κεφάλαιο

25

Περιγράμματα Κλάσεων

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα μάθουμε πώς μπορούμε να υλοποιήσουμε δομές δεδομένων με προγραμματιστικά αντικείμενα της C++: τα περιγράμματα κλάσεων.

Προσδοκώμενα αποτελέσματα:
Όπως με ένα περίγραμμα συνάρτησης μπορείς να υλοποιήσεις προγραμματιστικώς έναν αλγόριθμο, με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων ώστε να μπορείς να έχεις στιγμιότυπα για τους τύπους που σε ενδιαφέρουν.

Έννοιες κλειδιά:

- *περίγραμμα κλάσης*
- *παράμετρος περιγράμματος*
- *εξειδίκευση περιγράμματος*
- *μερική εξειδίκευση περιγράμματος*
- *περιγράμματα και κληρονομιές*
- *περιέχουσα κλάση*
- *περιεχόμενη κλάση*
- *έξυπνα βέλη*
- *κλάσεις χαρακτηριστικών*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Όπως είναι χρήσιμο να έχουμε *περιγράμματα συναρτήσεων*, όπου μεταφράζουμε σε C++ γενικούς αλγόριθμους με τους τύπους των στοιχείων να είναι παράμετροι, είναι χρήσιμο να έχουμε και **περιγράμματα κλάσεων**[1] (class templates) στα οποία μπορούμε να μεταφράσουμε σε C++ δομές δεδομένων και τους αλγόριθμους για τον χειρισμό τους.

Παρόμοιες δυνατότητες προσφέρουν και άλλες γλώσσες προγραμματισμού και –μεταξύ αυτών– τα άλλα «παιδιά της C»:

- Η Java έχει τους **γενικούς τύπους** (generic types).
- Η C# έχει τις **γενικές κλάσεις** (generic classes).

Ο τύπος (*std*::)*string*, που χρησιμοποιούμε συνεχώς στα προγράμματά μας, είναι ένα στιγμιότυπο του περιγράμματος *basic_string*. Ακολουθώντας και εμείς αυτήν την ιδέα(!) θα ξεκινήσουμε την παρουσίαση των περιγραμμάτων κλάσεων ως εξής:

- Από τη *BString* θα πάρουμε το περίγραμμα κλάσης *BStringT*.
- Θα ξαναπάρουμε την αρχική κλάση ως στιγμιότυπο του περιγράμματος.

25.1 Από την Κλάση *BString* στο Περίγραμμα

Στα προηγούμενα κεφάλαια αναπτύξαμε το παράδειγμα της κλάσης *BString*. Επειδή, η κλάση αυτή είναι χρήσιμη γενικότερα, θα θέλαμε να την μετατρέψουμε σε *περίγραμμα* ώστε να μπορούμε να τη χρησιμοποιήσουμε, όχι μόνον με τον `char`, αλλά και με άλλους τύπους, όπως ο `unsigned char` ή ο `wchar_t`. Η C++ μας δίνει τη δυνατότητα να έχουμε, εκτός από περιγράμματα συναρτήσεων, και περιγράμματα κλάσεων.

Στην κλάση, όπως την έχουμε μέχρι τώρα, διαχωρίζουμε τον ερήμην δημιουργό από τον δημιουργό με αρχική τιμή:

```
 0: class BString
 1: { // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
 2: friend bool operator==( const BString& lhs,
 3:                         const BString& rhs );
 4: public:
 5:    BString();
 6:    BString( const char* rhs, int n=0 );
 7:    BString( const BString& rhs );
 8:    ~BString() {  delete[] bsData;  };
 9:    BString& operator=( const BString& rhs );
10:    BString& assign( const BString& rhs )
11:                                    {  return (*this = rhs);  }
12:    const char* c_str() const
13:                     {  bsData[bsLen] = '\0';  return bsData;  }
14:    size_type length() const {  return bsLen;  }
15:    bool empty() const {  return ( bsLen == 0 );  }
16:    char& at( int k ) const;
17:    char& operator[]( int pos ) const {  return bsData[pos];  }
18:    BString& operator+=( const BString& rhs );
19:    BString& append( const BString& rhs )
20:                                   {  return (*this += rhs);  }
21:    void swap( BString& rhs );
22:    int compare( const BString& rhs ) const;
23: private:
24:    static const unsigned int bsIncr = 16;
25:    char*     bsData;
26:    size_type bsLen;
27:    size_type bsReserved;
```

[1] Θα δεις και τον όρο **παραμετρικές** (parametric) κλάσεις.

```
 28:
 29:    static size_type cStrLen( const char* cs );
 30:    static int stringCmpr( const char* lhs, const char* rhs,
 31:                           int n );
 32: }; // BString
```

Ο ορισμός του ερήμην δημιουργού είναι:

```
BString::BString()
{
   try {  bsData = new char[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BString::BString
```

Βάλαμε μέσα στη δήλωση της κλάσης τους ορισμούς των τριών μεθόδων `inline` και του καταστροφέα.

Πέρα από τη φίλη καθολική `operator==()` να θυμηθούμε ότι έχουμε και την καθολική:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

Για να πάρουμε περίγραμμα κλάσης, ας το πούμε *BStringT*, από την ειδική περίπτωση ορμαθών με στοιχεία τύπου `char`, κάνουμε τα εξής:

- Αλλάζουμε το `'\0'` σε "`char(0)`" (στη γρ. 13)
- Αντικαθιστούμε τον τύπο "`char`"[2], όπου τον βρούμε (γρ. 6, 12, 16, 17, 25, 29, 30), με μια παράμετρο, ας την πούμε "`K`".
- Αντικαθιστούμε το "`BString`", όπου τον βρούμε, με "`BStringT`".

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT& lhs,
                        const BStringT& rhs );
public:
   BStringT();
   BStringT( const K* rhs, int n=0 );
   BStringT( const BStringT& rhs );
   ~BStringT() {  delete[] bsData;  };
   BStringT& operator=( const BStringT& rhs );
   BStringT& assign( const BStringT& rhs )
                                        {  return (*this = rhs);  }
   const K* c_str() const
                          {  bsData[bsLen] = K(0);  return bsData;  }
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   K& at( int k ) const;
   K& operator[]( int pos ) const {  return bsData[pos];  }
   BStringT& operator+=( const BStringT& rhs );
   BStringT& append( const BStringT& rhs )
                                       {  return (*this += rhs);  }
   void swap( BStringT& rhs );
   int compare( const BStringT& rhs ) const;
private:
   static const unsigned int bsIncr = 16;
   K*        bsData;
   size_type bsLen;
   size_type bsReserved;
```

[2] Με το "replace" του κειμενογράφου!

```
   static size_type cStrLen( const K* cs );
   static int stringCmpr( const K* lhs, const K* rhs, int n );
}; // BStringT
```

Βλέπεις φυσικά και το –γνωστό μας από τα περιγράμματα συναρτήσεων– «καπέλο» "`template < typename K >`".

Τώρα, αντί για τη δήλωση:

```
   BString s1( "abc" ), s2, s3;
   BString bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

μπορούμε να γράψουμε:

```
   BStringT<char>  s1( "abc" ), s2, s3;
   BStringT<char>  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Λέμε δηλαδή ότι ο τύπος (κλάση) είναι ένα *στιγμιότυπο* του περιγράμματος κλάσης *BStringT* για παράμετρο-τύπο `char`.

Στη βιβλιοθήκη της C++ ορίζεται κατ' αρχάς το *basic_string* που είναι περίγραμμα κλάσης. Η *string* ορίζεται ως στιγμιότυπο του περιγράμματος:

```
typedef basic_string<char> string;
```

Παρομοίως «βαφτίζεται» και ένα άλλο στιγμιότυπο:

```
typedef basic_string<wchar_t> wstring;
```

Μιμούμενοι τα παραπάνω, μετά τον ορισμό του περιγράμματος, θα μπορούσαμε να ορίσουμε:

```
typedef BStringT< char >                  String;
```

και να γράψουμε τις παραπάνω δηλώσεις:

```
   String  s1( "abc" ), s2, s3;
   String  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Η *String* δεν είναι άλλη από τη *BString*.

Μπορούμε ακόμη να ορίσουμε:

```
typedef BStringT< wchar_t >               WString;
typedef BStringT< unsigned char >         UString;
```

και να δηλώσουμε:

```
   WString s1( L"abcd" ), s2( L"abcd" );
```

Να δούμε τώρα πώς γράφονται οι ορισμοί των μεθόδων. Για τον καταστροφέα, τη *c_str()*, τη *length()*, την *empty()*, την *assign()* και την *append()* δεν υπάρχει πρόβλημα· ορίζονται με τις δηλώσεις τους. Ας δούμε την `operator+=()` που επιφορτώνει τον "`+=`" και την ορίζουμε έξω από τη δήλωση της κλάσης. Όπως καταλαβαίνεις, αφού η *BStringT* έγινε περίγραμμα κλάσης και η `operator+=()` είναι πια *περίγραμμα συνάρτησης*. Φυσικά θα έχει τις ίδιες παραμέτρους με την κλάση. Άρα ο ορισμός της θα αρχίζει με τα: "`template < typename K >`".

