

**Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems**

V. A. Markin, S. V. Maslov, R. M. Novikov, A. A. Sulimov

EXTENDED PASCAL TO C++ CONVERTER

**Preprint
92**

Novosibirsk 2002

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

**Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова**

В.А. Маркин, С.В. Маслов, Р.М. Новиков, А. А. Сулимов

КОНВЕРТОР С РАСШИРЕННОГО ПАСКАЛЯ В C++

**Препринт
92**

Новосибирск 2002

В работе описаны основные схемы трансляции, разработанные для конвертора с существенного расширения языка Паскаль в C++. Конвертор был написан по заказу крупной телекоммуникационной корпорации и должен был не только транслировать исходный M-Pascal код в функционально эквивалентный C++ код, но и отвечать ряду требований. Совокупность расширений Паскаля и требований к конвертору значительно повлияла на разработку схем трансляции.

INTRODUCTION

The methods of compiler (and converter as a special case) development are sufficiently investigated and described [1–3]. There is a number of converters [4–6] from the standard Pascal language [7] to C/C++ languages [8].

In this work, some translation schemes (having the 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 an input M-Pascal code to the functionally equivalent C++ code, but also meet some requirements. M-Pascal language extensions and requirements for the converter have essentially influenced on 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.
- UNIV type in a formal parameter list. This causes type checking to be suppressed; that is, both the calling and called routines must declare a type for each parameter, but the types do not have to be compatible.
- The type STRING [n] is a special case of PACKED ARRAY [1..n] OF char. M-Pascal contains predefined functions and procedures for string manipulation and a special notation for denoting sub-strings.
- Statements or declarations can be copied from a library by using the *include* option.
- Constant expressions (e.g., 5+3) can be used wherever constants are allowed.
- The strict ordering of definitions in the standard Pascal has been weakened.
- Array and record types that do not contain files can be compared for equality and inequality.
- 'Integers' are represented by two different predefined types, *integer* and *longint*.
- Dynamic arrays allow the specification of array bounds at run-time.

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 chapter 1, a conception of Data Layout Preservation (DLP) is presented. Chapter 2 describes the general approaches to initialization. Chapter 3 is overview of the translation schemes of the most interesting aggregate M-Pascal types: records (without and with variant part), arrays and sets. In chapter 4, the translation schemes of nested procedures in M-Pascal are given. In chapter 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 to 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 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 a 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 ensuring 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 one.

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

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.

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, 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
```

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.

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 doesn't 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
{built-in data type}
A: integer := 1;

{record with variant part}
B: RECORD
    ID: (one, two, three);
    CASE Boolean OF
    True: (name: STRING[10]);
    False: ();
```

```

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

{array}
C: ARRAY [1..5] OF char := ['a','b','c'];

{set }
D: PACKED SET OF 1..10 := [1..3, 5];

```

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

C++

There are two major possibilities of object 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 the class T may be initialized using the list of initializers, if only:

- a) T does not define constructors;
- b) all members of T are public;
- c) T does not have base classes;
- d) T does not have virtual functions.

Therefore, only the classic C structures can be initialized by the list of initializers. Moreover, according to the standard of C++ [8], only the global scope objects are initialized by zeros, while the nested scope objects' initial value 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 demands to preserve Data Layout, semantics and appearance of code forces 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 requirement b) 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 requirement a) 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 the initializer-list

```

Since we are initializing the C++ classes (which correspond to the M-Pascal records) by constructors, all the fields of these classes should have a constructor (at least, a default one). That is, for every structured type in M-Pascal, a C++ class is being 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 a 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 Initialization chapter as well):

- there is no Packed Arrays in C++;
- there is a requirement to preserve the data layout in arrays (see DLP chapter);
- 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 of M-Pascal arrays.

The template C++ class, which models the 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 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).

Packed arrays

An array with the element size more than 8 bits is not being 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 auxiliary BitRef class has been 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. Access to elements of packed arrays is implemented through the interface of this class.

Array initialization

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

- a constructor with the 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 the 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 being passed through ellipsis ('...'). That is, such 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 the array elements. Once the temporary object is filled with the initializers, it is being 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 DLP chapter) 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 of 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
};
```

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 really used. The low and high bound checking is carried out dynamically using the first two template parameters (`LOW` and `HIGH`).

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 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.

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 con-

structured 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 using the UNION construction for variant parts.

Example 3.3.1 (see Examples 1.5). The 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. The 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.

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++) – 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 an unnamed field to preserve data layout:

```
class T1 {
    unsigned char a;      /* byte offset byte size */
    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 almost similarly translated. The type T6 from Example 1.6 can be translated into C++ in the following manner:

```
class T6{short a:3;short b:3;char:2; /*char:2 - unnamed bit-field
of 2 bits.*/
    short c:3;short d:3;char:2;
    short e:3;
    . . .                //access methods
}__attribute__((packed)); //19 bits
```

The size of T6 without `__attribute__((packed))` is 4 bytes.

Data Layout Preservation for 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 chapter). One of possible solutions is to move the UNION construction to the external level:

```
class T9 {
    union {
        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.

