Joint NCC & IIS Bull., Comp. Science, 18 (2002), 115–142

Extended Pascal to C++ converter

V. A. Markin, S. V. Maslov, A. A. Sulimov

Introduction

The methods of compiler (and converter as a special case) development are sufficiently investigated and described [2, 3]. There are a number of converters [4–6] from the standard Pascal language [7] to C/C++ languages [8].

In this work, some translation schemes (that are of theoretical interest, in the authors' opinion) elaborated for a converter from an essential Pascal extension (later M-Pascal) to C++ are described. The converter has been developed by order of a large telecommunication company, and it must not only translate input M-Pascal code to functionally equivalent C++ code, but also meet some requirements. M-Pascal language extensions and requirements for the converter have essentially influenced the development of translation schemes.

In this work, the following extensions of M-Pascal are considered:

- Variable initialization, which is syntactically close to the initialization in the C language, but has slightly different semantics.
- Conditional Compilation, which differs from the conditional compilation in C/C++ in the program constants usage and absence of the preprocessor.
- Statements or declarations can be copied from a library by using the *include* option.
- The strict ordering of definitions in the standard Pascal has been weakened.
- "Integers" are represented by two different predefined types, *integer* and *longint*.

The most important requirements to the converter are the following:

- Data Layout Preservation — binary correspondence between M-Pascal and C++ data types;
- preservation of Conditional Compilation directives;

- textual similarity of M-Pascal and C++ codes, e.g., textual representation of array index expressions (the ranges and index types in M-Pascal differ from those in C++) and preservation of copy and comparison operations of structured types.

Moreover, the following is desirable:

- encapsulation — capability of hiding some aspects of data representation. For example, the M-Pascal record fields become private members in C++ classes and one has access to them through methods.

In Section 1, a conception of Data Layout Preservation (DLP) is presented. Section 2 describes the general approaches to initialization. Section 3 is an overview of the translation schemes of the most interesting aggregate M-Pascal types: records, arrays and sets. In Section 4, the translation schemes of nested procedures in M-Pascal are given. In Section 5, the translation scheme of conditional compilation directives is described, the sets of translated and non-translated Conditional Compilation Clauses (CCC) are defined.

All the examples are concerned with the M-Pascal compiler on M68000 and the GCC (GNU) compiler on SPARC station.

1. Data layout preservation

One of the main requirements to the converter is data layout preservation (DLP). This term means the following:

- preservation of the size of each data item, including every component of its decomposition;
- preservation of data items locations (up to bits) in the memory (this includes alignment issues and overlapping in variant records).

Alignment is an extension of the memory allocated for all values of a data type up to the boundary of some unit of memory — byte, word (2 bytes) or double word (4 bytes).

Depending on the processor, data layout can be such that:

- the most significant bit can be either the highest or the lowest bit in a byte;
- the most significant byte can be either at the highest or at the lowest address in a word.

The implementation of DLP allows ensured compatibility of modules of composite software-hardware systems when some components are being modified (re-engineered), for example, when some modules written in one program language are translated into another language or one part of hardware is replaced by another.

Further, in this part we describe the main distinctions in data layout and alignment between the M-Pascal compiler on M68000 platform and the GCC (GNU) compiler on SPARC Station.

Distinction 1 (unpacked data types)
M-Pascal: the **boolean** type is allocated in a byte.
GCC: the standard **bool** type is allocated in 4 bytes.
For DLP of the **boolean** type

```
 typedef unsigned char boolean n;
```

is used.

Distinction 2 (unpacked data types)
M-Pascal: Any field that requires more than a byte (integer — 2 bytes, longint — 4 bytes, pointer — 4 bytes, set — 32 bytes, the enumerated types with the number of members more than 256) is always allocated at an even offset (in bytes) from the start of the record.

Example 1.1

```
T1= record     { byte offset   byte size }
   a:char;     {     0             1     }
   b:longint;  {     2             4     }
   c:char      {     6             1     }
end;
   (sizeof(T1) =  7 bytes)
```

Array/record is word aligned (the word aligned means 2 byte aligned in this paper), but the size of a record can have an odd number of bytes, i.e., word alignment of a record type is executed only if this type is used in other record or array types and only if it is not the last field in record types.

Example 1.2

```
T2=record a:integer; b:char; end; {sizeof(T2) =  3 bytes}
T21=record a,b:T2 end;            {sizeof(T21) = 7 bytes}
T22=array [1..2] of T1;           {sizeof(T22) = 8 bytes}
```

C++: The fields of classes are allocated in successive bytes from low to high memory addresses and follow the same alignment and size rules, as their corresponding types require.

Any field that requires more than one byte and less than or equal to 2 bytes (e.g., **short**) is always located at an even offset from the start of the class.

Example 1.3

```
struct T3 {     /* byte offset byte size */
    char a;     /*     0           1     */
    short b;    /*     2           2     */
    char c;     /*     4           1     */
}; // sizeof(T6) =  6 bytes
```

Any field that requires more than 2 bytes is always located at an offset, which is a multiple to four, from the start of the record.