Αν γράφαμε τη μέθοδό μας για την κλάση `BStringT<char>` η επικεφαλίδα της θα ήταν:

```
BStringT<char>& BStringT<char>::operator+=( const BStringT<char>& rhs )
```

ενώ, αν τη γράφαμε για τη `BStringT<wchar_t>` θα ήταν:

```
BStringT<wchar_t>&
                BStringT<wchar_t>::operator+=( const BStringT<wchar_t>& rhs )
```

Μπορούμε λοιπόν να μαντέψουμε ότι όταν έχουμε στοιχεία κλάσης *K* θα πρέπει να έχουμε:

```
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
```

Πράγματι, το περίγραμμα της μεθόδου (συνάρτησης) θα είναι:

```
template < typename K >
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
```

```
      K* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new K[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringTXptn( "operator+=",
                             BStringTXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BStringT<K>::operator+=
```

Πρόσεξε ότι οι αλλαγές είναι

- στην επικεφαλίδα,
- στη δήλωση "`char* tmp`" που έγινε "`K* tmp`" και
- στη "`new char[tmpRes]`" που έγινε "`new K[tmpRes]`".

Να δούμε πώς γίνεται ο ερήμην δημιουργός, που είδαμε και πιο πάνω:

```
template < typename K >
BStringT<K>::BStringT()
{
   try {  bsData = new K[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringTXptn( "BStringT",
                          BStringTXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BStringT<K>::BStringT<K>
```

Πρόσεξε ότι το όνομα του δημιουργού είναι `BStringT` και όχι `BStringT<K>`.

25.2 Φίλες Συναρτήσεις Περιγραμμάτων

Ας δούμε τώρα τι γίνεται με τις φίλες συναρτήσεις: Κατ' αρχήν, δηλώνονται όπως ακριβώς ξέρουμε:

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Φυσικά, και αυτές γίνονται τώρα περιγράμματα συναρτήσεων και έτσι δίνονται οι ορισμοί τους, π.χ.:

```
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int int fv( BStringT<K>::stringCmpr(lhs.bsData, rhs.bsData,
                                       min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==( const BStringT<K>&
```

Ωραία όλα αυτά, αλλά –όπως είπαμε– «κατ' αρχήν». Το πιο πιθανό είναι ότι ο μεταγλωττιστής σου θα τα απορρίψει! Γιατί; Διότι δεν έχει την «υπομονή» να περιμένει να βρεί τον ορισμό της συνάρτησης `operator==()` και μόλις βρει τη δήλωση `friend` μας λέει ότι δεν ξέρει τέτοια συνάρτηση.

Η πιο απλή λύση στο πρόβλημά μας είναι η εξής: βάζουμε τον ορισμό της συνάρτησης μαζί με τη δήλωση `friend`:

```
template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int fv( stringCmpr(lhs.bsData, rhs.bsData,
                      min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Αυτό θα περάσει από τον μεταγλωττιστή. Πάντως ορισμένοι μεταγλωττιστές (π.χ.: gcc) δεν θα καταλάβουν ότι αυτό είναι περίγραμμα συνάρτησης και θα πρέπει να δηλώσεις:

```
template < typename K1 >
friend bool operator==( const BStringT<K1>& lhs, const BStringT<K1>& rhs )
// . . .
```

Σημείωση:▶
Αν θέλεις να βάλεις τον ορισμό της φίλης συνάρτησης έξω από τον ορισμό της κλάσης θα πρέπει θα *πρέπει να βάλεις πριν από την κλάση προειδοποιητικές δηλώσεις (§22.4) της κλάσης και της συνάρτησης*:

```
template < typename K > class BStringT;
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );

template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
// . . .
```

Στη συνέχεια, μπορείς να βάλεις τον ορισμό της συνάρτησης όπου θέλεις.◀

25.3 Καθολικές Συναρτήσεις για Περιγράμματα

Περιγράμματα συναρτήσεων θα γίνουν και οι καθολικές συναρτήσεις που έχουν σχέση με κάποιο περίγραμμα κλάσης. Δες πώς θα γίνει η (καθολική συνάρτηση) `operator<<()` που επιφορτώνει τον "`<<`":

```
template < typename K >
ostream& operator<<( ostream& tout, const BStringT<K>& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BStringT<K>
```

Παρατήρηση:▶
Θα πρέπει να παρατηρήσουμε εδώ ότι αν πάρουμε στιγμιότυπο του περιγράμματος για `wchar_t`, η `operator<<()` δεν δουλεύει όπως θέλουμε! Θα χρειαστείς εξειδίκευση της συνάρτησης (όχι της κλάσης).◀

25.3.1 Ένα Απλό Παράδειγμα: *pair*

Αν βάλεις στο πρόγραμμά σου "`#include <utility>`" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*`::`*pair* που βάζει σε μια `struct` δύο μεταβλητές και χρησιμοποιείται στην STL. Εδώ θα γράψουμε ένα περίγραμμα κλάσης

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;
// . . .
}; // PairT
```

που θα έχει ακριβώς τις ιδιότητες του *pair*: Μια **struct** με δύο μέλη και λεξικογραφική διάταξη. Τι θα πει αυτό; Ορίζεται ο "<" ως εξής: Αν

```
PairT<T1, T2> x, y;
```

τότε η x < y ισχύει αν και μόνον αν:

$$(x.first < y.first) \;||\; (x.first == y.first \;\&\&\; x.second < y.second)$$

ή, ισοδυνάμως:

$$x.first < y.first \;||\; (!(y.first < x.first) \;\&\&\; x.second < y.second$$

Αυτή η μορφή[3] *δεν έχει σύγκριση για ισότητα* και επομένως είναι προτιμότερη αν ο *T1* είναι τύπος κινητής υποδιαστολής.

Ο ερήμην δημιουργός θα είναι:

```
   PairT()
      :  first( T1() ), second( T2() ) { };
```

πράγμα που *προϋποθέτει την ύπαρξη ερήμην δημιουργών για τους T1 και T2*.

Ένας δημιουργός με αρχικές τιμές θα είναι:

```
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
```

Για να μπορούμε να γράψουμε τα "**first(x)**" και "**second(y)**" θα πρέπει *οι T1 και T2 να έχουν δημιουργούς με αρχική τιμή*.

Για τύπους *U* και *V* που έχουν δημιουργούς μετατροπής του *U* στον *T1* και του *V* στον *T2* μπορούμε να γράψουμε

```
   PairT( const PairT<U,V>& a )
      : first( a.first ), second( a.second ) { };
```

Και αυτό πώς μπαίνει μέσα στη δήλωση του περιγράμματος; Ως περίγραμμα μέσα σε περίγραμμα!

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT
```

Όπως καταλαβαίνεις, κάθε στιγμιότυπο του *PairT* θα είναι μια κλάση που θα περιέχει ένα περίγραμμα δημιουργού.

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε τον "<" ως:

```
template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
                          (!(y.first < x.first) && x.second < y.second) );  }
```

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε και τον "==":

[3] που την προτιμάει το πρότυπο…

```
template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }
```

όπου φυσικά δεν μπορούμε να αποφύγουμε τη σύγκριση για ισότητα όποιοι και αν είναι οι *T1*, *T2*.

Τέλος, το περίγραμμα καθολικής συνάρτησης:

```
template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }
```

ζευγαρώνει δύο τιμές *x*, *y* τύπων *T1*, *T2* αντιστοίχως.

Ας καταγράψουμε τις απαιτήσεις από τους *T1*, *T2* για να μπορούμε να ζευγαρώνουμε τιμές τους με το *PairT*: Θα πρέπει

- Να έχουν ερήμην δημιουργούς.
- Να έχουν δημιουργούς αρχικής τιμής.
- Να έχουν τον τελεστή "`<`".
- Να έχουν τον τελεστή "`==`".

Και κάτι ακόμη: χωρίς να μπορούμε να επιφορτώσουμε τον "`=`" –αφού δεν ξέρουμε τους *T1*, *T2*– θα πρέπει η εκχώρηση να γίνεται σωστά. Αυτό σημαίνει ότι: *ο τελεστής εκχώρησης "`=`" δουλεύει σωστά για τους T1, T2*.

Όσο για τη χρήση ενός τέτοιου περιγράμματος να πούμε τα εξής: Όπως θα καταλάβεις αργότερα, χρησιμοποιώντας τα εργαλεία της STL στο παράδειγμα με τους φοιτητές και τα μαθήματα, θα δημιουργηθούν ζεύγη τύπων όπως:

```
PairT< CourseKey, Course >
PairT< unsigned int, Student >
```

Βέβαια, μπορεί να σου κατέβουν και άλλες ιδέες. Θα μπορούσες να γράψεις τη *minmax* που είδαμε στην §13.9.3 (Παρ. 1) έτσι:

```
PairT< int, int > minmax( int x, int y )
{
   PairT< int, int > fv;

   if ( x < y )
      fv = PairT< int, int >( x, y );
   else // x >= y
      fv = PairT< int, int >( y, x );
   return fv;
} // minmax
```

ή την *pqer* (που είδαμε στις §14.9 και §15.10) ως εξής:

```
PairT< double, double > pqer( double x, double y, double z )
{
   if ( x == y || x == -y || z <= 0 )
      throw ApplicXptn( "pqer", ApplicXptn::paramErr, x, y, z );
   PairT< double, double > fv;
   fv.first = x*y*pow(z, x-y)/(x*x-y*y);
   fv.second = (x*y-1/x)/z;
   return fv;
} // pqer
```

Έτσι, έχουμε συναρτήσεις με τύπο που λέγαμε ότι είναι προτιμότερες από τις "`void`". Βεβαίως, αλλά στην περίπτωση αυτήν ο τύπος είναι κάπως «φτιαχτός»…

25.4 Κατανομή σε Αρχεία

Πριν προχωρήσουμε, ας κάνουμε μια επισήμανση, συμπλήρωμα στην §19.3: Το περίγραμμα μιας κλάσης γράφεται, κατ' αρχήν, σε ένα αρχείο, το "`.h`". Έτσι, για παράδειγμα, το *PairT* θα φυλαχτεί σε ένα αρχείο, το `PairT.h`, που θα έχει:

```
#ifndef _PAIRT_H
#define _PAIRT_H

using namespace std;

template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT

template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }

template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
           (!(y.first < x.first) && x.second < y.second) );  }

template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }

#endif // _PAIRT_H
```

25.5 Παράμετροι και Εξειδικεύσεις

Στην §14.7.1 λέγαμε ότι ένα περίγραμμα (συνάρτησης) «*μπορεί να έχει ως παραμέτρους*

- *έναν ή περισσότερους τύπους ή/και*
- *ακέραιους,*
- *βέλη ή αναφορές.*»

Ας τα δούμε με περισσότερες λεπτομέρειες. Οι παράμετροι ενός περιγράμματος μπορεί να είναι:

- Το λεξικό σύμβολο "`class`" ή "`typename`" ακολουθούμενο από όνομα της παραμέτρου. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμη (τύπο) της παραμέτρου.

  Τα "`class`" και "`typename`" δεν έχουν οποιαδήποτε νοηματική διαφορά. Μπορείς να χρησιμοποιείς όποιο θέλεις.[4]
- Το λεξικό σύμβολο "`template`" ακολουθούμενο από λίστα παραμέτρων περιγράμματος μέσα σε "`<`" και "`>`" ακολουθούμενη από το σύμβολο "`class`" και ένα όνομα. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμή (περίγραμμα) της παραμέτρου. Το παρακάτω παράδειγμα είναι από το πρότυπο C++03:

```
template< typename T > class myarray { /* ... */ };

template< typename K, typename V,
          template< typename T> class C = myarray >
class Map
```

[4] Πάντως το "`typename`" είναι πιο σωστό –παρ' όλο που ορισμένοι μεταγλωττιστές έχουν πρόβλημα με αυτό– αφού μπορείς εκεί να βάλεις "`int`" ή "`double`" (ή άλλα παρόμοια) που δεν είναι κλάσεις.

```
{
   C< K > key;
   C< V > value;
// ...
};
```

Όπως βλέπεις η ερήμην καθορισμένη τιμή "`myarray`" είναι περίγραμμα κλάσης της μορφής "`template< typename T > class C`".

- Παράμετρος ακέραιου ή απαριθμητού τύπου.
- Παράμετρος-βέλος προς αντικείμενο ή συνάρτηση.
- Παράμετρος-αναφοράς αντικειμένου ή συνάρτησης.
- Παράμετρος-βέλος προς μέλος.

Αν κάποιο στιγμιότυπο –ας πούμε το *BStringT*<*AClass*>– που βγαίνει αυτόματως δεν καλύπτει τις απαιτήσεις σου μπορείς, όπως και στα περιγράμματα συναρτήσεων, να κάνεις μια εξειδίκευση:

```
template<> class BStringT<AClass> { /* . . . */ };
```

Αλλά πρόσεχε:

♦ ***Για την εξειδίκευση θα πρέπει να ορίσεις από την αρχή τα πάντα.***

Η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!

Παρατήρηση:▶

Αυτό που θα λύνει το πρόβλημά σου σε πολλές περιπτώσεις είναι η εξειδίκευση μιας ή περισσότερων μεθόδων. Δηλαδή: αφού για κάθε μέθοδο γράφουμε ένα περίγραμμα συνάρτησης μπορούμε να κάνουμε εξειδίκευση του περιγράμματος της μεθόδου για τις περιπτώσεις που μας ενδιαφέρουν.

Έστω για παράδειγμα το περίγραμμα κλάσης:

```
template < typename K > class A
{
public:
   A( const K& b=0 ) {  a = b;  }
   void operator+=( const A& q );
   const K& getA() {  return a;  }
private:
   K a;
}; // template < typename K > class A
```

Για την περίπτωση που παίρνουμε στιγμιότυπο του *A* για `bool` –δηλαδή για την κλάση `A<bool>`– θέλουμε να έχουμε την πράξη "`||`" αντί για τη "`+`" και τη "`&&`" αντί για τη "`*`". Ο τελεστής που έχουμε εδώ θα πρέπει να γίνεται για την κλάση αυτήν "`||=`"(ας πούμε).

Μπορούμε λοιπόν να πούμε:

```
template < typename K >
void A<K>::operator+=( const A<K>& q )
{  a += q.a;  }
```

και να εξειδικεύσουμε:

```
template<>
void A<bool>::operator+=( const A<bool>& q )
{  a = a || q.a;  }
```

Αυτό είναι προτιμότερο από το να γράψουμε μια ολόκληρη κλάση:

```
template<> class A< bool > { /* . . . */ }; // ◀
```

Μια δυνατότητα που υπάρχει για περιγράμματα κλάσεων αλλά όχι για περιγράμματα συναρτήσεων είναι οι *προκαθορισμένες τιμές παραμέτρων του περιγράμματος*. Για παράδειγμα, μετά τον ορισμό:

```
template < typename T1=int, typename T2=char, int n=1 >
class C { /* . . . */ }; // C
```

μπορείς να δώσεις δηλώσεις όπως:

```
C<> ob0;
C< double > ob1;
C< long, int > ob2;
C< unsigned, wchar_t, 2 > ob3;
```

που είναι ισοδύναμες με:

```
C< int, char, 1 > ob0;
C< double, char, 1 > ob1;
C< long, int, 1 > ob2;
C< unsigned int, wchar_t, 2 > ob3;
```

Όπως είναι φυσικό, ισχύει αυτό που είπαμε στην §14.2 για παραμέτρους συναρτήσεων: «*Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.*» Δηλαδή: μη σκεφτείς να γράψεις "`C<3> x;`" επειδή βαριέσαι να γράψεις το σωστό: "`C<int, char, 3> x;`"

25.5.1 Μερική Εξειδίκευση Περιγράμματος Κλάσης

Όπως είδαμε, σε ένα περίγραμμα συνάρτησης μπορείς να επιφορτώσεις ένα ή περισσότερα περιγράμματα συναρτήσεων ή/και μια ή περισσότερες απλές συναρτήσεις. Επιφόρτωση κλάσεων ή περιγραμμάτων κλάσεων δεν υπάρχει αλλά στην εξειδίκευση περιγραμμάτων κλάσεων έχεις μια δυνατοτητα που δεν υπάρχει για την εξειδίκευση περιγραμμάτων συναρτήσεων: η **μερική εξειδίκευση** (partial specialization) *που δίνει περίγραμμα κλάσης και όχι κλάση.*

Δες μερικά παραδείγματα (από το πρότυπο C++03). Ας πούμε ότι έχουμε το πρωτεύον περίγραμμα:

```
template< typename T1, typename T2, int I > class A { /* . . . */ };
```

που έχει δύο παραμέτρους-τύπους και μια ακέραιη παράμετρο.

α) Μια εξειδίκευση είναι η εξής:

```
template< typename T1, typename T2, int I >
class A< T1*, T2, I > { /* . . . */ };
```

Εδώ περιορίζεται ο τύπος της πρώτης παραμέτρου του περιγράμματος: είναι τύπος βέλους.

β) Η εξειδίκευση:

```
template< typename T1, typename T2, int I >
class A< T1, T2*, I > { /* . . . */ };
```

είναι σαν την πρώτη αλλά θέλουμε τύπο βέλους στη δεύτερη παράμετρο.

γ) Μια άλλη εξειδίκευση είναι η εξής:

```
template< typename T, int I > class A< T, T*, I > { /* . . . */ };
```

όπου οι *T1* και *T2* δεν είναι άσχετες μεταξύ τους: η δεύτερη είναι βέλος προς αντικείμενο της πρώτης.

δ) Μια τέταρτη εξειδίκευση είναι η:

```
template< typename T > class A< int, T*, 5 > { /* . . . */ };
```

Εδώ καθορίζεται η πρώτη παράμετρος να είναι ο τύπος "`int`" και η τρίτη να έχει τιμή "`5`".

Θα πρέπει να επιστήσουμε την προσοχή σου στον ορισμό των μεθόδων ενός μερικώς εξειδικευμένου περιγράμματος κλάσης.

Όταν ζητήσουμε ένα στιγμιότυπο ενός περιγράμματος κλάσης τότε γίνεται το εξής: Ο μεταγλωττιστής προσπαθεί να ταιριάσει τις τιμές των παραμέτρων του στιγμιότυπου με αυτά των εξειδικεύσεων.

- Αν βρει μια μόνον εξειδίκευση που να ταιριάζει παράγει το στιγμιότυπο από αυτήν.
- Αν βρει δύο ή περισσότερες εξειδικεύσεις επιλέγει την πιο εξειδικευμένη. Αν δεν μπορεί να διαλέξει βγάζει λάθος.
- Αν δεν βρει εξειδίκευση που να ταιριάζει θα βγάλει το στιγμιότυπο από το αρχικό περίγραμμα.

Ας πούμε ότι έχουμε το περίγραμμα και τις εξειδικεύσεις που είδαμε παραπάνω και έχουμε την

```
A< int, int, 1 > a1;
```

Οι παράμετροι του στιγμιότυπου δεν ταιριάζουν με οποιαδήποτε από τις εξειδικεύσεις. Έτσι, θα παραχθεί από το αρχικό περίγραμμα βάζοντας "`int`" στους *T1*, *T2* και "`1`" στην *I*.

Ας δούμε τώρα τη δήλωση:

```
A< int, int*, 1 > a2;
```

Οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στους *T1* και *T2* και "`1`" στην *I*.
- Με την εξειδίκευση (γ) αν βάλουμε "`int`" στον *T* και "`1`" στην *I*.

Η (γ) είναι πιο εξειδικευμένη και από αυτήν θα παραχθεί το στιγμιότυπο.

Για τη δήλωση:

```
A< int, char*, 5 > a3;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`5`" στην *I*.
- Με την εξειδίκευση (δ) αν βάλουμε "`char`" στον *T*.