For example:

```
T10=packed record
  CASE a: boolean OF
    TRUE: (b: integer;
  case c: 0..1 of
    0: (d:boolean);
    1: (e:char););
  FALSE: (f: char);
end;
```

T10 is translated into the following:

```
class T10 {
  union {
    struct {
      boolean a__;
      char :8; //additional field for word alignment
      short b__;
      unsigned char c__;
      boolean d__;
    };
    struct {
      char Filler_[5]; //five bytes
      unsigned char e__;
    };
    struct {
      char :8;
      unsigned char f__;
    };
  };
  . . . //access methods
};
```

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. For example, the type T6 will have the following form:

```
class T6{ char:2; short b:3; short a:3; //first byte
         char:2; short d:3; short c:3; //second byte
         char:5; short e:3;           //third byte
         . . . //access methods
}__attribute__((packed));
```

So, by using

- `__attribute__((packed))` when a record has a field whose alignment differs in M-Pascal and C++ (for example, field b in type T1);
- 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 those considered in item 4 of chapter 1 (see examples 1.7 and 1.8).

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 {
    char a: 3;
    char b: 4;
    ...
};
/* bit offset  bit size */
/* 0 bit      3 bit */
/* 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 the value from the BitRef's copy. In this particular case we shall access the field of the type PackedArray<boolean, 1, 1, 3> which starts in the first byte at the offset of 3 bits:

```

class T7 {
    char a: 3;
    char b: 3;
public:
    BitRef<T0, 3> B() { return BitRef<T0, 3>((char*)this, 3); }
    ...
};

```

The method B() is used to access the field b of type char, as it were of the packed array type (T0). With the translation scheme proposed, the access to the field b in M-Pascal

```

var rec : T7;
. . .
rec.b:=true;

```

will be translated into:

```

T7 rec;
. . .
rec.B().access()[1] = true;

```

Note that the access() method of the BitRef class is used to extract an object of the type T (PackedArray<boolean,1, 1, 3> in this particular case) from the object of the type BitRef<T, W>. Now the class T7 preserves the data layout of the original record of the type T7. This technique may be either automated or performed manually, depending on the number of occurrences of these bad types in the source code.

Record initialization

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

- 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.

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 the 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 {
    unsigned char a;
    int b;
    unsigned char c;
    . . . //access methods
    T1(){memset(this, 0, sizeof(T1));}
    T1(unsigned char a_=0, int b_=0, unsigned char c_=0)
        {a=a_; b=b_; c=c_;}
};
```

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 number of parameters.

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 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 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 the fixed size (4 byte), such changes can resolve the problem with macro `va_arg`. In general terms, this new approach complicates implementation a bit and increases the cost of initialization:

```

t10 (boolean variant_a, ...)
{
    va_list Marker;
    va_start(Marker, variant_a);
    a(variant_a);
    switch (variant_a) {
    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);
}

```

The constructor call looks like the following:

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

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

3.4. Bound

The BOUND attribute in M-Pascal indicates that the variable will be associated with another allocated variable or memory reference at run-time through the use of the built-in BIND procedure. No storage is directly associated with the BOUND variable (except a pointer to the variable to which it is bound). Only local variables of procedures or functions can be declared with the BOUND attribute. After the bind procedure has been executed, any references to the BOUND variable will reference the store occupied by the variable it was bound to. This variable may be of any type.

The actual BIND-ing may appear anywhere in the procedure body and is valid until execution reaches the next BIND or returns from the procedure.

Thus, a BOUND variable can be translated only to a pointer rather than to C++ reference (since a reference may be initialized only once). The following translation scheme is proposed:

```
M-Pascal: bvar: BOUND type;
C++:      type *bvar;
```

Further, each call

```
BIND(bvar, ref);
```

is translated into

```
bvar = (type*) &ref;
```

Example.

```
TYPE SomeType = RECORD f: INTEGER END;
VAR  bvar: BOUND SomeType;
BEGIN
    BIND(bvar, recl);
    WITH bvar DO
    SomeProc(f);
END;
```

is translated to:

```
class SomeType {
    short f;
};
SomeType *bvar;
```

```
bvar = (SomeType*) &recl;
someproc( (*bvar).f );
```

3.5. For-statement

The for-statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable called the control-variable of the for-statement. It is an error to have an assigning-reference to the control variable within the repeated statement. Evaluation of the lower and upper bounds of a loop is performed once before the execution of a loop body in M-Pascal. Let us consider the following example:

```
VAR I, J: INTEGER;
J:=10;
FOR I:=1 TO J DO
BEGIN
  J:=J+1;
  WRITELN(I, ' ', j);
END;
```

This loop will stop after 10 iterations. To the contrary, in the C++ for-statement, the for-condition (*for-cond_{opt}*) and expression (*expr_{opt}*) are performed every time after *stmt* is performed.

