect to return statements: What shall happen at compile and/or run time, if a procedure, function, or method (whatever terminology is used in a particular language) is declared to return a value of some type, but does not contain any or does not execute a return statement at run time?

While the compilers of many programming languages are required (by their language definitions) to accept this (possibly issuing a warning), and the behaviour at run time is completely undefined, a Java compiler again has to perform a conservative flow analysis to guarantee that a return statement will be executed under all (theoretical) circumstances. While an undefined behaviour at run time is, of course, undesirable, the Java solution is unsatisfactory, too, since again the compiler is not able to detect all legal cases. If, for example, the return value of a method is determined in a `switch` statement with several cases, each containing an appropriate return statement, one must frequently add a logically unnecessary `default` branch containing a dummy return statement in order to get the program compiled without errors.

Here, the fact that an expression might explicitly possess *no value at all*, can be exploited in a natural and straightforward manner to define that functions that do not explicitly return a value by executing a return statement implicitly return *no value*, i. e., null. Similar to the solution for uninitialized variables, this approach avoids the burden for the language designer to precisely define when a sequence of statements "can complete normally" [3][1], the burden for the compiler (implementer) to perform the necessary flow analysis, and the burden for the normal programmer to write some actually unnecessary return statements in order to be able to successfully compile his programs.

Furthermore, it might be semantically useful for a function that has been declared with a return type (other than `void`) to explicitly return *no value* by executing a return statement without an accompanying expression. For example, a function reading an input character might naturally return null (instead of, e. g., `-1`) if it has reached the end of the input stream, while a function searching for a character in a string and returning its position, might naturally return null (instead of, e. g., `-1`) if the character is not present. Similarly, a function returning the character at a specified position of a string, might return null (instead of throwing, e. g., an `IndexOutOfBoundsException`) if the position is invalid, while a function returning the first element of a container might return null (instead of throwing a `NoSuchElementException`) if the container is empty (cf. Sec. 2.6).

2.4 Detecting Null Values

If variables and expressions might possess no value, it is necessary for a programmer to check for this exceptional case. Syntactically, one might think of various different possibilities to perform such a test, e. g., using standard comparison operators such as `==` or `!=` to compare a value with an explicit `null` value (as in Java), using special `is null` or `is not null` constructs (as in SQL), or to implicitly convert expressions of any type to Boolean values where appropriate by interpreting all real values as `true` and null as `false` (as with pointers in C and C++).

The latter approach reflects the fact that a variable or expression of some type `T` might conceptually be viewed as a pair consisting of a Boolean indicator and an actual value of type `T` that is meaningful only if the indicator's value is `true`. In this interpretation, the conversion of any value to a Boolean value simply returns the value of the indicator.

When using this approach, variables and expressions of any type might be used as conditions of conditional and repetition statements, which is rather convenient in practice. In contrast to C and C++, however, a numerical value of zero is treated just like any other real value, i. e., its conversion to a Boolean value yields `true`.

As a consequence of interpreting all real values of a type as `true` and null as `false`, it turns out that `true` is the only "real" value of the Boolean type `bool` or `boolean`, while `false` is actually equivalent to null. In particular, the familiar two-valued Boolean algebra is preserved instead of introducing an unusual logical calculus based on three truth values `true`, `false`, and `null` resp. `unknown`, as in SQL [6] (cf. Sec. 5).

2.5 Properties of Null Values

In order to be maximally useful, it should be possible to use null values much like ordinary values without losing their distinguished meaning, however. This means in particular, that null values are *propagated* through all kinds of arithmetic operations (including bit operations such as shift or bitwise complement), i. e., the value of an arithmetic expression is null if one of its operands is null. This is very similar to the behaviour of NaN (not a number) values defined by IEEE floating point arithmetics [2].

Furthermore, a null value is different from and *incomparable to* any real value of some type, i. e., comparing a real value `x` with null using any of the standard comparison operators (`==`, `<`, `>`, `<=`, and `>=`) always returns `false`, while `x !=`

[1] This is in fact missing from the Java Language Specification: While it defines the notions of "normal and abrupt completion of statements," i. e., under which circumstances a statement *does* complete normally or abruptly, it actually does not define when a statement *can* complete normally. Rules similar to those for "definite assignment" would be necessary for that purpose.