Το στιγμιότυπο θα παραχθεί από τη (δ) που είναι πιο εξειδικευμένη.

Για τη δήλωση:

```
A< int, char*, 1 > a4;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν μόνο με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`1`" στην *I*.

Τέλος, για τη δήλωση:

```
A< int*, int*, 2 > a5;
```

οι παράμετροι ταιριάζουν με την (α) και τη (β) που όμως είναι το ίδιο εξειδικευμένες. Εδώ ο μεταγλωττιστής, μη μπορώντας να δημιουργήσει τον τύπο της *a5*, θα βγάλει λάθος.

Το πρόγραμμα που ακολουθεί δοκιμάζει τα παραπάνω:

```
#include <iostream>

using namespace std;

template< typename T1, typename T2, int I >
struct A
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1, T2, I >::f()  {   cout << "f 0" << endl;  }

template<> void A< int, int, 0 >::f()
{   cout << "f 0 ovrld" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1*, T2, I >
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1*, T2, I >::f()  {  cout << "f a" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1, T2*, I >
{
   void f();
};
```

```
template< typename T1, typename T2, int I >
void A< T1, T2*, I >::f()  {  cout << "f b" << endl;  }

template< typename T, int I >
struct A< T, T*, I >
{
   void f();
};

template< typename T, int I >
void A< T, T*, I >::f()  {  cout << "f c" << endl;  }

template< typename T >
struct A< int, T*, 5 >
{
   void f();
};

template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }

int main()
{
   A< int, int, 0 > a0;
   A< int, int, 1 > a1;
   A< int, int*, 1 > a2;
   A< int, char*, 5 > a3;
   A< int, char*, 1 > a4;
   A< int*, int*, 2 > a5;

   a0.f();
   a1.f();
   a2.f();
   a3.f();
   a4.f();
   a5.f();
}
```

Για την "`A< int*, int*, 2 > a5;`" ο μεταγλωττιστής[5] θα μας βγάλει λάθος: "`Too many candidate template specializations from 'A<T1,T2,2>' in function main()`". Αν διαγράψουμε αυτήν τη δήλωση καθώς και την εντολή "`a5.f();`" όλα πάνε καλά και παίρνουμε αποτέλεσμα:

```
f 0 ovrld
f 0
f c
f d
f b
```

Τώρα κάνουμε την εξής δοκιμή: αφαιρούμε την

```
template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }
```

Αυτήν τη φορά θα διαμαρτυρηθεί ο συνδέτης: "`Error: Unresolved external 'A<int, char *, 5>::f()' referenced from …`" Όπως λέγαμε πιο πριν «*η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!*» Το ίδιο ισχύει και για τα περιγράμματα που προκύπτουν από μερικές εξειδικεύσεις ενός πρωτεύοντος περιγράμματος: *δεν κληρονομούν ότιδήποτε από αυτό.*

[5] Borland C++ v.5.5.

25.6 Περιγράμματα και Κληρονομιές

Τι δυνατότητες κληρονομιάς έχουν τα περιγράμματα κλάσεων; Ό,τι μπορείς να σκεφτείς.

25.6.1 Κληρονομιά Κλάσης από Περίγραμμα Κλάσης

Μια κλάση μπορεί να κληρονομεί *στιγμιότυπο* περιγράμματος κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

class D :  public B< char >
{
public:
   D( char a1=0, int a2=0 )
      :  B<char>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<char>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Πρόσεξε την προετοιμασία που κάναμε στη βασική κλάση (περίγραμμα) για να μπορεί να κληρονομηθεί: έχουμε δηλώσει **virtual** τον καταστροφέα. Ακόμη κάναμε το ίδιο και στη μέθοδο *display()* αφού περιμένουμε –σε περίπτωση κληρονομιάς– να υπάρξει μέθοδος της παράγωγης κλάσης με το ίδιο όνομα που θα πρέπει να υπερισχύσει.

25.6.2 Κληρονομιά Περιγράμματος Κλάσης από Κλάση

Μερικές φορές μπορεί να θέλεις να χρησιμοποιήσεις –σε ένα περίγραμμα που γράφεις– εργαλεία που έχεις σε κάποια κλάση. Ένας τρόπος είναι και η κληρονομια. Για παράδειγμα:

```
class B
{
public:
   B( int a=0 ) :  bCount( a ) { };
   virtual ~B() { };
   unsigned int getCount() const {  return bCount;  }
// . . .
protected:
   unsigned int bCount;
}; // B

template < typename C, size_type sz=10 >
class D :  public B
{
public:
   D( int a=sz )
   {  if ( a <= 0 )  throw a;
      dArr = new C[a];   bCount = a;  };
   virtual ~D() {  delete[] dArr;  };
// . . .
private:
   C* dArr;
}; // D
```

25.6.3 Κληρονομιά Περιγράμματος Κλάσης από Περίγραμμα

Η πιο ενδιαφέρουσα από τις τρεις περιπτώσεις είναι η κληρονομιά περιγράμματος κλάσης από άλλο περίγραμμα κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

template < typename C >
class D :  public B< C >
{
public:
   D( C a1=0, int a2=0 )
      :  B<C>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<C>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Όπως βλέπεις και εδώ προετοιμάζουμε για κληρονομιά το περίγραμμα κλάσης *B*, από το οποίο θα προέλθει η βασική κλάση. Από αυτά που έχουμε μέχρι εδώ δεν έχει ορισθεί κάποια κλάση.

Κλάση δημιουργείται όταν εκτελεσθεί η

```
   D<char> x( 'x', 7 );
```

όπου ζητείται κλάση-στιγμιότυπο του *D* για τιμή παραμέτρου "`char`". Πριν από αυτήν θα πρέπει να δημιουργηθεί η `B<char>` την οποία θα κληρονομήσει η `D<char>`.

Ας δούμε και ένα πολύ απλό παράδειγμα από την STL. Για να μπορούμε να περάσουμε μια συνάρτηση (κατηγόρημα) θα πρέπει να τη «μετατρέψουμε» σε κλάση· για την ακρίβεια σε συναρτησοειδές (§22.6.2). Η C++ έχει έτοιμα μερικά περιγράμματα τέτοιων συναρτησοειδών και μπορείς να τα χρησιμοποιήσεις βάζοντας στο πρόγραμμά σου "`#include <functional>`". Εκεί μπορείς να βρεις τα εξής περιγράμματα (μεταξύ άλλων):

```
template < typename Arg1, typename Arg2, typename Result >
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};


template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Το *binary_function* είναι ένα «άδειο» περίγραμμα ενώ το δεύτερο είναι περίγραμμα συναρτησοειδούς. Μέσα στο πρόγραμμά σου μπορείς να γράψεις:

```
#include <functional>
// . . .
using namespace std;
// . . .
   less<int> lt;
```

Η "`less<int>`" είναι κλάση (συναρτησοειδές) και η "`lt(1,-1)`" θα δώσει τιμή "`false`".

Δες και μιαν άλλην περίπτωση:

```
template < typename C1, typename C2 >
class D1 :  public B< C1 >
{
public:
   D1( C1 a1=0, C2 a2=0 )
      :  B<C1>( a1 ), md( a2 ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B<C1>::display( tout );  tout << "  " << md;  }
private:
   C2 md;
}; // D1
```

Εδώ, η πρώτη παράμετρος (*C1*) καθορίζει το στιγμιότυπο της βασικής το οποίο θα κληρονομήσει η παράγωγη ενώ η δεύτερη (*C2*) αναφέρεται μόνον στην παράγωγη.

Τέλος, θα αναφέρουμε δύο πιθανές δυσάρεστες εκπλήξεις που σε περιμενουν στην περίπτωση κληρονομιάς περιγράμματος από περίγραμμα.