Example 1.4

```
struct T4 {     /* byte offset byte size */
    char a;     /*     0           1     */
    int b;      /*     4           4     */
    char c;     /*     8           1     */
}; // sizeof(T7) =  12 bytes
```

Distinction 3 (packed data types)
M-Pascal: No field will be allocated starting in other than bit 0 of a byte, if this allocation would cause the field to extend into the next byte.

Example 1.5

```
T5= packed record a,b,c,d,e:0..7 end;         {19 bits}
sizeof(T5) =  3 bytes
```

The field `c (e)` is entirely contained in the second (third) byte.

GCC: Bit fields of some types (e.g., short) can be allocated in two bytes (in the same word).

Example 1.6

```
struct T6 {
    short a:3;
    short b:3;

    short c:3;
    short d:3;
    short e:3;};//15 bits
sizeof(T6) = 2 bytes
```

Two bits of *c* are allocated in the first byte, the third bit — in the second one.

Distinction 4 (packed data types).
M-Pascal: A packed record or array of the size less than 8 bits can be allocated with other fields in the same byte.

Example 1.7

```
T0 = packed array [0..3] of boolean; {4 bits}
T7 = packed record      { bit offset    bit size }
    a: 0..7;            {     0             3    }
    b: T0;              {     3             4    }
end; { sizeof(T7) =  1 byte }
```

GCC: C++ always allocates the class/array in a separate byte starting in no other bit than 0; it does not allow specifying any bit-field for a field of type class/array (or their part).

Example 1.8

```
boolean  T01[3];
struct T8 {     /* bit offset  bit size    */
    short a:3;  /*     0       3           */
    T01 b;      /*     8       4           */
}   // sizeof(T8) =  2 bytes
```

Translation schemes have been developed taking into account these data layout differences.

2. Initialization

M-Pascal. In contrast to the standard Pascal, variables in M-Pascal can be initialized at the time of their declaration, that is, an initial value can be specified explicitly for variables of any type at any scope of declaration (global scope or procedure/function scope). For example:

```
VAR
A: integer := 1;        { built-in data type}
B: RECORD       {record with variant part}
    ID: (one, two, three);
    CASE Boolean OF
         True: (name: STRING[10]);
         False: ();
```

```
ENDREC := [one, true, 'ONE'];

C: ARRAY [1..5] OF char := ['a','b','c']; {array}
D: PACKED SET OF 1..10 := [1..3, 5];      { set }
```

It should be noticed that the objects, both global and local, not initialized explicitly will be set at zero implicitly by the M-Pascal compiler.

C++. There are two major possibilities of objects initialization in C++: using an object class constructor and using a list of initializers (this feature is rather an inheritance of C). Generally, an object of class T may be initialized using the list of initializers, if only:

- `T` does not define constructors;
- all members of T are public;
- `T` does not have base classes;
- `T` does not have virtual functions.

Therefore, only classic C structures can be initialized by the list of initializers. Moreover, according to the standard of C++ [8], only global scope objects are initialized by zeros, while an initial value of the nested scope objects is undefined, if not specified explicitly.

The syntax of the list of initializers in M-Pascal and C is much alike, which is an essential argument for the constructor-less approach. However, there are a number of notes which make such a scheme impracticable:

- The presence of the following structured types in M-Pascal
  - variant records,
  - sets,
  - arrays,
  - strings and sub-strings

  and the demands to preserve Data Layout, semantics and appearance of code force us to use C++ classes to simulate the types mentioned. Furthermore, it would be desirable to secure the internal data of these classes making it "private" (this would conflict with the second requirement on using the list of initializers).
- There is no possibility to specify an interval (range) in a list of initializers in C++. This means that initialization of sets (i.e VAR a: SET of char := ['a'..'z']) will require a special constructor to be defined (this would conflict with the first requirement on using the list of initializers).

- M-Pascal records are naturally converted into C++ classes/structures. Taking into account that, in the general case, a record can contain a field of type SET, which has a constructor, it comes out that this record cannot be initialized by a list of initializers, being of a non-aggregate type. The following example illustrates this:

```
struct Set {. . . Set() {} };

class Record {
             Set field;
} r = {Set()}; // error: non-aggregates cannot be
               // initialized with initializer-list
```

Since we are initializing C++ classes (which correspond to the M-Pascal records) by constructors, all the fields of these classes must have a constructor (at least, a default one). That is, for **every** structured type in M-Pascal, a C++ class is introduced. All such classes have the following properties:

- Internal representation of data is closed.
- A convenient (and similar to the original) interface is provided (i.e.,the assignment operators for all types, arithmetic operations for sets, and so on)
- Special constructors are defined to handle the initialization.

This approach to initialization gives us a guarantee that local variables, not initialized explicitly, would be implicitly initialized by zeroes — by call of a default constructor, which is a subject to be defined in every such class.

3. Translation schemes

3.1. Arrays

Arrays of M-Pascal are not different from those of the standard Pascal; neither syntax nor semantics is changed. It was decided to translate M-Pascal arrays to C++ classes on account of the following (see Section 2 as well):

- there are no Packed Arrays in C++ language;
- there is a requirement to preserve data layout in arrays (see DLP Section);
- as distinct from C++, there is a bound checking mechanism for the M-Pascal arrays;

- the target C++ code would preserve the text notation of array index expressions;
- it is necessary to preserve the assignment operator and the comparison operation for M-Pascal arrays.