`null` returns `true`. The former is a little bit unusual, since, e. g., `x < y` and `x >= y` will both be `false` if one of the operands is null. It should be noted, however, that this is not a contradiction to the well-established two-valued Boolean algebra, but simply a result of the rather obvious fact that a null value is neither equal to nor smaller than nor larger than any real value. (In a partially ordered set, `x < y` and `x >= y` might also be both `false`.)

In contrast to NaN floating point values, which exhibit the rather strange behaviour that NaN is different from NaN (which might be explained by the fact that NaN values might have many different causes), null is considered equal to null, i. e., `null == null` yields `true`.

2.6 Dereferencing Null Pointers

As has been mentioned in Sec. 1, null values for pointer (or reference) types are very common in programming languages. Since such a pointer value does not refer to any object, dereferencing it usually leads to a run time error, such as a `SIGSEGV` (segmentation violation) signal or a `NullPointerException` in Java. However, given the fact that an expression might have no value, it is also possible − and in fact more reasonable and consistent, even though it might appear strange at first sight − to define that dereferencing a null pointer simply returns *no value*, i. e., null. For example, if an `int` pointer `p` in C or C++ is null, the dereferencing operation `*p` should return null, too, instead of causing a `SIGSEGV`. To give another example, if a `Person` variable `p` in Java is null, requesting its name `p.name` − which is actually a combination of a dereferencing and a field selection operation − should also return null instead of throwing a `NullPointerException`.

For the same reason, one might argue that accessing a nonexistent array (or other container) element need (and should) not cause an error such as an `IndexOutOfBoundsException`, but rather should return *no value*, i. e., null.

After a short time of familiarization, these alternative rules turn out to be quite convenient in practice, because they make many explicit checks superfluous. If, for example, a Java class `Person` contains fields `name` of type `String` and `spouse` of type `Person`, one might test, e. g., whether the fifth character in the name of `p`'s spouse is `'a'` simply by writing the condition `if (p.spouse.name.charAt(4) == 'a')`, implying the checks that `p` actually references a person (`p != null`), that this person actually has a spouse (`p.spouse != null`), that the spouse actually has a name (`p.spouse.name != null`), and that this name has at least five characters (`p.spouse.name.length() > 4`). If any of these implied conditions (which are usually omitted in colloquial language, too) does not hold, the subexpression `p.spouse.name.charAt(4)` simply returns `null` which is different from `'a'`.[2]

Furthermore, these alternative rules might lead to more robust programs, since unusual circumstances which have not been anticipated by a programmer and therefore are not explicitly checked (e. g., a person without a name) will not lead to run time errors or program crashes, but in many cases simply to unsatisfied conditions (as in the example above) causing correct program continuation. (Of course, one might argue that the opposite might also happen in practice, i. e., that real programming errors will not lead to a "fail fast" behaviour, but to an incorrect program continuation. Empirical studies would be necessary to assess the probabilities of both cases more precisely.)

If "reading" the value or object referenced by a null pointer does no longer cause a run time error, but exhibits a well-defined behaviour, the question arises whether "writing" to such a value or object (e. g., by assigning a value to a dereferenced null pointer) should still cause an error or whether it is possible to define a meaningful behaviour for that case, too.

When thinking of a null pointer as pointing to "nowhere," a value written to this "place" should simply disappear without trace, i. e., such an operation should simply do nothing. Such a behaviour is very similar to that of the Unix pseudo file `/dev/null`: While reading from this device always returns null/nothing (i. e., end of file), writing to it simply has no effect. And just like using this device as a data sink turns out to be quite useful in some circumstances (e. g., to fade out the output produced by a program), it might be useful in some circumstances, too, to write to a dereferenced null pointer.

For example, if a function in C shall return multiple values, at most one of them can be returned directly as the function's result, while the others must be conveyed indirectly through variables whose addresses have been passed as arguments. If some of these values are only meaningful to particular clients, other clients might simply want to pass null pointers instead of valid addresses to indicate that they are not interested in the corresponding values. In normal C code, where this technique is rather popular, the implementer of such a function must explicitly check each pointer for not being null before dereferencing it and assigning a value to its target. Given the modified definition above, however, real and null pointers can be treated uniformly, without any need for explicit checks. Again, this might lead to more robust and flexible code, since it allows to pass null pointers even to functions whose implementers have not taken their possibility into account.