1. Ας πούμε ότι στο «βασικό περίγραμμα» *B* έχεις μια συνάρτηση-μέλος

```
template < typename T >
class B
{
public:
   void f() {  /* . . . */  }
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* καλεί την *f()*:

```
   void g() { /* . . . */ f(); /* . . . */ }
```

Είναι πιθανό ότι ο μεταγλωττιστής σου (π.χ. gcc) δεν θα το επιτρέψει.

Πώς θα τον πείσουμε να το δεχτεί; Γράφοντας την κλήση της *f()* ως "`this->f()`" ή "`B<C>::f()`".

2. Ας πούμε ότι στο «βασικό περίγραμμα» *B* ορίζεις έναν τοπικό δικό σου τύπο, π.χ. μια κλάση:

```
template < typename T >
class B
{
public:
   struct Qaz {  /* . . . */  };
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ σε κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* δηλώνεις μια μεταβλητή τύπου *Qaz*:

```
   void g() { Qaz x; /* . . . */ }
```

Και αυτό μπορεί να απορριφθεί από τον μεταγλωττιστή σου.

Πώς διορθώνεται; Έτσι:

```
   void g() { typename B<C>::Qaz x; /* . . . */ }
```

Για περισσότερα παραπέμπουμε στο (Cline 1999), §35.18 και 35.19.

25.7 Περιέχουσες Κλάσεις

Περιέχουσα κλάση (container class) είναι *περίγραμμα* για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων. **Περιέχον** (container) είναι ένα αντικείμενο στιγμιοτύπου περιέχουσας κλάσης. Παραδείγματα:

- Η κλάση *SList* που είναι συλλογή αντικειμένων κλάσης *GrElmn* (§21.11) θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.

- Οι κλάσεις *CourseCollection*, *StudentCollection* και *StudentInCourseCollection* –που είδαμε στα Project 4 και 6 και είναι συλλογές αντικειμένων κλάσης *Course*, *Student* και *StudentInCourse* αντιστοίχως– θα μπορούσαν να είναι στιγμιότυπα μιας τέτοιας κλάσης.[6]
- Όλοι οι πίνακες που έχουμε δει στα παραδείγματά μας είναι συλλογές αντικειμένων του ίδιου τύπου. Βέβαια είναι λιγο δύσκολο να τους δούμε ως στιγμιότυπα ενός περιγράμματος κλάσης.

Τώρα θα πεις: Ένα αντικείμενο κλάσης *CourseCollection* περιέχει πίνακα με (βέλη προς) αντικείμενα κλάσης *Course*. Τι έχουμε εδώ; Περιέχον μέσα σε περιέχον; Όχι! Το περιέχον είναι ο πίνακας. Το αντικείμενο που τον περιέχει είναι το εργαλείο μας για να κάνουμε τον χειρισμό του πίνακα σύμφωνα με ορισμένους κανόνες (που προκύπτουν από την αναλλοίωτη). Λέμε ότι η κλάση *CourseCollection* (ακριβέστερα: το περίγραμμα κλάσης του οποίου στιγμιότυπο θα ήταν η *CourseCollection*) είναι **προσαρμογέας περιέχοντος** (container adaptor) ή **περιέχον-προσαρμογέας**. Αυτά ξεκαθαρίζουν στο παρακάτω

Παράδειγμα – Μια στοίβα σε πίνακα

Με τις

```
K   arr[sz];
int top;
```

και με δύο συναρτήσεις –*push()* και *pop()*– μπορούμε να υλοποιήσουμε μια **στοίβα** ((LIFO) stack) με στοιχεία τύπου *K*. Σε κάθε στιγμιότυπο όπου καθορίζεται ο τύπος *K* (π.χ. `int` ή *Date* ή *Course* κλπ) ο πίνακας είναι μια συλλογή αντικειμένων αυτού του τύπου. Έχουμε λοιπόν ένα περιέχον.

Αλλά το περιέχον μας έχει πρόβλημα: Πολλά μπορούμε να κάνουμε με τα στοιχεία ενός πίνακα! Έτσι:

- Σε μια στοίβα η θέση της κορυφής (*top*) μπορεί να μεταβάλλεται μόνον από τις *push()* και *pop()* κατά "`+1`" ή "`-1`" αντιστοίχως. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`top = 47`" ή "`top += 19`".
- Ακόμη, σε μια στοίβα μπορούμε να «δούμε» μόνο το στοιχείο που δείχνει η κορυφή. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`arr[47] = ...`" ή "`q = arr[29]`".

Για να εμποδίσουμε τέτοια λάθη «κρύβουμε» τα `arr` και `top` μέσα σε έναν προσαρμογέα περιέχοντος:

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
// . . .
   void push( const K& v );
   void pop();
   const K& getTop() const;
// . . .
private:
   K   arr[sz];
   int top;
}; // StackT
```

Ολόκληρο το περίγραμμα μπορεί να είναι:[7]

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
   StackT() { top = -1; }  // empty stack
```

[6] Θα το δούμε σε επόμενα Project.

[7] Το κατηγόρημα για την άδεια στοίβα θα το βαφτίζαμε *isEmpty()*, αλλά προτιμούμε να ακολουθήσουμε την ονοματολογία της STL.

```
   StackT( const StackT& aSt );
   void push( const K& v );
   void pop();
   const K& getTop() const;
   bool empty() const {  return ( top == -1 );  }
   bool isFull() const {  return ( top == sz-1 );  }
private:
   K   arr[sz];
   int top;
}; // StackT

struct StackTXptn :  public exception
{
   static const int fnLength = 100;
   enum { stackFull, stackEmpty };
   char funcName[fnLength];
   int  errorCode;

   StackTXptn( const char* fn, int ec ) :  errorCode(ec)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
}; // StackTXptn

template < typename K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   if ( top == sz-1 )
      throw StackTXptn( "push", StackTXptn::stackFull );
   ++top;  arr[top] = v;
} // push

template < typename K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
} // pop

template < typename K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( top == -1 )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return arr[top];
} // getTop
```

Το περίγραμμα είναι πολύ απλό, αλλά όχι και τόσο χρήσιμο. Πρόβλημα; Το σταθερό μέγεθος του πίνακα. Μπορούμε να τον κάνουμε δυναμικό και να χρησιμοποιήσουμε τη *renew()*. Αλλά, αν πρέπει να κάνουμε κάθε τόσο αντιγραφές ολόκληρου του πίνακα οι καθυστερήσεις θα είναι μεγάλες.

Να και ένα προγραμματάκι που δοκιμάζει την παραπάνω υλοποίηση:

```
int main()
{
   StackT<int> intStack;

   cout << "Pushing integers on intStack" << endl;
   for ( int i = 0; i < 5; i++ )
   {
      intStack.push( i );       // put items in the stack
      cout << i << ' ';
   }
   cout << endl;

   cout << endl << "Popping integers from intStack" << endl;
   while ( !intStack.empty() )
   {
```

```
      cout << intStack.getTop() << ' ';
      intStack.pop();                    // remove items
   }
   cout << endl;

   StackT<char> charStack;
   cout << endl << endl
        << "Pushing characters on charStack" << endl;
   for ( char c = 'A'; c < 'E'; c++ )
   {
      charStack.push( c );      // put items in the stack
      cout << c << ' ';
   }
   cout << endl;

   cout << endl << "Popping characters from charStack" << endl;
   while ( !charStack.empty() )
   {
      cout << charStack.getTop() << ' ';
      charStack.pop();   // remove items
   }
   cout << endl;
} // main
```

Αυτή η στοίβα ήταν το «ορεκτικό». Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.

Παρατήρηση: ▶

Όλος ο κόσμος θα έγραφε ένα περίγραμμα σαν

```
template < typename K, size_type sz >
const K& StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
   return arr[top+1];
} // pop
```

Γιατί εδώ γράψαμε δύο χωριστές μεθόδους-περιγράμματα; Για ασφάλεια στις εξαιρέσεις!

Ας εξηγηθούμε: έστω ότι η *v* είναι μεταβλητή κλάσης *K*, το *aStack* είναι αντικείμενο κλάσης `StackT<K>` και ρίχνεται εξαίρεση κατά την εκτέλεση της:

```
v = aStack.pop();
```

Έστω ότι η εκχώρηση στην κλάση *K* έχει ισχυρή εγγύηση ασφάλειας πράγμα που σημαίνει ότι η τιμή της *v* δεν θα αλλάξει. Αλλά τι γίνεται με το *aStack*; Η τιμή που αφαιρέθηκε από την κορυφή του χάθηκε!

Με την υλοποίηση που δώσαμε η ίδια δουλειά θα γίνει με τις:

```
v = aStack.getTop();
aStack.pop();
```

Αν κατά την εκτέλεση της πρώτης εντολής ριχτεί εξαίρεση η δεύτερη εντολή δεν θα εκτελεσθεί.

Πάντως αν το πρόγραμμά σου έχει σχεδιαστεί για πολυνηματική εκτέλεση και δύο ή περισσότερα νήματα παίρνουν τιμές από το *aStack* υπάρχει άλλο πρόβλημα. Οι δύο λειτουργίες θα πρέπει να γίνονται από *μια* συνάρτηση που θα πρέπει να έχει και **ατομικότητα** (atomicity), που με απλά λόγια σημαίνει ότι εκτελείται σαν *μια εντολή* (πριν τελειώσει η εκτέλεσή της δεν μπορεί να αλλάξει νήμα εκτέλεσης). Η C++11, αλλά και οι API των ΛΣ, σου δίνουν εργαλεία για να λύσεις παρόμοια προβλήματα.◀

25.7.1 Το Περίγραμμα μιας Λίστας

Πιο πάνω είπαμε «*Η κλάση SList, που είναι συλλογή αντικειμένων κλάσης GrElmn (§21.11), θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.*» Ας το δούμε λοιπόν. Θα ξεκινήσουμε από την τελευταία μορφή της *SList*, όπως την είδαμε στην §21.11 και στην §22.9 όπου κάναμε εισαγωγή των προσεγγιστών.