The template C++ class which models M-Pascal arrays is defined as follows:

```
template<class T, long L, long H, bool is word aligned>
class Array {
   uchar storage[(H-L+1)*((is_word_aligned && (sizeof(T)%2)) ?
              sizeof(T)+1:   sizeof(T))];
public:
   T& operator[](long index);
   const T& operator[](long index) const;
   bool operator==( const Array & other ) const;
   bool operator!=( const Array & other ) const;
   Array& operator=(const Array &other);
};
```

Here, the template parameters are:

- Class `T` is the array's element type;
- `L` and `H` are lower and upper array bounds, correspondingly;
- `is_word_aligned` is set to TRUE if the array element is word aligned and the element type is of odd size (see Example 1.2).

3.1.1. Packed arrays

An array with the element size more than 8 bits is not packed. If the element size is smaller than one byte, then it occupies the minimal amount of bits divisible by exponent of 2 required to contain values of the array element. For example, the 3-bit size element occupies 4 bits, the 5-bit size element — 8 bits.

The signature of the C++ class for packed arrays is the following.

```
template<class T,unsigned long B, long L, long H,
                 unsigned long W =
                 (B == 1 ? 1 : (B==2) ? 2 : (B<=4) ? 4 : 8)>
class PackedArray {
   uchar storage[((H-L+1)*W%8 ? 1+(H-L+1)*W/8 : (H-L+1)*W/8)];
public:
   BitRef<T,B> operator[]( long index );
   bool operator==(const PackedArray& other) const;
   bool operator!=(const PackedArray& other) const;
};
```

Because of impossibility to address a bit within a byte in a packed array, an auxilary BitRef class was introduced:

```
template< class T, unsigned long W >
class BitRef {
protected:
   unsigned char* byte; // a pointer to the byte
   unsigned long bit;   // bit position within the byte
public:
   BitRef( unsigned char* byte, unsigned long bit );
   BitRef& operator=(const T& value);
   BitRef& operator=(const BitRef& value);
};
```

The BitRef class contains a pointer to the byte which contains the element of the array and the bit offset of the element inside of this byte. An access to the elements of packed arrays is implemented through the interface of this class.

3.1.2. Array initialization

There are two approaches to implementation of class constructors to initialize the array:

- a constructor with a variable number of parameters;
- a constructor that accepts an object of an auxiliary initializing class.

Let us consider an example:

```
TYPE A = array [0 .. 5] of integer;
var V : A := [128, 256];
```

With the first approach, the declaration of a constructor looks like

```
Array(int num_of_params, ...);
```

where num_of_params is the number of elements in the initializing list, which in the general case is not equal to the number of array elements. In this case, the remaining elements are initialized by zeros:

```
A V(2, 128, 256);
```

This approach uses the mechanism of working with a variable number of parameters (refer to macro va_arg). According to the latest standard of C++, the behavior of the mechanism is undefined if complex types (non POD-types [8]) are passed through ellipsis ('...'). That is, such a behavior of a constructor highly depends on a particular C++ compiler used.

The second approach removes this restriction by using a temporary object for initialization. This object simulates an input stream for a variable number of initial values of array elements. Once the temporary object is filled with the initializers, it is passed to a constructor of Array. For these purposes, a supplementary interface is added to the class Array:

```
class Array {
// ...
      typedef Init<T, 0, L,H, is_word_aligned> Init;
public:
      Init operator << (const T &elem);
      Array (const Init &ar);
// ...
};
```

In this case, the array initialization in C++ looks like

```
A V = A() << 128 << 256;
```

Both approaches do not conflict with each other and are implemented within one template class Array.

3.2. Sets

A set type is by no means a new type in M-Pascal, but it is still worthy to describe the translation scheme for this type because of the global requirements: Data Layout Preservation (see Section 1) and initialization handling.

```
TYPE T = SET OF char;
```

A set is a well-known data container; therefore, there are many different implementations of it. Probably, the most famous of them is the template class std::bitset from Standard Template Library (STL). As there is no proper mechanism to handle Pascal-like initialization, and STL library is highly system-dependent (some bitset implementations restrict the size of a set to be a multiple to four bytes), a new class is proposed to simulate M-Pascal sets.

The base class for all set implementations is the template class Set whose parameters are lower and upper set bounds and its size is determined by M-Pascal data layout rules (see below about packed and unpacked sets). The set itself is represented by a byte-array (storage data) of the corresponding size with bit addressing organized. Besides, all set operations (union, intersection, difference, set comparison and element membership checking) are implemented within the Set class.

```
#define MAX_SET_SIZE 255
#define __BITS_PER_WORDT (8*sizeof(char))
#define __BITSET_WORDS(__n) \
 ((__n)< 1 ? 1 :((__n) + __BITS_PER_WORDT - 1)/__BITS_PER_WORDT)

template <size_t LOW, size_t HIGH, size_t SIZE = MAX_SET_SIZE >
class Set {
private:
    unsigned char storage[__BITSET_WORDS(SIZE)];

public:
    // Interface ...
};
```

3.2.1. Unpacked sets

The size of any unpacked M-Pascal set is equal to 32 bytes, as it contains 256 elements. The class BitSet represents all unpacked sets and is inherited from the base class Set with the third template parameter instantiated by 255 (0 is included in the set):

```
template <size_t LOW, size_t HIGH>
class BitSet: public Set<LOW, HIGH, MAX_SET_SIZE> {
public:
    typedef Set<LOW, HIGH, MAX_SET_SIZE> Base;
    BitSet();
};
```

In the general case, only a part of memory reserved for the BitSet object (32 bytes) is used indeed. The low and high bound checking is carried out dynamically using first two template parameters (LOW and HIGH).