Finally, the question remains whether it is possible to reasonably define the case of calling a (dynamically bound) method via a null pointer. Since there is actually no target object of the method call, it is impossible to determine the actual method implementation that shall be executed using normal dynamic scheduling (e. g., via virtual function tables). Since, consequently, there is actually *no* method to execute, the method call should again simply do nothing, i. e., behave like executing an empty method. Furthermore, since an empty method does not contain a return statement and therefore returns nothing (cf. Sec. 2.3), the result of such a method call should be null.

3. General Implementation Approaches

Having laid the conceptual basis for null values as a general concept in programming languages, the current section presents general approaches to implement the concept for different kinds of data types typically occurring in programming languages.

3.1 Floating Point Types

Since the NaN values of IEEE floating point arithmetics [2] exhibit almost the same behaviour as a null or missing value as proposed in this paper − in particular, they are propagated through all arithmetic operations −, it is straightforward as

[2] In Java, one might achieve a similar effect by misusing its exception mechanism: If the simple test is enclosed in a `try` statement that catches `NullPointerException`s and `IndexOutOfBoundsException`s, the explicit checks could be omitted, too.

well as maximally time- and space-efficient to implement null floating point values as NaN values. The only operators which have to be modified are the comparisons `==`, `<=`, `>=`, and `!=`, since null should be equal to null (cf. Sec. 2.5) while NaN is always different from NaN.

3.2 Integer Types

To implement integer types with null values, there are several alternatives with different run time and storage overheads.

The most space-efficient solution would be to exclude a particular value of the type (e. g., the smallest available integer value) from the set of regular values and use it to represent a null resp. missing value. While this solution does not require any additional storage, it incurs substantial run time overhead since every arithmetic operation on the type has to check its operands for this special null value before performing the real operation and to return the null value if at least one of the operands is equal to it. Similarly, every comparison operation has to check its operands before performing the real comparison.

Alternatively, a value of a type might be represented as a pair consisting of a Boolean indicator and a "real value" of the type, where the indicator is used to distinguish real from null values: If it is `false`, the "compound value" is interpreted as null, actually ignoring the real value, while otherwise the real value is used.

To perform arithmetic operations on such compound values, the real operation is simply performed on the real values, while the indicators are combined by a logical AND operation, without the necessity to perform any additional checks. To be most space-efficient, the indicator could be represented as a single bit which is "stolen" from the regular representation of the type (reducing, e. g., 32 to 31 bit integer values, which is irrelevant for most practical applications). Ideally, the compound arithmetic operation described above (e. g., the normal arithmetic operation on the 31 numerical bits plus the logical AND operation on the indicator bit) should be performed by a single tailored hardware instruction to avoid any run time overhead. Since off-the-shelf processors do not support such operations, however, it is usually necessary to implement them in software. In that case, the space-efficient representation of the indicator as a single bit turns out to cause rather high run time overhead since additional bit operations are required to split the compound operands into real values and indicator bits as well as to combine the resulting real value and indicator bit into a compound result.

A third alternative, which might look strange at first glance, but turns out to be a reasonable tradeoff in practice, is to represent integer values as corresponding floating point values and null as NaN (cf. Sec. 3.1). Using, e. g., 32 bit IEEE floating point values allows to represent 24 bit integer values without loss of information, which is sufficient for most practical applications. Depending on the processor architecture, performing a single floating point operation might be as fast as or even faster than performing an integer operation plus a logical operation as described before.

3.3 Character Types

If character types are not used as synonyms for `byte` (which might possess any value from 0 to 255), but rather represent real character values, the first solution described above for integer types appears to be most appropriate: A particular value (e. g., 128) that does not represent a real character of the underlying character set can be used to represent a null or missing value.

Since arithmetic operations on character values are rather rare (if supported at all), the efficiency argument mentioned above does not hold here. On the other hand, comparisons for (un)equality are maximally simple and efficient since they simply have to perform a bitwise comparison. Other comparisons, which are partially artificial since, e. g., a digit and a letter are actually incomparable, could be supported by checking both operands for null before performing a numeric comparison.

3.4 Pointer Types

Since every programming language that supports pointer (or reference) types also supports the notion of null pointers, no additional implementation effort seems to be necessary for these types. However, the unusual behaviour mentioned in Sec. 2.6 of dereferencing a null pointer requires some modifications to standard implementations: If a null pointer is represented as address zero (or any other illegal address), reading a value from this address must be treated differently from reading a value from a legal address in order to avoid a segmentation fault or similar error. Instead, a null value (of the pointer's target type) must be returned in that case. Similarly, writing to this particular address must be explicitly avoided. Finally, calling a method via this address must not execute any method at all, but rather return a null value (of the method's result type).