```
C++:
for (init-stmtopt; for-condopt; expropt) stmt
```

This means that, in the case of “straight” translation of M-Pascal for-statement to C++, the above loop will become infinite after compilation by a C++ compiler.

To preserve the source M-Pascal semantics, the following translation scheme is proposed:

M-Pascal:

```
VAR var:T;
FOR var := init-val TO final-val DO stmt
```

C++:

```
T var;
var = init_val;
T last_var;
for (last_var = final-val; var <= last_var ; var++) {
  stmt;
}
```

For implementation of the DOWNTO for-cycle, the loop variable increment is replaced by decrement.

The above scheme does not work in the cases when the loop-variable runs the values from the lower to upper value of a type. Let us consider the example:

M-Pascal:

```
type fc_range = 0 .. 255;
var i : fc_range;
for i := lower(fc_range) to upper(fc_range) begin
  // functions lower and upper returns correspondingly the low (0)
  and upper (255) value of the type fc_range
  //...
end;
```

C++:

```
typedef char fc_range;
const fc_range LOW_fc_range = 0;
const fc_range HIGH_fc_range = 255;
fc_range i = LOW_fc_range;

fc_range last_var;
for (last_var= HIGH_fc_range; i <= last_var; ++i) {
  //...
}
```

In this example, the loop translated will become infinite, because semantics of the for-statements in M-Pascal and C++ is different. When the value of the loop-variable *i* becomes 255, the loop body executes, but after the succeeding increment of the variable *i* its value becomes 0, which satisfies the for-condition. Such a situation occurs, for example, when the DOWNTO loop-variable is of enumeration type and the loop lower bound is 'LOWER (enumeration_type)'. The condition loop_variable >=0 will never become false.

This leads to the following translation scheme in C++:

```
T last_var;
var = init-val;
for (last_var = final-val; ; var++) {
  stmt;
  if (var == last_var) break;
}
```

3.6. UNIV formal parameters

The UNIV keyword is an utterly specific feature of M-Pascal. The following is a citation from the language guide: "The prefix UNIV in a parameter-group suppresses compatibility checking of the actual and formal parameters. UNIV cannot be used when either the actual-parameter or formal-parameter but not both is declared as having the STRING type. The compiler issues a warning mes-

sage whenever the size of the actual UNIV parameter is less than the size of the formal UNIV parameter.”

The following example will be considered below:

```
VAR a: AT;
PROCEDURE p (UNIV a: FT);
BEGIN ... END;
...
p(a);
```

It is obvious that an explicit casting C++ mechanism should be used to reproduce the behavior of the M-Pascal compiler, since C++ itself does not provide this kind of suppressing, being an extremely type-safe language.

There is a set of casting operators available: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, and C-like type casting. The above example could have been translated into:

```
AT a;
void p (FT a) {}
...
p(*reinterpret_cast<FT*>(&a));
```

Here, the actual parameter `a` of the type `AT` is being explicitly cast to the type `FT` before passing it to the procedure `p`. Unwise extraction of a value of the type `FT` from a value of the type `AT` is taking place, and this extracted value is being passed to the procedure. This scheme has the right to exist with the assumption that none of the external modules (from the outside of the set of files being converted into C++) call this procedure. Otherwise, the following properties of the UNIV-mechanism should be considered to propose a proper translation scheme:

- UNIV parameters in M-Pascal are always passed by address.
- If the actual parameter is an expression then the value is copied to a temporary and the address of the temporary is passed.

The following algorithm, therefore, needs to be implemented:

1. Get an address (memory location) of an actual parameter of the type `AT`.
2. Pass the address obtained to the procedure.

In the procedure:

3. Accept the address of the UNIV parameter.
4. Dereference the address, treating it as a pointer to the value of the type `FT`.

Different implementations of the algorithm are possible. The one below is rather interesting:

```

template <class FT = int>    // FT - formal type of a parameter
class UNIV {                // Auxiliary class UNIV (define once)
    const void *value;
public:
    /**) Constructor to convert from UNIV<T1> to UNIV<T2>
    template <class T> UNIV(const UNIV<T> &a) { memcpy(this, &a,
4); }

    /**) Constructor that gets an address of an actual parameter
    template <class AT> UNIV(const AT &a)    { value = &a; }

    /**) Type cast operator, used to cast from UNIV<T> to T type
    operator FT& () { return *(FT*)value; }
};
AT a;
void p (UNIV<FT> _a) // Casting from UNIV<> to UNIV<FT> (*) is
performed
{
    FT a = _a; // Dereferences the value member (***)
    ...
}
...
p(UNIV<>(a)); // Initializes through (**) the value member of
UNIV<>() with an address of a

```

Besides the comments in the code, there are the following important notes:

- Semantics of passing UNIV-parameters is preserved here.
- A temporary variable `_a` is used to accept the address of the actual parameter. Then it is being dereferenced, initializing a local variable `a`. In the case, when it is a VAR UNIV parameter, this initialization will be converted to