3.2.2. Packed sets

Unlike the case of unpacked sets, the memory size necessary to keep all elements of a packed set depends on the real set bounds, or, to be precise, on its upper bound (HIGH). The M-Pascal compiler does not use the information about the lower bound when determining the size of a set. That is, in the general case similar to unpacked sets, memory is used partially. The size of an unpacked set varies from 1 to 32 bytes according to its upper bound. The class PackedBitSet, inherited from the class Set with the third template parameter equal to the real packed set size, preserves data layout for this type:

```
template <size_t LOW, size_t HIGH>
class PackedBitSet: public Set<LOW, HIGH, HIGH+1 > {
public:
    typedef Set<LOW, HIGH, HIGH+1> Base;
    BitSet();
};
```

Bound checking is carried out just as in the case of unpacked sets.

Since the packed and unpacked sets are inherited from the same base class Set, they can be mixed in any set operations (assignment, relations, difference, union, intersection). Initialization is handled similarly in both packed and unpacked sets. Let us consider initialization of unpacked sets for simplicity.

3.2.3. Set initialization

Any M-Pascal set can be initialized by a set of values or by an interval (range) of values. Correspondingly, two auxiliary initializing classes are introduced.

```
#define ENDSET -1

class SetValues: public Set<0, MAX_SET_SIZE, MAX_SET_SIZE> {
public:
    SetValues(int first, ...);
};

class SetRange: public Set<0, MAX_SET_SIZE, MAX_SET_SIZE> {
public:
SetRange (size_t from, size_t to);
};
```

The class SetValues is used to create an object of the type Set and has a constructor with a variable number of parameters (this is possible, because the Pascal sets cannot contain negative elements and the value ENDSET(-1) can be used as the end of an initializing list). The class SetRange, likewise, is used to create the temporary set which contains the elements of the range.

Thus, during initialization of a set, temporary objects of the type SetValues or SetRange are created first. After that, a temporary set object is being constructed as a union of the constructed temporary set objects and passed as a parameter to the BitSet constructor:

```
template <size_t LOW, size_t HIGH>
class BitSet: public Set<LOW, HIGH, MAX_SET_SIZE> {
public:
    typedef Set<LOW, HIGH, MAX_SET_SIZE> Base;
    BitSet();
    BitSet (const Set<0, MAX_SET_SIZE, MAX_SET_SIZE > &other);
};
```

To correctly initialize M-Pascal sets (to empty the set), the default constructor of the BitSet class is redefined.

Let us consider an example of set initialization:

```
TYPE T = SET OF 1..10;
VAR V: T := [1..3, 5];
VAR V1: PACKED SET OF char;
```

translated to:

```
typedef BitSet<1,10> T;
T V = SetRange(1,3) + SetValues(5, ENDSET);
PackedBitSet<0,255> V1; //default constructor is called
```

3.3. Records

Taking into account the encapsulation request, M-Pascal records are naturally translated into C++ classes with using UNION construction for variant parts.

Example 3.3.1 (see Example 1.5). M-Pascal record type T1

```
  T1= record
      a:char;
      b:longint;
      c:char
  end;
```

can be converted into the C-class:

```
  class T1 {
      unsigned char a;
      int b;
      unsigned char c;
      . . .    //access methods
  };
```

Example 3.3.2. Type T9

```
T9=packed record              { bit offset    bit size  }
    a:boolean;                {    0             1      }
    CASE b: boolean OF        {    1             1      }
      TRUE: (c: '0'..'9';     {    2             6      }
             d:'a'..'z');     {    8             7      }
      FALSE: (e: 'A'..'z');   {    8             7      }
end; {sizeof(T9) =  2 bytes }
```

can be converted into the C-class:

```
class T9 {                    /* bit offset  bit size */
    boolean a:1;              /*     0          1     */
    boolean b:1;              /*     1          1     */
  union {
   struct {
     unsigned char c:6;       /*     8             6  */
     unsigned char d:7;       /*     16            7  */
    };
   unsigned char e:7;         /*     8             7  */
  };
. . .    //access methods
};// sizeof(T9) =  3 bytes
```

The necessity to preserve data layout and initialization features essentially influences the *natural* translation scheme shown above.

3.3.1. Data Layout Preservation for records without variant part

The size of M-Pascal type T1 is equal to 7 bytes, the size of T1 (in C++) is 12 bytes. With the use of GCC compiler extension, *__attribute__((packed))* (attached to an `enum, struct`, or `union` type definition, it specifies that the minimum required memory will be used to represent the type), it is possible to get the following T1 representation:

```
class T1 {                  /* byte offset byte size */
    unsigned char a;        /*     0           1     */
    int b;                  /*     1           4     */
    unsigned char c;        /*     5           1     */
. . .   //access methods
}__attribute  ((packed)); // sizeof(T1) =  6 bytes
```

It is just *enough* to add unnamed field to preserve data layout:

```
class T1 {                  /* byte offset byte size */
     unsigned char a;       /*     0           1     */
     char:8;                /*     1           1     */
     int b;                 /*     2           5     */
     unsigned char c;       /*     6           1     */
. . .   //access methods
}__attribute__((packed)); // sizeof(T1) =  7 bytes
```

The packed records without variant parts are similarly translated.