Furthermore, adding the offset of a record field to the address representing null pointers should have no effect, too, i. e., return the same address, since accessing a field of a record referenced by a pointer is a dereferencing operation, too, that should exhibit the same behaviour with respect to null pointers as other dereferencing operations.

4. User-Level Implementation in C++

Having sketched general implementation approaches for types with null values in the previous section, the current section presents a prototypical user-level implementation in C++, i. e., an implementation that does not require any compiler modifications. Therefore, this implementation can be used to experiment and gain experience with the concepts outlined in Sec. 2 with rather little effort. For that purpose, the fact that this implementation is not maximally efficient and even the fact that some parts are not 100% portable is deemed less important.

4.1 Basic Types

The basic idea is to provide and use wrapper types for all C++ basic types, e. g., `integer` for `int`, `character` for `char`, etc., and to provide overloaded definitions of all built-in operators (such as `+`, `-`, etc.) for these wrapper types. Furthermore, implicit conversions between a wrapper type and its corresponding basic type are defined to allow, e. g., `int` literals to be used as `integer` values. Finally, implicit conversions between different wrapper types reflecting the corresponding conversions between their base types are defined.

By defining a parameter-less constructor for each wrapper type that initializes its object as null, variables of that type which are not explicitly initialized will get implicitly initialized with null by this constructor [8]. To test for null values, a conversion operator to the basic type `bool` is provided that returns `true` for real values and `false` for null.

Figure 1 shows the basic skeleton of such a wrapper type and the accompanying operator definitions using `integer` as an example.

```
// Wrapper type for int.
struct integer {
  // Internal representation.
  float val;

  // Null value representation as NaN.
  static const float null = 0/0;

  // Default initialization as null.
  integer () : val(null) {}

  // Implicit conversion from int value.
  integer (int i) : val(i) {}

  // Implicit conversion to bool.
  operator bool () const {
    // Return true if val is not NaN.
    return !isnan(val);
  }

  // Explicit construction from float value
  // (needed by operator implementations).
  explicit integer (float f) : val(f) {}

  // Copy constructor catching read access
  // to dereferenced null pointers.
  integer (const integer& i)
  : val(&i ? i.val : null) {}

  // Assignment operator catching read
  // and write access to dereferenced
  // null pointers.
  integer& operator= (const integer& i) {
    if (this) val = &i ? i.val : null;
    return *this;
  }
};

// Arithmetic operations.
integer operator+ (integer x, integer y) {
  return integer(x.val + y.val);
}
integer operator- (integer x, integer y) {
  return integer(x.val - y.val);
}
......

// Comparison operations.
bool operator< (integer x, integer y) {
  return x.val < y.val;
}
bool operator== (integer x, integer y) {
  // Return true, if float values are equal
  // or both values are NaN.
  return x.val == y.val
  || (isnan(x.val) && isnan(y.val));
}
......
```

Figure 1: Wrapper type `integer` for `int`

To implement the unusual behaviour of dereferencing null pointers described in Sec. 2.6, a copy constructor is provided that is implicitly called whenever an `integer` object is copied, i. e., read. Since it receives a reference `i` (of type `const integer&`) to the source object, it is able to determine its address `&i` (of type `const integer*`) and check whether it is a null pointer, before accessing its internal representation `i.val`, i. e., before actually performing the dereferencing operation. If the pointer is null, a null object is constructed and returned and the dereferencing operation (which would cause the program to crash) is avoided.

Similarly, an overloaded assignment operator is provided that is implicitly called whenever an `integer` object is assigned a new value, i. e., is written. Like the copy constructor mentioned before, it receives a reference `i` to the source object (i. e, the RHS of the assignment), while the implicit parameter `this` (of type `integer*`) represents a pointer to the target object (i. e, the LHS of the assignment). Therefore, it is possible to check whether `this` is a null pointer, before accessing its internal representation `val`, i. e., before actually performing the dereferencing operation on the LHS. If the pointer is null, the assignment operator simply does nothing. Since the RHS might be a dereferenced null pointer, too, the same check as in the copy constructor is used here, too, if an assignment is actually performed.