```

...
void p (UNIV<FT> _a)
{
    FT &a = _a; //reference sign added
    ...
}
...

```

To finalize the discussion, the following caution is quite appropriate: UNIV parameters are unsafe and often unnecessary. Of course, they are left for backward compatibility. However, we would not recommend using this technique in the subsequent development based on the converted code.

3.7. Strings and Sub-Strings

One of the differences between M-Pascal and Standard Pascal is a double notation of the type String. M-Pascal holds two concepts of a string: the standard Pascal string which is of the type `PACKED ARRAY[1..n] OF char` and the M-Pascal string which is of the type `STRING[n]`. The correspondence of a character-string with a string-type is set by relating the individual characters of the character-string to the components of the string type.

A variable of the type `STRING[n]` has all the properties of a variable of the string-type `PACKED ARRAY[1..n] OF char`. In addition,

- it can be assigned to other variables of the predefined types `STRING`, `char`, or `PACKED ARRAY[1..m] OF char`, even if the declared lengths differ ($m < n$);
- it can be compared with other variables of the predefined types `STRING`, `char`, or `PACKED ARRAY[1..m] OF char`, even if the declared lengths differ ($m < n$);
- sub-strings can be extracted from it; strings can be modified by assignment into sub-strings;

Example

```
VAR a,b: STRING[10];
...
a := 'hello'; { assigns string from char }
b := a;      { assigns string from string }
b[1,3] := 'Hello'; {extracts 3 symbols starting from the first
one
                in b }{ assigns into sub-string of b}
writeln(b);  { would output: "Hello" }
```

- it can be passed as a variable parameter to a procedure or function even if the formal parameter has a different declared length. In this case, the size of the formal parameter is ignored; instead, the actual length of the string being passed is used within the called routine;
- it can be passed as a value-parameter in a call, even if the formal parameter has a different declared length. The actual parameter is assigned to the formal parameter, so padding or truncation can occur if the actual and formal parameters have different lengths. The length of the string in the called routine is always the declared length of the formal parameter.

For simulation of the above-indicated properties, the class `MString` is used.

```

template <int N>
class MString: public Array<char, 1, N> {
public:
    // Construct MString from literals and other MStrings
    MString();
    MString (const char &ch);
    MString (const char *str);
    template <int M>
    MString(const MString<M> &other);
    MString(const MSubString &other); /*MsubString class will be
                                     described below */

    // type casting operators
    DynArray<char>&(); // (dynamic array) by reference
    operator MSubString (); // MString parameter by reference

    // substrings
    MSubString operator () (long index, long sub_length);

    // assignment
    template<int M>
    MString& operator =(const MString<M> &other); // any string
    MString& operator =(const MSubString &other); // any substring
    MString& operator =(const char *str); // any literal
    MString& operator =(const char ch); // any char

    // aggregate comparison
    template<int M> bool operator ==(const MString<M> &other);
    template<int M> bool operator !=(const MString<M> &other);
    ...

    // Pascal functions for arrays
    const size_t ArraySize();
    const long LowerIndex() const;
    const long UpperIndex() const;
    const size_t StrSize() const;

    // MString StrScan
    long StrScan(const MSubString &pattern);
};

```

The given class is inherited from the base type *Array* (see Chapter *Array*), which allows applying to the object of the given type all of the array operations: element access, array size, and array comparison. Besides, the given template class allows us to preserve the semantics of the source program and its external representation, which would be impossible with the use of the standard C strings (array of char). The *MString* class provides a rich interface to handle operations with strings.

As it was emphasized at the beginning of the chapter, there is an additional M-Pascal mechanism of extracting a sub-string from a string. A sub-string of a string object can be considered as a separate object of the string type inherited in the parent string but limited by the specified range. It can be clearly demonstrated by the example of passing a string to a procedure/function by reference (as a VAR parameter). For such string parameters the size of the string and the address of the string are passed. So the string size of the formal parameter is ignored and the actual length of the string is considered inside the routine.

To preserve this string type semantics, a new class MSubString has been introduced.

```
class MSubString: public DynArray<char> {
public:
    // Construct MSubString from literals
    MSubString();
    MSubString (const char &ch);
    MSubString (const char *str);

    // Construct MSubString by binding it to an existing string
    MSubString(const MSubString &other);
    MSubString(const char *address, long from, long length);

    // substrings
    MSubString operator()(long index, long sub_length);

    // assignments
    template<int M>
    MSubString& operator =(const MString<M> &other); // any string
    MSubString& operator =(const MSubString &other); /* any sub
                                                    string*/
    MSubString& operator =(const char *str) // any literal
    MSubString& operator =(const char ch) // any char
    // aggregate comparison
    template<int M> bool operator ==(const MSubString<M> &other);
    template<int M> bool operator !=(const MSubString<M> &other);
    ...
    // Concatenation
    MSubString operator +(const MSubString &other); // any string
    MSubString operator +(const char *s); // any literal
    MSubString operator +(const char &ch); // any character
    // Pascal functions
    const size_t StrSize() const;
    long StrScan(const MSubString &patTern);
};
```

The class MSubString denotes the type of a string/sub-string, whose actual size is not possible to be calculated at compile-time. For example, if a string is passed into a routine as a VAR parameter, then it is being cast to the MSub-

String type to provide a safe mechanism of manipulation with the string at run-time. MSubString also denotes the type of a constant of a string type, since its size is only known at run-time when concatenation is performed (in M-Pascal, concatenation is performed at compile-time). The cast operator DynArray<char>() in the class MString is used to cast a parameter when passing an actual parameter of the STRING[N] type and accepting a formal parameter of the dynamic array type. The StrSize method is properly defined to return a run-time length of the string.

MSubString contains a pointer to the string storage, instead of the storage itself. Therefore, the size of the object of the type MSubString is equal to four bytes, which preserves the data layout in parameter passing. For that purposes the base class DynArray has the only member bounds of the pointer type to the following structure:

```
template < class T>
  struct Dynvar {
    T* data; /*In the class MSubString it points to the beginning
             of the string*/
    int lower_bound, upper_bound; //The bounds of the dynamic array
    bool preserve_data; /*serves to indicate that this dynarray
                        is casted from the static array */
  };

template< class T >
class DynArray
{
protected:
    Dynvar<T>* bounds;
    . . . // class DynArray implementation
}
```

Such an approach fully simulates the M-Pascal string-type parameter passing and keeps the semantics of working with sub-strings in M-Pascal.

Let us consider the example of MSubString usage:

```
type
  any_string = string[32767];

procedure procla(VAR s : any_string);
var
  x : longint; p : ^any_string; len : integer;
begin
  p := ptr(s); len := strsize(s);
  x := ptrdiff( ptr(s), nil );
  writeln('procla: p strings is ', p^[1, len]);
endproc;
```

```

var
  p_string : string[20] := 'abcd -- test string.';
begin
  procla(p_string);
endprog.

```

C++ code looks like this:

```

typedef MString<32767> any_string;
static void procla(MSubString s){ /* MSubString is used instead
                                   of VAR String-type parameter*/
    int x = 0;
    any_string *p = NULL;
    short len = 0;

    p = s.Ptr<any_string>();
    len = s.StrSize();
    cwrite << ("procla: p strings is ") << ((*p)(1,len)) << endl;
//extracting a sub-string
}

MString<20> p_string = "abcd -- test string.";
int main(void) {
    procla(p_string);
return 1;
}

```

4. NESTED FUNCTIONS AND PROCEDURES

In contrast to the C++ language, 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 := a+b;
  ENDFUNC;

BEGIN

```

```

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.

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 references 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).
- The syntax of a nested function call changes. The additional parameter list is required, which is not a problem for the converter but becomes a problem for future maintenance of the C++ code.

Usage of C++ namespaces for the nested structure preservation

The structure of nesting is described in terms of C++ namespaces, which can be nested in turn. There is a namespace defined for each function, which automatically includes the namespace of the parent function. At the beginning of the namespace description, every function is declared using its own namespace with the help of using-directives.

```

short outer();
namespace namespace_outer {
    typedef short t;
    t a,b;
    t inner();
    namespace namespace_inner {
    }
}

namespace namespace_outer::t namespace_outer::inner()
{
    using namespace namespace_inner;
    return a+b;
}

short outer()
{
    using namespace namespace_outer;
    a = 1;
    b = 2;
    return inner();
}

```

Regardless of the fact that namespaces allow us:

- to preserve the logical structure of the source code;
- with the name encapsulation, to avoid any name conflict, which is almost inevitably originating from de-nesting;
- to achieve readability and maintainability of the target program.

Such an approach appears to be invalid as soon as multithreading is considered. In this case two or more threads can concurrently use the same data objects from the same namespace, as a matter of fact, destroying the data of competing threads. Since it is not known beforehand that only one control thread exists, the translation scheme using namespacing can be expanded to the scheme that uses classes instead of namespaces. It becomes possible to create the own set of objects of such classes for each control thread.

Simulation of nested function structure by nested data structure

Every function is represented by a class, containing:

- nested type declarations;
- private members corresponding to the function parameters (they are initialized in a constructor);
- private members corresponding to the function local data (variables and constants), the constants can be declared statically with the purpose of optimization;
- a reference to the parent function-class (only in nested function-classes) initialized in a constructor;
- nested class declarations corresponding to the nested functions in the source code;

and also:

- a public constructor accepting the list of function arguments and (for nested functions only) a reference to the parent function-class;
- the single public method (e.g., an overloaded operator ()) containing implementation of the function body.

For the best readability of the target C++ code and more exact source semantics, for each nested class it is possible to create a private wrapper-method in parent class engaged only in correct initialization of the object of the nested function and call of the “execute” method (the operator () in our example).