3.3.2. Data Layout Preservation for the records with variant part

From Example 3.3.2, one can see that the sizes of the type T9 in M-Pascal and in C++ are different. In this case *__attribute__((packed))* does not help us, since the fields of non-integral types in C++ are allocated in separate bytes (see DLP Section). One of the possible solutions is to move the UNION construction to the external level:

```
class T9 {
 union {                       /* bit offset  bit size */
  struct {
    boolean a:1;               /*     0            1  */
    boolean b:1;               /*     1            1  */
    unsigned char c:6;         /*     2            6  */
    unsigned char d:7;         /*     8            7  */
   }__attribute__((packed));
  struct {
    boolean :1;                /*     0            1  */
    boolean :1;                /*     1            1  */
    unsigned char:6;           /*     2            6  */
    unsigned char e:7;         /*     8            7  */
  }__attribute__((packed));
. . .   //access methods
}
};// sizeof(T9) =  2 bytes
```

Some remarks:

- *__attribute__((packed))* should be used exactly as shown;
- the number of unnamed fields in structures can be reduced by increasing their size, e.g., in the second structure three unnamed fields can be replaced by one *char:8*;
- there are no nested unions; all enclosed variant parts are transferred to the top level.

If in a target hardware the most significant bit has a number different from that in a source one, the fields contained in each byte should be enumerated in inverse order to preserve the bit order in a byte.

So, by using

- *__attribute__((packed))* when a record has a field whose alignment differs in M-Pascal and C++ (for example, field b in the type T1), or
- additional empty fields (char:n ($n < 8$) for byte alignment and char:8 for word alignment, see types T6 and T9),

it is possible to preserve data layout for almost all records except for ones considered in item 4 of Section 1 (see examples 1.7 and 1.8).

3.3.3. DLP for the records with fields of non-integral types

The translation scheme described does not preserve data layout for packed records similar to the type T7, i.e., for records containing a field of a packed record or array (of size less than 8 bits) allocated in the same byte with other fields (see Example 1.7.).

To translate such records, we shall treat the non-integral types of fields as integral types. For example, the bit field b in the type T7 is defined as follows:

```
class T7 {                        /* bit offset     bit size  */
    char a: 3;                    /* 0 bit           3 bit    */
    char b: 4;                    /* 3 bit           4 bit    */
       ...
};                       {sizeof(R) = 1 bytes}
```

All we have to do now is to implement some interface inside the class T7, which will allow us to access the field b as it were of packed array type. For this purpose, we shall use an auxiliary class BitRef, the same one we used to access an element of a packed array class (see Arrays). An object of the type BitRef is used to extract the value of the bit-object being referenced at the time of its (BitRef) construction. All the modifications are carried out over the temporary BitRef object which keeps a copy of the actual bit-object. After that, at the time of destruction, the actual bit-object is being updated by value from BitRef's copy. In this particular case, we will access the field of type $PackedArray < boolean, 1, 1, 3 >$ which starts in the first byte at the offset of 3 bits:

```
class T7 {                    /* bit offset       bit size */
    char a: 3;                /* 0 bit             3 bit   */
    char b: 4;                /* 3 bit             4 bit   */
public:
    BitRef<T0, 3> B() { return BitRef<T0, 3>((char*)this, 3); }
      ...
};                     {sizeof(R) = 1 bytes}
```

Now the class T7 preserves the data layout of the original record of the type T7. The method B() is used to access field b of type char, as it were of the packed array (T0). This technique may be either automated or performed manually, depending on the number of occurrences of these "bad" types in the source code.

3.3.4. Record initialization

As discussed in Section 2, the initialization of variables of class types corresponding to records is carried out through the constructor calls. For this purpose, in a C++ class two constructors are introduced:

- The default constructor for the case without an initializing list. It initializes all data by zero, just as in M-Pascal;
- The initializing constructor with a variable number of parameters. It receives the number of parameters known for each particular record and related to the fixed-part fields in the record. After that, it receives all the other parameters (through ellipsis ('...')) to initialize the fields of variant-parts of the record.

3.3.5. Records without variant part

M-Pascal allows initialization of a part of a record — an initial list can contain a number of values less than it is necessary to initialize all the elements in the record. For example, in the following initialization of a variable of type T1, only the fields a and b can be initialized:

```
VAR var_rec: T1= ['q', 10];
```

To properly handle initialization in the target C++ code, it is necessary to initialize the rest fields by their default values in the constructor definition:

```
class T1 {
. . .
T1(unsigned char a_=0, int b_=0, unsigned char c_=0);
};
T1 var_rec('q', 10);
```

3.3.6. Records with variant part

As the amount of fields and their types in variant parts can differ, the initializing constructor for variant part would receive the variable parameters number.