These definitions cause code such as:

```
integer* p = 0;
integer i = *p;
*p = 5;
```

which would normally crash to behave well-defined, i. e., the initialization of `i` with the dereferenced null pointer `*p` produces a null `integer` value, while the assignment to the dereferenced null pointer `*p` simply does nothing.

To support not only direct dereferencing operations using the `*` operator, but also those using the `->` operator (e. g., `p->x` where `p` is a pointer to a structure possessing an element `x`), pointer values resulting from incrementing a null pointer by an offset of such an element should be treated like null pointers, too. If null pointers are represented as address zero (which is common), the resulting pointers have "small" values when interpreted as integer values, where the precise definition of "small" is "less than the size `S` of the largest data type appearing in a program." Therefore, checks such as `if (this)` must be replaced by `if (this > S)`. Since such "small" addresses do not occur as real addresses of data in typical environments (since the program's code resides at these addresses), treating them like null pointers does not cause any trouble in practice.[3]

It should be noted, however, that dereferencing null pointers behaves only well-defined if the target type of the dereferencing operator is "null-aware," i. e., possesses a copy constructor and assignment operator as described above.

4.2 Pointer Types

If a null pointer to another pointer (or a null pointer to a data structure containing pointers) should be safely dereferencable, pointer types themselves must be null-aware, i. e., possess appropriate copy constructors and assignment operators. This can be achieved by providing and using a generic wrapper type `pointer<T>` for the basic pointer types `T*` (cf. Fig. 2).

4.3 User-Defined Types

If a programmer defines a new class (or structure), it possesses an implicitly defined default constructor, copy constructor,

[3] Strictly speaking, of course, assumptions such as null pointers are represented as address zero and "small" addresses do not occur as real addresses of data, are implementation-dependent and not portable.

```
template <typename T>
struct pointer {
  // Internal representation.
  T* ptr;

  // Null value representation.
  static const int null = 0;

  // Default initialization as null.
  pointer () : ptr(null) {}

  // Implicit conversion from real pointer.
  pointer (T* p) : ptr(p) {}

  // Implicit conversion to bool.
  operator bool () const { return ptr; }

  // Copy constructor catching read access
  // to dereferenced null pointers.
  pointer (const pointer& p)
  : ptr(&p ? p.ptr : null) {}

  // Assignment operator catching read
  // and write access to dereferenced
  // null pointers.
  pointer& operator= (const pointer& p) {
    if (this) ptr = &p ? p.ptr : null;
    return *this;
  }

  // Dereferencing operators.
  T& operator* () { return *ptr; }
  const T& operator* () const {
    return *ptr;
  }
  T* operator-> () { return ptr; }
  const T* operator-> () const {
    return ptr;
  }
};
```

Figure 2: Generic wrapper type `pointer<T>` for `T*`

and assignment operator, which simply call the corresponding functions for all components (i. e., base classes, if any, and data elements) of the new type (cf. [8]). Therefore, if all components of the type are null-aware, the new type is automatically null-aware, too, i. e., variables of this type which are not explicitly initialized will be implicitly initialized as null (by initializing all components as null) and dereferencing null pointers to this type will be well-defined.

The only function that has to be provided explicitly, is an appropriate conversion operator to `bool`. Typically, this checks some or all components of the type for being null and combines the results by a logical AND or OR operation, depending on the particular semantics of the type. For example, a `Person` object might be considered null if an essential component such as `name` is null, while other components such as `spouse` will not be taken into account. On the other hand, a `Rectangle` object possessing `width` and `height` components might be considered null if at least one of its components is null.

If the class possesses virtual member functions (i. e., dynamically bound methods), these must be wrapped by non-virtual functions (i. e., statically bound methods), since the former cannot be called via null pointers. Such a non-virtual wrapper function must check whether `this` is null and either return null (if it is) or call the corresponding virtual function. If the wrapper function possesses the same name as the original virtual function and the latter is renamed to some unique internal name, the presence of the wrapper function remains transparent for clients calling the function. Furthermore, it is possible to create the wrapper function automatically by a precompiler in order to completely hide its presence from programmers.