```
class outer {  
  
    typedef short t; // local type  
    t a, b;          // local variables  
  
    class inner {  
        outer& _outer;  
    public:  
  
        t operator () ();
```

```

        inner (outer& outer_): _outer(outer_) {}
};

friend class inner;
t inner () { return class inner(*this)();}

public:
    short operator () ();
    outer() {}
};

short outer () { return class outer()(); }

outer::t outer::inner::operator () () { // inner body
    return _outer.a + _outer.b;
}

short outer::operator () () { // outer body
    a = 1;
    b = 2;
    return inner();
}

```

Certainly, such a scheme is the fullest in the sense of saving the program logical structure (in comparison to the de-nest approach) and the program functionality (in comparison to the namespaces approach). Nevertheless, we can list the following (rather negative) specific properties of this approach:

- rather difficult class implementation simulating nested structure;
- losing of simple access to the objects of parent functions (this approach requires a full name specification).

The last issue can be considered as positive, because the names of objects of nested functions do not overlap the names of parent functions and, on the whole, clarify the functionality of nested functions.

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, its value will
                      further be used in expressions of CCC
                      (and
                      only in them)}
...
{%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 value of a CCC-variable may be changed by a later `%set` command. To avoid the warnings of GCC about redefining constants in `#define` directives, the scheme of translation can be slightly changed:

```

{%set debug:=TRUE} → #define debug_ccc TRUE
...
{%set debug: = FALSE} → #ifndef debug_ccc
                        #undef debug_ccc
                        #endif
                        #define debug_ccc FALSE

```

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.

Example 1.

```
{%if debug} if a=1 then begin {%end}
    b:=c; d:=e;
{%if debug} end; {%end}
```

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 “not translated” CCCs so that they become “translated”.

Here is an example of the CCC which cannot be translated:

```
B: = {%if ...} A; {%endif}
    {%if ...} C; {%endif}
```

'B: = A; ' will be successfully parsed, but an error message will be given when 'C;' is parsed.

Definition 1. Let us call a simple CCC the one of the kind

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

...

Example 1 contains two simple CCCs.

Definition 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:

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

or

2) any_text1 and any_text2 are declarations and they declare the same identifier.

Definition 3. Two CCCs of kind 2 in which 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 2.

```
{%IF MODE = MAIN}
  table_infor: table_type;
{%END}
{%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 3.

```
table_infor:
{%IF MODE = MAIN}                                {1st CCC}
  table_type;
{%END}
{%IF MODE = DEBUG}                                {2st CCC}
  ARRAY [debug_files] OF select_record;
{%END}
{%IF MODE = RELEASE}                                {3st 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 4.

```
{%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 1, Example 2 and Example 4 if there are no CCCs similar to Example 3. Example 3 is “bad” because parser does not know if the second (third) CCC is alternative to the first (second) 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 “itera-

tive” 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), initial-value, "ORIF" Boolean-expression "THEN" statement (grammar sequence), case-list-element.

Assertion.

1. 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. (*)
2. Any simple CCC containing any program fragment can be translated into C++ CCC successfully. (**)
3. CCCs meeting the syntactic property (*) or property (**), and only them, are valid for translation.

Example 5.

```
{%if boolean_expr} IF expression THEN BEGIN {%END}
    statement-sequence
{%if cond_expr}
    END;
{%END}
```

The CCC is simple; therefore, it meets condition (**).

Example 5a.

The CCCs with alternative (CCSsA)

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

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

Example 6.

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).

Example 6a.

The CCCs with alternative

```
a: integer: = {%IF DEBUG}      [1] {%END}
              {%IF not DEBUG} [2] {%END};
```

do not meet property (**) by its definition and (*) because they contain program fragments derived from non-iterative non-terminal (*initial-value-list*).

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 an 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 of IR – “Variable A declaration node”.

- To preserve the semantics of the routine’s input parameters, only one out of two existing “A”s should be considered. For example, in the call of *proc*, it is necessary to know the exact type of “A” to generate 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:

1. 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.
2. 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 CCC-stack is being copied to a special storage inside 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*.
3. 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 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 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 can not 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