Let us consider the init constructor for the type T10:

```
 T10 (boolean variant_a, ...)
 {
         va_list Marker;
         va_start(Marker,variant_a);
         a(variant_a);      // tag value receiving
         switch (variant_a) { // according to the tag value the
 // number and the order of receiving of the last parameters is
 // determined in switch operator
         case true:
                b(va_arg(Marker, short ));
                c(va_arg(Marker, uchar ));
                switch (c()) {

                case '0': d(va_arg(Marker, boolean )); break;
                default:;
                } break;
         case false:
                f(va_arg(Marker, uchar )); break;
         default:;
         }
         va_end (Marker);
   }
```

Within this approach, the beginning of the parameter list relates to the fix-part fields and to the tag field. Further, all parameters are treated by C++ mechanism of working with the variable number of parameters. The use of this constructor requires that initial values for all elements of a record are to be specified explicitly in the constructor call. For this purpose, each incomplete initial list in the source code is extended by the default values of corresponding elements during translation to C++.

The approach described above, just as in the case of an array initialization, uses the mechanism of passing the variable number of parameters (va_arg macro), whose behavior is undefined when passing objects of non POD-types, according to the C++ language standard. In the general case, the record can contain any number of fields of such types. In order to eliminate the dependence on a C++ compiler implementation, the scheme of translation can be slightly changed. It is proposed to pass not the initializing objects themselves but the pointers to them. As the pointer type is the POD-type and has a fixed size (4 byte), such changes can resolve the problem with macro `va_arg`. In this case, the constructor call looks like follows:

```
#define INIT(type, value) auto_ptr<type>(new_type(value)).get()

T10 var_rec =
T10(true, INIT(short, 1), INIT(char, 0), INIT(boolean, true));
```

4. Nested functions and procedures

In contrast to C++, M-Pascal allows us to declare nested functions and procedures. Though some C++ compilers permit nested functions (GCC extension), it is really hasty to rely on extensions of "clever" implementations. For simplicity, only nested functions will be considered (all further discourses are correct for procedures as a special case of functions). Let us consider the following example:

```
FUNCTION outer : INTEGER;
    TYPE t = INTEGER;
    VAR a, b : t;
    FUNCTION inner : t;
    BEGIN           {inner}
        inner := a+b;
    ENDFUNC;            {end inner}

BEGIN               {outer}
    a := 1;     b := 2;
    outer := inner;
ENDFUNC;                {end outer}
```

Three translation schemes of conversion are possible:

- de-nesting — moving the nested functions to the global level;
- usage of C++ `namespaces` for preserving the nested structure;
- simulation of nested functions by nested data structures.

In this paper, we will consider only the first schema, the others are described in [1].

4.1. De-nesting functions

All nested functions are moved to the global level (file scope), as required by the C++ standard. All the variables used in nested functions and not declared at the file-scope are passed to these functions by reference as additional input parameters.

```
typedef short t;
t inner(t &a, t &b)  {  return a + b; }
short outer() {
    t a, b;
    a = 1; b = 2;
    return inner(a,b);
}
```

This translation scheme, apparently, is the most obvious conversion resulting in a desirable functionality. A little less obvious are the consequences of such a conversion:

- The type declared in the scope of an upper level function requires moving it to the global level, if it is used at least in one nested function or it is used as a template parameter in some type declarations, e.g., the type of an array element (see Arrays). This can easily result in a name conflict at the global level, if two or more nested types were of the same name and were both brought to the global level. Conflict is avoided by the use of automatic unique names for types moved or by manual modification of the source Pascal code. The latter appears to be more acceptable with the restrictions on readability of the target C++ code.
- Nested constants moved out, as in the case of types, can result in the name conflict, which is avoided by the same methods as above. An alternative would be passing the constants as additional parameters of constant reference type (as compared to variable passing).

5. Conditional compilation

5.1. Translation scheme

M-Pascal provides several language features not to be found in the standard Pascal. One of them, which is required to be kept at translation, is conditional compilation. To arrange Conditional Compilation, M-Pascal has directives `%if` and `%end`:

```
{%if boolean_expression} any text {%end}
```

Any program code and other directives may be enclosed within these two directives. Let us call the pair of conditional compilation directives and a program code between them 'Conditional Compilation Clauses' (CCC).

In addition, there is a set directive {%set identifier: = expression} which is used to assign a constant value to a compile-time identifier. The difference between M-Pascal conditional compilation and C/C++ one is in the following:

- there is no preprocessor in the compiler of M-Pascal — the compiler calculates a boolean expression and compiles or not the code inside CCC;
- as a result, in expressions of CCC, both the CCC-variables and program constants (including constants of enumerated types) can be used. For example:

```
const a=1;
{\%if a}             {a is a program constant}
{%set b: = 10}            {b is a CCC-variable}
{%set a: = true}      {a is a new CCC-variable }
  ...
{%end}
```

The M-Pascal conditional compilation directives are translated directly into the C ++ ones. The `%set` directive is translated into `#define` directive:

```
{%set debug: = TRUE} -> #define debug_ccc TRUE
```

The postfix '_ccc' will be used to distinguish program constants from CCC-variables, since the preprocessor, having executed `#define` directives, would otherwise replace the program constants. The directives `%if` and `%end` are translated accordingly into `#if` and `#endif` directives:

```
 #if C++_boolean_expression
       any text
 #endif
```

5.2. Translated and not translated CCC

Directives of the conditional compilation are external constructions of the source language but not members of the input grammar. However, for preservation of CCCs in the target code, it is necessary to parse all CCCs and to build their internal representation. The extension of the source language grammar considering an arbitrary arrangement of CCCs is impossible — inside CCCs, there can be enclosed not only a complete syntactical construction (derived from a non-terminal), but also any sequence of terminals. It is possible to extend grammar if we agree to restrict the form of constructions enclosed in CCCs, but in our case the source code would need a lot of changes.