To give a simple example of a user-defined type, Fig. 3 shows the definition of `struct Rectangle` possessing two `integer` components `width` and `height`. As described above, the conversion operator to `bool` returns true if both components are not null. Conceptually, the type possesses a virtual member function `area` to compute the rectangle's area. To catch calls of this function via null pointers, however, it has been renamed to `area__` and replaced by a non-virtual wrapper function `area` that checks whether `this` is null before calling `area__`. If it is null, a null `integer` value (created by calling `integer`'s default constructor) is returned instead. (This is explicitly necessary, since C++ does not implement the rule postulated in Sec. 2.3 that a function that does not explicitly return a value implicitly returns null. However, it could be implemented rather easily by a precompiler, too.)

```
struct Rectangle {
  // Components.
  integer width, height;

  // Implicit conversion to bool.
  operator bool () {
    // Return true if both components
    // are not null.
    return width && height;
  }

  // Virtual member function computing area.
  virtual integer area__ () {
    return width * height;
  }

  // Non-virtual wrapper function
  // catching dereferenced null pointers.
  integer area () {
    if (this) return area__();
    else return integer();
  }
};
```

Figure 3: Simple example of a user-defined type

If a subclass of `Rectangle` shall override the member function `area`, it actually must provide a redefinition of `area__`. Again, this renaming might be performed by a precompiler.

5. Conclusion

Even though programming languages offering pointers or references provide the notion of a null pointer or reference, they usually lack the concept of general null resp. missing values for all types of the language. Starting with the observation that such a concept would make some aspects of a language more convenient and reliable − variables which are not explicitly initialized are initialized to null and functions which do not explicitly return a value return null −, its strict application led to some interesting and unexpected consequences with respect to the interpretation of null pointers: Reading the value referenced by a null pointer should return null, while writing to such a value should simply do nothing. Again, this makes programming both more convenient (since

many explicit checks for null pointers become obsolete) and more reliable (since omitted checks will not lead to run time errors).

Some of the ideas presented in this paper can be found in existing languages, too, in particular in various scripting languages. For example, variables which are not explicitly initialized are usually initialized to a well-defined default value in these languages instead of leaving their value undefined or random (as, e. g., in C and C++). Similarly, functions which are not explicitly returning a value usually return a well-defined default value, too.

While in some languages (e. g., Awk [1] and Perl [9]), this default value is a regular value such as zero or an empty string, others provide a distinguished value such as `nil` (e. g., Lisp [7]) or `None` (e. g., Python [5]) that is different from all real values. In contrast to the null values proposed in this paper, however, which can be used just like regular values in operations such as arithmetics and comparisons, these distinguished values usually cause run time errors when used that way. The NaN values of IEEE floating point arithmetics [2] are an exception that demonstrates the usefulness of propagating null values through arithmetic operations.

Another exception are `NULL` values in relational database systems and their treatment in the query language SQL [6]. While their behaviour in arithmetic expressions is identical to that proposed in this paper, their interpretation in Boolean expressions and comparisons differs substantially. Based on the general view that a `NULL` value represents an *unknown* (rather than a missing) value, it is consistent to define the result of comparing a regular value with `NULL` as *unknown* (rather then `false`), too, and consequently also to define the value of a Boolean expression containing one or more operands with unknown values as unknown.

In contrast, a null value as proposed in this paper actually denotes *no value at all* (causing the notion of "null value" to be actually a contradiction in itself). Based on this view, it is reasonable to define that null is definitely different from and incomparable to all real values, i. e., corresponding comparisons simply return `false` instead of an unknown Boolean value. An important side effect of this decision is the fact that the well-known and familiar two-valued Boolean algebra is still applicable, i. e., the rather strange and unusual three-valued logic of SQL can be avoided.

References

[1] A. V. Aho, B. W. Kernighan, P. J. Weinberger: "Awk − A Pattern Scanning and Processing Language." *Software—Practice and Experience* 9 (4) April 1979, 267−279.

[2] D. Goldberg: "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys* 23 (1) March 1991, 5−48.

[3] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[4] K. Jensen, N. Wirth: *Pascal User Manual and Report* (Second Edition). Springer-Verlag, New York, 1978.

[5] M. Lutz: *Programming Python*. O'Reilly, Beijing, 2001.

[6] J. Melton, A. R. Simon: *SQL:1999. Understanding Relational Language Components*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[7] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[8] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[9] L. Wall, T. Christiansen, J. Orwant: *Programming Perl* (3rd Edition). O'Reilly, 2000.

[10] N. Wirth: *Programming in Modula-2*. Springer-Verlag, 1982.

[11] N. Wirth: "The Programming Language Oberon." *Software—Practice and Experience* 18 (7) July 1988, 671−690.