Όπως στη *BString* αντικαταστήσαμε το "`char`" με "*K*" εδώ θα αντικαταστήσουμε το "*GrElmn*" με "*K*". Πριν από αυτό όμως χρειάζονται μερικές αλλαγές. Οι μέθοδοι *find1Elmn()*, *get1Elmn()* και *delete1Elmn()* έχουν μια παράμετρο τύπου `int` αφού περιμένουν εκεί ένα όρισμα, τον ατομικό αριθμό, που είναι και το κλειδί για τη *GrElmn*. Αν τις αφήσουμε όπως είναι το περίγραμμα θα μπορεί να δώσει μόνον στιγμιότυπα-συλλογές αντικειμένων με κλειδί κάποιον ακέραιο. Φυσικά δεν θέλουμε κάτι τέτοιο και τις αλλάζουμε σε:

```
bool SList::find1Elmn( const GrElmn& aDtItem ) const
{
   return ( findPtr(aDtItem) != slTail );
} // SList::find1Elmn

const GrElmn& SList::get1Elmn( const GrElmn& aDtItem ) const
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::get1Elmn

void SList::delete1Elmn( const GrElmn& aDtItem )
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::delete1Elmn
```

Στη συνέχεια θα ξαναδούμε αυτές τις μεθόδους.

Παρατήρηση: ▶

Τουλάχιστον η *get1Elmn()* φαίνεται κάπως περίεργη: Ζητούμε να μας επιστρέψει κάτι που το δίνουμε ως όρισμα όταν την καλούμε; Όχι βέβαια. Η κλήση της θα γίνεται συνήθως ως εξής:

```
tmp = lst.get1Elmn( GrElmn(aa) );
```

Δηλαδή στο *aDtItem* θα περνάει ένα αντικείμενο που θα έχει μόνο το κλειδί. Για τα υπόλοιπα θα στηριζόμαστε στον ορισμό της ισότητας (`operator==()`) που έχουμε για τη *GrElmn*.◀

Κάνοντας τώρα την αντικατάσταση του "*GrElmn*" με "*K*" έχουμε:

```
template < typename K >
class SListT
{
private:
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      K& operator*();
      K* operator->();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
```

```
      ListNode* iPNode;
   }; // Iterator

   SListT();
   SListT( const SListT& rhs );
   ~SListT();
   SListT& operator=( const SListT& rhs );
   void swap( SListT& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
   bool find1Elmn( const K& aDtItem ) const;
   const K& get1Elmn( const K& aDtItem ) const;
   void insert1Elmn( const K& aDtItem );
   void delete1Elmn( const K& aDtItem );
   void save( ostream& bout,
              void (*wrProc)(const K&, ostream&) ) const;
   void display( ostream& tout ) const;
private:
   ListNode* slHead;
   ListNode* slTail;

   ListNode* findPtr( const K& aDtItem ) const;
   void push_front( const K& aData );
}; // SList
```

και η αντίστοιχη κλάση εξαιρέσεων:

```
struct SListTXptn
{
   enum { allocFailed, notFound, listEnd };
   char  funcName[100];
   int   errorCode;
   int   errIntVal;
   SListTXptn( const char* fn, int ec, int iv=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errIntVal = iv;  }
}; // SListXptn
```

Ας δούμε τα περιγράμματα συναρτήσεων που επιφορτώνουν τους τελεστές "++", "*" και "->" για τον *SListT*<*K*>::*Iterator*:

```
template < typename K >
SListT<K>::Iterator& SListT<K>::Iterator::operator++()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator++",
                        SListTXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SListT<K>::Iterator::operator++

template < typename K >
K& SListT<K>::Iterator::operator*()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator*",
                        SListTXptn::listEnd );
   return ( iPNode->lnData );
} // SListT<K>::Iterator::operator*

template < typename K >
K* SListT<K>::Iterator::operator->() const
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator->",
                       SListXptn::listEnd );
   return ( &(iPNode->lnData) );
} // SList::Iterator::operator->
```

Στη συνέχεια θα δούμε μερικές από τις μεθόδους του περιγράμματος ξεκινώντας με την

```
template < typename K >
void SListT<K>::push_front( const K& aData )
{
   ListNode* pnln;
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListTXptn( "push_front", SListTXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = slHead;
   slHead = pnln;
} // SList::push_front
```

Εδώ τώρα πρόσεξε τα εξής:

- Για να εκτελεσθεί η "`new ListNode`" θα πρέπει να δημιουργηθεί και το μέλος *lnData* από τον ερήμην δημιουργό της *K*.
- Εκτέλεση της "`pnln->lnData = aData`" σημαίνει εκτέλεση του (αντιγραφικού) τελεστη εκχώρησης της *K*.

Επομένως:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***

Βέβαια θα πεις ότι θα μπορούσαμε να αποφύγουμε τον ερήμην δημιουργό και τον τελεστή εκχώρησης αν εφοδιάσουμε την *ListNode* με δημιουργούς:

```
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
      ListNode()
         : lnNext( 0 ) { };
      explicit ListNode( const K& aData )
         : lnData( aData ), lnNext( 0 ) { };
   }; // ListNode
```

και αντί για:

```
   pnln = new ListNode;  pnln->lnData = aData;
```

να γράφουμε

```
   pnln = new ListNode( aData );
```

Εδώ καλείται ο δεύτερος δημιουργός που –με τη σειρά του– θα καλέσει τον δημιουργό αντιγραφής (`lnData(aData)`) της *K*.

Σωστό! Αλλά πρόσεξε το εξής: Εμείς κάνουμε αυτές τις επισημάνσεις για να κατάλάβεις τις απαιτήσεις που (θα δούμε ότι) βάζει η STL για τις κλάσεις που πιθανόν θα θελήσεις να βάλεις σε κάποιο από τα περιέχοντά της. Εκεί δεν θα έχεις επιλογή για το πώς θα γράψεις το περιέχον· είναι έτοιμο. Έτσι, η παρατήρηση που κάναμε πιο πάνω όχι μόνον δεν αναιρεί τους περιορισμούς που είχαμε αλλά βάζει έναν ακόμη:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***

Ας σκεφτούμε τώρα λιγάκι τη *findPtr()*. Κατ' αρχήν θα μπορούσαμε να γράψουμε το περίγραμμα:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   slTail->lnData = aDtItem;
   fv = slHead;
   while ( (fv->lnData) != aDtItem )  fv = fv->lnNext;
   return fv;
} // SListT<K>::findPtr
```

αλλά εκείνη η "`slTail->lnData = aDtItem`" –αντιγραφή ολόκληρου αντικειμενου κλάσης *K*–[8] έχει πρόβλημα. Αν μπορούμε να θεωρήσουμε ότι ένα αντικείμενο *GrElmn* είναι μικρό και η αντιγραφή του δεν είναι χρονοβόρα δεν μπορούμε να πούμε το ίδιο πράγμα για το αντικείμενο οποιασδήποτε κλάσης *K* θα κληθεί να φιλοξενήσει το *SListT*.

Εγκαταλείπουμε λοιπόν την αντιγραφή στον φρουρό και επιστρέφουμε στις διπλές συγκρίσεις:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   for ( fv = slHead; fv != slTail; fv = fv->lnNext )
      if ( fv->lnData == aDtItem ) break;
   return fv;
} // SListT<K>::findPtr
```

Αυτό το περίγραμμα όμως είναι σαν να μας λέει: «Άλλαξε τα βέλη, σε προσεγγιστές!» Ας το κάνουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::findIt( const K& aDtItem ) const
{
   Iterator fv;
   for ( fv = begin(); fv != end(); ++fv )
      if ( *fv == aDtItem ) break;
   return fv;
} // SListT<K>::findIt
```

Αν χρησιμοποιήσουμε τη νέα *findIt()* στη *get1Elmn()* θα έχουμε:

```
template < typename K >
const K& SListT<K>::get1Elmn( const K& aDtItem ) const
{
   Iterator itToNode( findIt(aDtItem) );
   if ( itToNode == end() )
      throw SListTXptn( "get1Elmn", SListTXptn::notFound,
                        aDtItem.getANumber() );
   return *itToNode;
} // SList<K>::get1Elmn
```

Εδώ όμως η *findIt()* δεν μας χρειάζεται· μπορούμε να χρησιμοποιήσουμε την *std*::*find()* –που είδαμε στην §22.9– και να γράψουμε:

```
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
```

Μπορείς να τροποποιήσεις με τον ίδιο τρόπο και τις *insert1Elmn()* και *delete1Elmn()*. Για την τελευταία θα χρειαστείς κάτι ακόμη: Στην περιοχή "`private`" της *Iterator* ορίζουμε:

```
// . . .
   private:
      ListNode* iPNode;
      ListNode* getPNode() const {  return iPNode;  }
   }; // Iterator
```

Έτσι, στην αρχή της *delete1Elmn()* θα έχουμε:

```
template < typename K >
void SListT<K>::delete1Elmn( const K& aDtItem )
{
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
   if ( itToNode != end() ) // υπάρχει
   {
      ListNode* ptrToNode( itToNode.getPNode() );
      ListNode* pnln( ptrToNode->lnNext );
   // . . .
```

[8] Ενώ χρειαζόμαστε μόνο το κλειδί!

Παρατήρηση:▶
Σχετικώς με τον φρουρό: οι αντιγραφές δεν είναι το μόνο πρόβλημα. Αν τα αντικείμενα της περιεχόμενης κλάσης είναι μεγάλα η ύπαρξη του φρουρού είναι και σπατάλη μνήμης.◀

25.7.1.1 Ο Καταστροφέας

Τέλος, ας πούμε και δυο λόγια για τον καταστροφέα. Το μόνο πράγμα που άλλαζει είναι η επικεφαλίδα από "`SList::~SList()`" σε "`template <typename K> SListT<K>::~SListT()`". Υπάρχει όμως ένα άλλο ερώτημα σχετικώς με τη δήλωση: Μήπως θα έπρεπε να τον δηλώσουμε "`virtual`" ώστε να υπάρχει δυνατότητα να κληρονομηθούν οι κλάσεις-στιγμιότυπα του περιγράμματος; Αυτό είναι ένα ερώτημα που τίθεται από πολλούς για τα περιέχοντα της SLT και μάλιστα με κάπως διαφορετικό τρόπο: πώς «ξέφυγε» μια τόσο σοβαρή παράλειψη και οι καταστροφείς των περιεχόντων της STL δεν είναι "`virtual`";

Φυσικά δεν πρόκειται για λάθος ούτε για παράλειψη και ας το σκεφτούμε. Στη συνέχεια θα δούμε υλοποίηση στοίβας με μια λίστα. Για να δηλώσουμε μια στοίβα ακεραίων μπορούμε να κάνουμε ένα από τα εξής:

- Να ορίσουμε το περίγραμμα:

```
template < typename K >
class StackT
{
public:
// . . .
private:
   SListT<K> cont;
}; // StackT
```

και στη συνέχεια να δηλώσουμε:

```
StackT< int > aStack;
```

- Να ορίσουμε:

```
class IntStack : public SListT<int>  { /*. . . */ }
```

και στη συνέχεια:

```
IntStack aStack;
```

Τι θα πρέπει να προτιμήσουμε; Για να απαντήσουμε θα πρέπει να καταλάβουμε το νόημα της κάθε επιλογής:

- Στην πρώτη περίπτωση έχουμε το περίγραμμα όπως το είχαμε αρχικώς (εκτός από τη δεύτερη παράμετρο "`size_type sz = 100`") όπου αλλάξαμε το περιέχον από πίνακα σε λίστα.
- Στη δεύτερη περίπτωση λέμε ότι μια στοίβα ακεραίων είναι μια (`is_a`) λίστα ακεραίων.

Η δεύτερη επιλογή έρχεται σε αντίθεση με αυτό που λέγαμε στην §23.14: «*οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και τη σωστή νοηματική σχέση.*» Η στοίβα *μπορεί να υλοποιηθεί* με μια λίστα. Αλλά *δεν είναι λίστα*· ήδη την έχουμε υλοποιήσει με πίνακα.[9]

Στις περισσότερες περιπτώσεις που θα πάει το μυαλό σου σε κληρονομιά από περιέχον ένα σκεπτικό σαν το παραπάνω έχει νόημα. Δεν βάζουμε λοιπόν το "`virtual`" για να αποθαρρύνουμε τέτοιες σκέψεις.

Δηλαδή η κληρονομιά από το περίγραμμα λίστα δεν έχει νόημα ποτέ; Θα είχε νόημα αν κάναμε μια νέα δομή δεδομένων που θα ήταν ειδική περίπτωση της λίστας με απλή σύνδεση.

---

[9] Με την ορολογία της §23.15: αυτά είναι επιχειρήματα υπέρ της σχέσης "`has_a`"· επομένως και υπέρ της κληρονομιάς "`private`".

25.7.2 Η Περιεχόμενη Κλάση

Να καταγράψουμε απολογιστικά τις απαιτήσεις για την περιεχόμενη κλάση που βρήκαμε από το παράδειγμά μας μέχρι τώρα:

- ***Πρώτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Δεύτερη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει δημιουργό αντιγραφής.***
- ***Τρίτη Απαίτηση: Για τον περιεχόμενο τύπο K πρέπει να έχει ορισθεί ο τελεστής εκχώρησης.***

Φυσικά, χωρίς να καταγράψουμε, χρησιμοποιήσαμε το γεγονός ότι:

- ***Τέταρτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

Αυτές οι απαιτήσεις τίθενται για όλα τα περιέχοντα της STL. Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Αν, ας πούμε, θέλεις να κάνεις αναζητήσεις τιμών θα πρέπει να έχεις τη δυνατότητα για συγκρίσεις.

25.7.3 Ένα Πρόβλημα, μια Λύση και ένα Άλλο Πρόβλημα

Το περίγραμμα *slist*[10] (αλλά και το *list* της STL) έχει μέθοδο:

```
iterator erase( iterator pos )
```

Αν *a* είναι αντικείμενο κλάσης *slist*<*K*> και *p* ένας *slist*<*K*>::*iterator* τότε η εντολή

```
a.erase( p );
```

καταστρέφει τον κόμβο-στόχο του *p*. Το μήκος της λίστας μειώνεται κατά 1 χωρίς να αλλάζουν οι σχετικές θέσεις των άλλων κόμβων. Η μέθοδος επιστρέφει προσεγγιστή με στόχο το στοιχείο της λίστας που ακολουθεί αυτό που διαγράφηκε.

Αν θέλουμε να γράψουμε και εμείς μια μέθοδο

```
Iterator erase( Iterator pos );
```

για το *SlistT* θα αντιμετωπίσουμε το εξής πρόβλημα: «*θα πρέπει να αλλάξουμε την τιμή του lnNext του προηγούμενου κόμβου που δεν ξέρουμε ποιος είναι!*» Στην §21.11 δώσαμε μια λύση –που φαίνεται στο Σχ. 21-1– γράφοντας την *SList*::*erase1Elmn*. Στην περίπτωσή μας θα έχουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::erase( SListT<K>::Iterator pos )
{
   if ( pos == end() )
      throw SListTXptn( "erase", SListTXptn::listEnd );
   ListNode* ptrToNode( pos.getPNode() );
   ListNode* pnln( ptrToNode->lnNext );
   ptrToNode->lnNext = pnln->lnNext;
   if ( pnln == slTail ) slTail = ptrToNode;
                    else ptrToNode->lnData = pnln->lnData;
   delete pnln;
   return pos;
} // SListT<K>::erase
```

Όπως βλέπεις, θα χρειαστούμε μια μέθοδο (`private`) της κλάσης *Iterator* που θα μας δίνει το βέλος που κρύβει ο προσεγγιστής:

```
      ListNode* getPNode() const {  return iPNode;  }
```

Φαίνεται λοιπόν ότι λύσαμε το πρόβλημά μας αλλά πρόσεξε τα προβλήματα της παραπάνω λύσης:

---

[10] της SGI. Στο C++11 προδιαγράφεται περιέχουσα κλάση με το όνομα "`forward_list`" και είναι λίστα με απλή σύνδεση. Η *forward_list* δεν έχει *erase()* με τα παραπάνω χαρακτηριστικά. Έχει μέθοδο *erase_after()* που καταστρέφει τον κόμβο που ακολουθεί τον κόμβο-στόχο.

- Που δείχνει ο `pos`; Στον κόμβο που έδειχνε αλλά που τώρα έχει ως περιεχόμενο αυτό του κόμβου που ακολουθούσε αυτόν που διαγράψαμε! Δηλαδή το `*pos` αλλάζει με τη διαγραφή. Ναι, αλλά ο `pos` είναι παράμετρος τιμής. Αυτή είναι μια περίεργη κατάσταση.
  Αν χρησιμοποιήσεις άλλη μέθοδο και ανακυκλώσεις ακριβώς τον κόμβο-στόχο του `pos` τότε τα πράγματα είναι χειρότερα.
- Αν δεν έχουμε φτάσει στον φρουρό κάνουμε μια αντιγραφή παραπάνω: "`ptrToNode->lnData = pnln->lnData`". Αν τα αντικείμενα του *K* είναι μεγάλα αυτή η αντιγραφή κοστίζει.

Το δεύτερο πρόβλημα μπορείς να το λύσεις –με άλλον αλγόριθμο– αλλά το πρώτο δεν λύνεται. Η διαγραφή κόμβου *ακυρώνει* τον προσεγγιστή. Αυτό είναι ένα γενικότερο πρόβλημα με τις περιέχουσες κλάσεις: *οι διαγραφές (αλλά και οι εισαγωγές) τιμών μπορεί να ακυρώνουν κάποιους προσεγγιστές*.