In our approach, we slightly restrict the use of CCCs, but the grammar is left unchanged. All CCCs are divided into two types: translated and not translated. One can manually change tt not translated CCCs so that they become `translated`.

Definition 5.2.1. Let us call a *simple CCC* the one of the kind

```
{% if boolean_expression} any text {%end} (1)
program fragment without CCC.
```

Definition 5.2.2. In CCCs of the kind

```
{% if boolean_expression1} any_text1 {%end}         (2)
{% if boolean_expression2} any_text2 {%end}
program fragment without CCC;
```

`any_text1` and `any_text2` are *alternative* to each other if:

– they are derived from the same non-terminal or grammatical sequence in the same grammatical construction;

– or `any_text1` and `any_text2` are declarations and they declare the same identifier.

Definition 5.2.3. Two CCCs of kind 2, where `any_text1` and `any_text2` are alternative, are called *CCCs with alternative* (CCCsA). Two CCCs, in which `any_text1` and `any_text2` are not alternative, are called *successive simple CCCs.*

Example 5.2.1.

```
{%IF MODE = DEBUG}
  table_infor: ARRAY [debug_files] OF select_record;
{%END}
```

```
{%IF MODE = RELEASE}
  table_infor: ARRAY [release_files] OF select_record;
{%END}
```

These CCCs are CCCsA, because they declare the same identifier — `table_infor`.

Example 5.2.2.

```
table_infor:
{%IF MODE = DEBUG}              {1st CCC}
ARRAY [debug_files] OF select_record;
{%END}
{%IF MODE = RELEASE}            {2nd CCC}
ARRAY [debug_files] OF select_record;
{%END}
```

These CCCs are CCCsA, because they are derived from the same non-terminal — `type-denoter` in the same grammatical construction — `variable-declaration` (see below).

Example 5.2.3.

```
{%IF MODE = DEBUG}
    wait_time: EXTERNAL DEFINE relative_time: = [0,0,6,0];
{%END}
{%IF MODE = MAIN}
    help_pid: process_id;
{%END}
```

These CCCs are two successive simple CCCs, because they declare different identifiers.

The main problem is to recognize if `any_text1` and `any_text2` in (2) are alternative to each other or not. It is not a problem to translate the CCCs similar to Example 5.2.1 and Example 5.2.3. Example 5.2.3 is "`bad`" because parser does not know if the second CCC is alternative to the first one or it is a new CCC declaration (successive CCCs). It may be solved for some cases but in the general case (especially in expressions) it is impossible (grammatical ambiguity).

Let us consider the following grammatical rules:

```
variable-declaration ::=
   identifier-list":" type-denoter [(": = " initial-value-list)].
identifier-list ::= identifier {"," identifier}.
```

```
initial-value-list ::=
constant |"[" initial-value{"," initial-value}"]"]
initial-value ::= expression.
index-expression ::= expression.

conditional-statement ::= if-statement | case-statement.
if-statement ::=
       "IF" Boolean-expression "THEN" statement
        {"ORIF" Boolean-expression "THEN" statement}
        [else-part].
 else-part ::= "ELSE" statement.
case-statement ::=
       "CASE" case-index "OF"
            case-list-element {";" case-list-element}
            [[";"] "OTHERWISE" [":"] statement-sequence] [";"]
          "END".
```

In each rule all non-terminals and grammar sequences can be divided into `iterative` and `non-iterative`. Iterative non-terminals (grammatical sequences) are enclosed in curly braces ({,}).

Examples of iterative non-terminals (grammar sequences) are `identifier (in identifier-list), "ORIF" Boolean-expression "THEN" statement (grammar sequence), case-list-element)`.

Assertion

- If two or more Conditional Compilation Clauses are alternative and contain program fragments derived from iterative non-terminals (grammatical sequences), then they can be translated into C++ CCCs successfully. (*)
- Any simple CCC can be translated into C++ CCC successfully. (**)
- CCCs meeting the syntactic property (*) or property (**), and only them, are valid for translation.

Example 5.2.4. The CCCs with alternative (CCSsA)

```
{%if cond_expr1} IF expression1 THEN {%END}
{%if cond_expr2} IF expression2 THEN {%END}
       statement-sequence
 ENDIF;
```

do not meet property (**) by its definition and (*) because they contain program fragments derived from non-iterative grammatical sequences.

Example 5.2.5. The CCCs with alternative

```
{%if DEBUG}
     [mls_key, '],
{%end}
{%if not DEBUG}
      [mls_key],
{%end}
```

meet properties (*), because they contain program fragments derived from iterative non-terminal (`initial-value`).