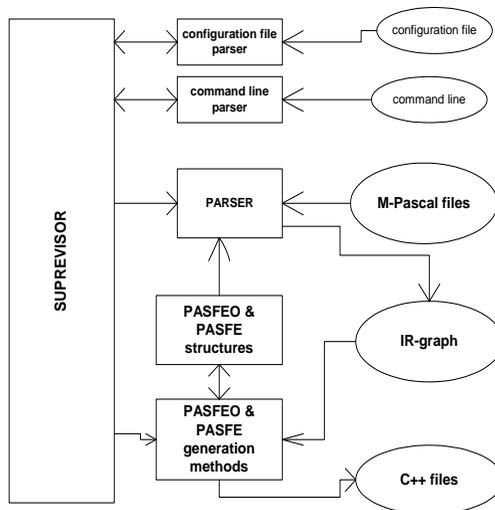
```

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. IMPLEMENTATION OF THE CONVERTER

6.1. The general structure of the converter

The converter consists of the following modules: supervisor, parser, PASFEO structures, PASFE structures, and code generator. The modules of PASFEO and PASFE structures can be considered as libraries.



Supervisor works with a configuration file, command line, and OS environment and calls other components.

Parser consists of lexical, syntactic and semantic analyzers. Parser transforms text files containing M-Pascal modules into the internal representation (IR) graph. IR-graph is an acyclic graph with nodes which are instances of PASFEO

and PASFE structures. The program parse consists of the following parts: the main algorithm (which is a syntax analyzer) reads lexemes from the specified input stream by invoking the lexical analyzer algorithm and parses M-Pascal declarations and statements according to some rules declared in *.y files. (For coding the parser stage of the converter, **GNU Bison** and **Flex** utilities were used. The input grammar of M-Pascal was written as the rules of the utility Bison and then were converted into the C code to build it into the whole converter.) Parsing a current statement, this algorithm also processes semantic attributes. For example, the semantic attribute of a constant declaration construction is a constant value.

PASFEO module. The PASFEO module (abbreviation PAS denotes Pascal, abbreviation FEO denotes Front-End Objects) is a set of structures intended to supply all types, data and functions necessary for checking scopes (visibility) of named program objects, like functions and types. This module serves for storing and retrieving the name scope information during the source code parsing and for holding information about the named program objects used in M-Pascal programs. Parser calls it when a new object in the program has to be created or existing one is queried by name. All PASFEO structures are inherited from one parent, PASFEO_Object. Each PASFEO object has a name member storing the name of this object. It is possible to determine which construction represents a certain PASFEO object. For this purpose a PASFEO object has a set of member functions used by the syntax analyzer.

PASFE module. The PASFE module (abbreviation PAS denotes PASCAL, FE denotes Front-End) is a set of structures intended to supply all types, data and functions necessary for so-called Front-End processing, i.e. for generation of Internal Representation (IR) from the parse tree. The module PASFE describes all possible kinds of IR-graph nodes and information contained in them. The IR-graph is a directed acyclic graph (in simple cases – a tree) obtained after parsing the source M-Pascal program code. Most of the nodes of this graph (or tree) correspond to some clauses (syntax constructions) of M-Pascal, comments and some compile-time facilities of the M-Pascal language environment. Thus, there are nodes for type, label, constant, variable declarations, procedure and function declarations, statements, operators, etc. and for STACKSIZE, INCLUDE, IF, END and SET compile-time facilities. Aside from nodes corresponding to syntax constructions, there are some auxiliary nodes regarding to the code generator needs. Each node represents a structure with pointers to other nodes and (or) program objects. Nodes also contain information obtained at the scope resolution stage – particularly the pointers to named and unnamed program objects, i.e., types, vari-

ables, constants, functions, etc. The same node may correspond to different syntax constructions depending on the value of one of its members.

Codegenerator. All classes corresponding to IR-graph nodes have a common parent – the class OM. The class OM encapsulates information about an object, such as its location, type of lexeme, etc. It implements memory allocation functions to allocate memory in a more definite and effective way. This class has a virtual method

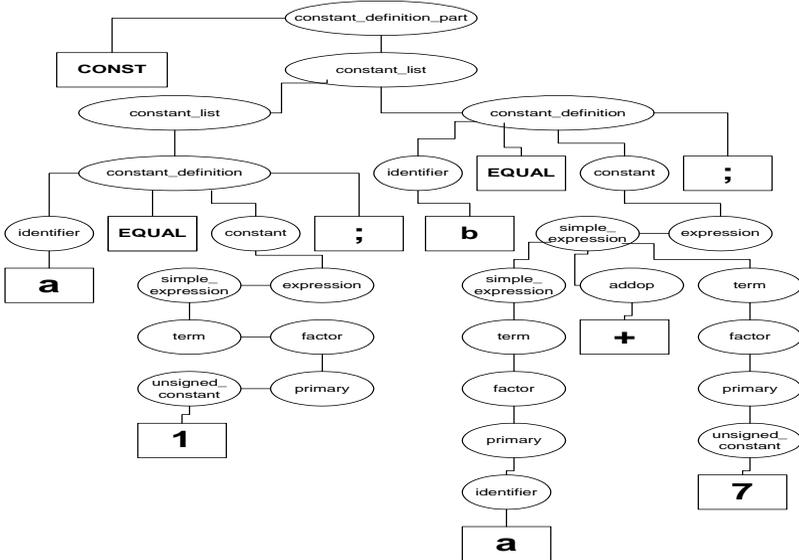
```
virtual void Iterate(Translate& msg)
```

which performs an action over an object. The action is described within *msg* function object. The *msg* parameter holds the translation message (output file name, buffers, and so on). The method is virtual and is overridden for every class of objects individually. A set of Iterate methods is Codegenerator itself. If the object contains other objects (i.e., it is a container), the method applies *msg* to these nested objects too, thus propagating the action through all levels of the object structure.

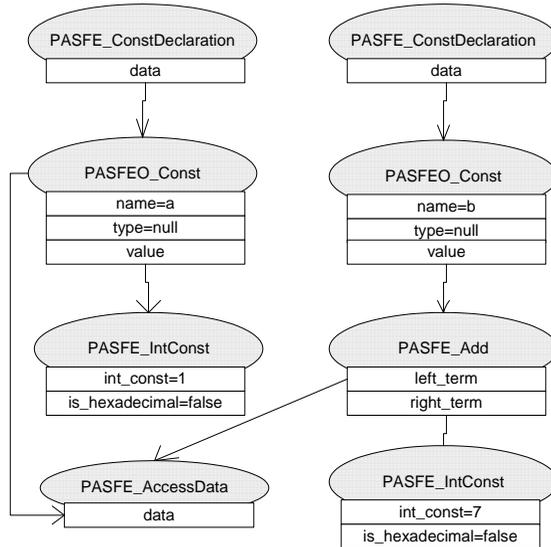
To illustrate the process of conversion, we consider application of the converter to a small piece of M-Pascal code:

```
const a=1;
      b=a+7;
```

By parsing this code, the parser builds the following parse-tree:



Simultaneously, the parser generates IR-graph from PASFEO and PASFE objects. The fragment of IR-graph corresponding to the parse-tree is the following:



This IR-graph is passed to Codegenerator.

The code generation process (this part of the converter is also called Back-End) calls the Iterate method for the root node of IR-graph prepared by Parser (in the figures above the root node is omitted). The root node of the IR-graph is always a structure of *PASFE_CompilationUnit* type. When Codegenerator finishes with the sample IR-graph, the following C++ code is generated:

```
const int a = 1;
const int b = a + 7;
```

It is possible that the name of an identifier in an M-Pascal program coincides with a C++ keyword. To avoid this conflict, the user inserts all C++ keywords (reserved identifiers, the names of the library functions, etc.) in a special file. According to the requirements to the converter, all M-Pascal identifiers are converted in the lowercase format. When the converter meets the identifier that corresponds to some C language identifier (from the special file), it converts the first letter of the identifier in the uppercase. That allows us to avoid errors at the C++ code compilation stage.

6.2. Operation modes of the converter

The converter can operate in the so-called batch mode by defining the set of input M-Pascal files. This set can be specified by wildcards in the command line. According to the indicated template, the converter searches for the corresponding files in folders specified in the configuration file and organizes the list of found files. Then, for each file from the list, the parser and codegenerator run.

As M-Pascal allows separate compilation of different modules, the M-Pascal software package consists of a large amount of so-called header files and common Pascal files. Header files contain declarations of global types, global variables and constants, and global procedures and functions. Common Pascal files contain the bodies of procedures declared in the header files. Besides, they may include many other header files (by using “include” directives similar to C++ ones). Since the sets of used header files by different Pascal files intersect, the converter needs to regenerate the target C++ code many times for the most of the header files in the batch mode.

Therefore, several other operation modes have been implemented to reduce the translation time on a large amount of input source files:

- Common mode — the converter generates the C++ code for each header and Pascal file at the parser/codegenerator stage of the converter.
- Only Pas mode — the converter parses all the header files found, builds the IR-graph in memory for them but does not generate the C++ file if it already exists. Therefore, the codegenerator stage is switched off for header files.
- Quick mode — the converter neither generates C++ file, as in the previous mode, nor rebuilds the IR-graph for a header file. The converter preserves all the pointers to the roots of trees built in a special table. As required, the converter searches for the necessary tree and attaches it to the current tree.

The last mode may be considered as a “precompiled headers” mode implemented in some compilers. The second mode is necessary for debugging purposes only, since it is impossible to obtain any debug information in the third mode.

Additionally, it takes much time for the converter to format the output C++ code because of complicated requirements stated. This essentially increases the converter run time on a large number of input files. So, an additional option has been introduced into the converter, which allows us to switch off the formatting, thus reducing translation time for debug purpose.

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 different approaches to the development of these schemes and we hope that it may be useful for other developers.

REFERENCES

1. **Aho A.V., Ullman J.D.** The Theory of Parsing, Translation and Compiling. Vol.1: Parsing. — Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. **Aho A.V., Sethi R., Ullman J.D.** Compilers: Principles, Techniques, and Tools. — Addison-Wesley, Reading, Mass, 1988.
3. **Kasyanov V.N., Pottosin I.V.** Methods of compiler development. — Novosibirsk, Nauka, 1986 (in Russian).
4. GNU Pascal To C converter:
http://www.ibiblio.org/pub/Linux/devel/lang/pascal/ptoc_3.34.tar.gz.
5. FreeBSD Pascal To C converter:
<ftp://ftp.freebsd.org/pub/FreeBSD/ports/i386/packages-4-stable/lang/ptoc-3.50.tgz>.
6. MSC Pascal To C/C++ converter:
<http://lib.buryatia.ru/cgi-bin/win/SOFTWARE/ptoc.README.txt>.
7. **Jensen K., Wirth N.** Pascal. User Manual and Reports. — Springer-Verlag, 1978.
8. Program language C++ standard (ANSI/ISO/IEC 14882).
http://www.techstreet.com/cgi-bin/detail?product_id=49964.

В. Маркин, С. Маслов, Р. Новиков, А. Сулимов

КОНВЕРТОР С РАСШИРЕННОГО ПАСКАЛЯ В C++

**Препринт
92**

Рукопись поступила в редакцию 25.12.01

Рецензент В.А. Евстигнеев

Редактор А. А. Шелухина

Подписано в печать 15.05.02

Формат бумаги 60 × 84 1/16

Тираж 50 экз.

Объем 1.8 уч.-изд.л., 2.0 п.л.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6