5.3. CCC implementation

In the approach proposed, the initial M-Pascal grammar is not being extended to provide the analysis of the source M-Pascal code with arbitrary insertion of CCCs. Instead, to preserve the layout of the source code, conditional compilation directives are treated as comments and attached to definite lexemes of the source program.

While an ordinary comment does not need any parsing to be performed on it, at the time of analysis of a CCC-comment an additional parser run. This parser creates an internal form of an expression contained in the CCC. The resulting parse tree of the expression is then being attached to the definite lexeme, and after that — to the object of an Intermediate Representation (IR) of a source program. Later, at the code generation stage, for every object in the IR, the code-generator checks if there were any comments or CCC-comments attached to the object and output them properly into the target code.

The other task arising during construction of the IR of the source program is to set up a correct correspondence between named objects in the IR (variable, constants etc.).

Let us consider the example:

```
{%if  expr1}  var A: integer; {1} {%end}
{%if  expr2}  var A: longint; {2} {%end}
......
proc (A);  // any routine call
```

During construction of IR for this source program, the following should be emphasized:

- Two named objects (variables) with the same name "A" are being created. Having the same name, they differ in their types, as a matter of fact. Each of the two objects is represented by a node IR — "Variable A declaration node".

- To preserve the semantics of the routine's input parameters, the only one "A" should be considered out of two existing "A"s. For example, in the call of `proc`, it is necessary to know the exact type of "A" to generate an appropriate typecasting operator in the target C++ code.

The following technical solutions are proposed to achieve the proper construction of IR and successful code generation:

- A so-called global CCC-stack is introduced. A pointer to the parse tree of a conditional expression is pushed onto the stack when encountering an if-directive. The pointer is being popped out as soon as an end-directive is encountered.
- During construction of a semantic (named) object corresponding to some named object of the source program (variable, constant etc., not IF node), the current state of the CCC-stack is being copied to a special storage inside of this named object. Thus, for every named object, we obtain a set of conditions (CCC-Set) under which this object was declared. In the above example, two objects of the variable A are created. The CCC-Set of the first object contains `expr1`, and of the second one — `expr2`.
- All the objects with the same name (A in our example) declared on the same scope, are collected into one list. A scope descriptor contains a pointer to the head of this list. Now, to set a correspondence between a name and an existing object in the current scope, the converter searches for the pointer to the list of objects with the name specified and then, if the list is found, searches for the object with a CCC-Set corresponding to the set of conditions in the global CCC-stack (its current state).

It is theoretically possible that an object has several declarations under different CCC directives, but is used outside of any CCC statement (an empty CCC-stack). In this case, the converter issues a warning and chooses the first found object. In our example, if there are no CCC directives around the call statement of the procedure `proc`, the converter produces a warning on this line, because it cannot find any object with the name A and empty CCC-Set. However, it will continue assuming that nothing depends on the type of A. In the general case, this may lead to erroneous (under some conditions set) output code. It should be recommended that a named object be used under the same set of conditions as it is declared. In our example, it must be something like the following:

```
{%if  expr1}  var A: integer; {1} {%end}
{%if  expr2}  var A: longint; {2} {%end}
......
{%if expr1}  proc (A);  {%end}  // routine call with object A-1
{%if expr2}  proc (A);  {%end}  // routine call with object A-2
```

The approach described allows us to realize all the theoretical issues of CCC translation and to translate the majority of the source M-Pascal code containing CCC directives.

6. Concluding remarks

We have described some translation schemes from extended Pascal to C++, which are of the most interest, in our opinion. The schemes satisfy the requirements, which are essential for preservation of functionality of a large software-hardware system after replacing a program language and/or a part of hardware. We tried to show our approaches to the development of these schemes and we hope that this may be useful for other developers. More details and other translation schemes may be found in [1].

References

[1] Markin V.A., Maslov S.V., Novikov R.M., Sulimov A.A. Extended Pascal to C++ converter. — Novosibirsk, 2001. — (Prep. / SB RAS. IIS; N 92).

[2] Aho A.V., Ullman J.D. The Theory of Parsing. Translation and Compiling Vol.1: Parsing. — N.J.: Prentice-Hall, Englewood Cliffs, 1972.

[3] Aho A.V., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools. — Addison-Wesley: Reading, Mass, 1988.

[4] Kasyanov V.N., Pottosin I.V. Methods of Compiler Development. — Novosibirsk: Nauka, 1986.

[5] GNU Pascal To C converter
`http://www.ibiblio.org/pub/Linux/devel/lang/pascal/ptoc_3.34.tar.gz.`

[6] FreeBSD Pascal To C converter
`ftp://ftp.freebsd.org/pub/FreeBSD/ports/i386/packages-4-stable/lang/ptoc-3.50.tgz.`

[7] MSC Pascal To C/C++ converter
`http://lib.buryatia.ru/cgi-bin/win/SOFTWARE/ptoc.README.txt.`

[8] Jensen K., Wirth N. Pascal. User Manual and Reports. — Springer-Verlag, 1978.

[9] *Program language C++ standard (ANSI/ISO/IEC 14882)* `http://www.techstreet.com/cgi-bin/detail?product_id=49964`.