25.7.4 Μια Άλλη Στοίβα

Υποσχεθήκαμε ότι «*Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.*» Θα την υλοποιήσουμε χρησιμοποιώντας –όχι πίνακα– αλλά λίστα με απλή σύνδεση.

Στην προηγούμενη υποπαράγραφο είδαμε ένα «άδειο» σχέδιο του περιγράμματος (*StackT*) που θα γράψουμε. Το μοναδικό κρυμμένο μέλος είναι:

```
   SListT<K> cont;
```

Ο ερήμην δημιουργός είναι απλούστατος:

```
   StackT() { };
```

αφού μόνη η δήλωση της λίστας δημιουργεί μια κενή στοίβα.

Ο δημιουργός αντιγραφής

```
   StackT( const StackT& aSt );
```

ορίζεται ως:

```
template < class K, size_type sz >
StackT<K,sz>::StackT( const StackT<K,sz>& other )
   :  cont( other.cont )  {  }
```

και έχει λυμένα τα προβλήματά του από τον δημιουργό αντιγραφής της λίστας.

Ο (σωστός) τελεστής εκχώρησης της λίστας μας επιτρέπει να μην γράψουμε τελεστή εκχώρησης για τη στοίβα.

Η *push()* ορίζεται ως εξής:

```
template < class K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   try {  cont.push_front( v );  }
   catch( SListTXptn& x )
   {  throw StackTXptn( "push", StackTXptn::stackFull );  }
} // StackT<K,sz>::push
```

αλλά εδώ έχουμε ένα πρόβλημα: Η *SListT*::*push_front()* είναι δηλωμένη "`private`"· θα πρέπει να την κάνουμε "`public`".

Για να ορίσουμε την

```
template < class K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( cont.empty() )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   cont.pop_front();
} // StackT<K,sz>::pop
```

θα πρέπει να ορίσουμε την *SListT*::*pop_front()*:

```
template < typename K >
void SListT<K>::pop_front()
{
   if ( slHead == slTail )
      throw SListTXptn( "pop_front", SListTXptn::listEmpty );
   ListNode* pnln( slHead );
   slHead = slHead->lnNext;
   delete pnln;
} // SListT<K>::pop_front
```

Τώρα, η *getTop()* ορίζεται ως εξής:

```
template < class K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( cont.empty() )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return *cont.begin();
} // StackT<K,sz>::getTop
```

Τέλος, τα δύο κατηγορήματα ορίζονται `inline` ως εξής:

```
   bool isEmpty() const {  return cont.empty();  }
   bool isFull() const {  return false;  }
```

Το δεύτερο είναι λίγο περίεργο; Όχι και τόσο αφού για να γεμίσει η στοίβα θα πρέπει να πάρουμε όλη τη διαθέσιμη μνήμη του υπολογιστή μας. Και τότε, αν αποπειραθούμε να καλέσουμε την *push()* θα πάρουμε εξαίρεση με κωδικό *StackTXptn*::*stackFull* που θα προκληθεί από τη σύλληψη εξαίρεσης με κωδικό *SListTXptn*::*allocFailed* που θα ρίξει η *push_front()*.

Δηλαδή όλες οι αλλαγές είναι στην υλοποίηση και το τμήμα διεπαφής δεν αλλάζει; Ακριβώς! Και δεν θα πρέπει να ξεφορτωθούμε τη δεύτερη παράμετρο ("`size_type sz = 100`") καθώς και την *isFull()*; Μπορούμε να το κάνουμε και έτσι· σε «πραγματικές» εφαρμογές όμως θα πρέπει να σκεφτείς το εξής: αν θέλουμε να περάσουμε στην καινούρια υλοποίηση, θα αλλάξουμε όλα τα προγράμματα που έχουμε γράψει και χρησιμοποιούν το περίγραμμα της στοίβας; Η απάντηση στο ερώτημα αυτό είναι συνήθως «όχι». Βέβαια, σε κάποιο πρόγραμμα που καλεί την *isFull()* μπορεί να συμβεί το εξής: οι

```
   if ( !aStack.isFull() )
      aStack.push( v );
```

μπορεί να προκαλέσουν έγερση εξαίρεσης *StackTXptn* με κωδικό *stackFull*! Ε, συμβαίνουν αυτά όταν κάνουμε βελτιώσεις…

Αλλαγές όμως θα έχουμε στο περίγραμμα *SListT*:

- Η *push_front()* έγινε "`public`" και
- Γράψαμε άλλη μια μέθοδο, την *pop_front()* που είναι και αυτή "`public`".

Αυτές μάλλον δεν προκαλούν προβλήματα σε εφαρμογές που χρησιμοποιούν το περίγραμμα.[11]

Παρατήρηση:▶

Για το πρόγραμμα του παραδείγματος της §25.7 η υλοποίηση με τον πίνακα είναι μάλλον πιο κατάλληλη. Για να το καταλάβεις υπολόγισε τη (σπατάλη σε) μνήμη αν υλοποιήσουμε τις δύο στοίβες με λίστα.◀

25.8 Έξυπνα Βέλη

Στην §16.7 εντοπίσαμε ορισμένα προβλήματα που μπορεί να έχουμε όταν διαχειριζόμαστε δυναμική μνήμη με μεταβλητές-βέλη.

---

[11] Αν κάποιος έγραψε κλάση ή περίγραμμα που να κληρονομεί το *SListT*… Εδώ μπορεί να υπάρχει πρόβλημα.

- **Διαρροή μνήμης** (memory leakage): «*το πρόγραμμά μας έχει δυναμική(-ές) μεταβλητή(-ές) αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί (ούτε να την ανακυκλώσει).*»
- **Μετέωρο βέλος** (pending ή dangling pointer): αν δύο ή περισσότερα βέλη δείχνουν την ίδια δυναμική μεταβλητή και την ανακυκλώσουμε τότε κάποιο(-α) βέλος(-η) μένει(-ουν) *μετέωρο*(-α) με την έννοια ότι δείχνει θέση της μνήμης που δεν ελέγχεται από το πρόγραμμά μας.
- **Πολλαπλή διαγραφή (ανακύκλωση):** Αν το βέλος *p* δείχνει κάποια περιοχή της δυναμικής μνήμης οι εντολές "`delete p; delete p;`" μπορεί να προκαλέσουν καταστροφικό λάθος στο σύστημα διαχείρισης της δυναμικής μνήμης.

Τα **έξυπνα βέλη** (smart pointers) είναι προγραμματιστικά εργαλεία που χρησιμοποιούμε για να ξεπεράσουμε τα προβλήματα που απαριθμήσαμε παραπάνω.

Παράδειγμα έξυπνου βέλους μας δίνει η STL της C++. Αναφερόμαστε στο περίγραμμα κλάσης *auto_ptr*. Αυτό το περίγραμμα κλάσης θα μελετήσουμε εδώ χωρίς να δεσμευόμαστε από κάποια συγκεκριμένη υλοποίηση. Για να μην υπάρχουν μπερδέματα, θα ονομάσουμε το δικό μας περίγραμμα *AutoPtr*.[12]

Η ιδέα είναι να κρύψουμε το βέλος μέσα σε ένα αντικείμενο που θα το κάνουμε να συμπεριφέρεται σαν βέλος: Μετά τις

```
Date*           pd( new Date(2010, 11, 26) );
AutoPtr< Date >  apd( pd );
```

το "`*apd`" και το "`*pd`" θα πρέπει να είναι το ίδιο πράγμα. Το ίδιο ισχύει και για τα "`apd->getYear()`", "`apd->getMonth()`", "`apd->getDay()`" και τα "`pd->getYear()`", "`pd->getMonth()`", "`pd->getDay()`" αντιστοίχως.

Κατά τα άλλα θα ρυθμίσουμε τη συμπεριφορά του *AutoPtr*<*K*> για να αντιμετωπίσουμε τα προβλήματα που παραθέσαμε παραπάνω.

Αυτό το κάναμε ήδη μια φορά με τους προσεγγιστές. Εκεί είδαμε ότι για να έχει το αντικείμενο συμπεριφορά βέλους θα πρέπει η (παραμετρική) κλάση μας να είναι:

```
template < typename K >
class Iterator
{
public:
   explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
   K& operator*() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListTXptn( "Iterator::operator*",
                           SListTXptn::listEnd );
      return ( iPNode->lnData );
   } // operator*
   K* operator->() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListXptn( "Iterator::operator->",
                          SListXptn::listEnd );
      return ( &(iPNode->lnData) );
   } // SList::Iterator::operator->
private:
   ListNode* iPNode;
   ListNode* getPNode() const {  return iPNode;  }
}; // Iterator
```

Ξεκινούμε και εδώ το περίγραμμα κλάσης ως εξής:

---

[12] Στο (Alexandrescu 2001) εξετάζονται άλλες επιλογές που υπάρχουν για τη σχεδίαση μιας παραμετρικής κλάσης για έξυπνα βέλη. Μπορείς να βρεις το Κεφ. 7 (για τα έξυπνα βέλη) αυτού του βιβλίου στο `http://ptgmedia.pearsoncmg.com/images/9780201704310/samplepages/0201704315.pdf`.

```
template < typename K >
class AutoPtr
{
// . . .
private:
   K* ap;
}; // AutoPtr
```

Η διαφορά μας από τον προσεγγιστή είναι ότι εκεί το βέλος *iPNode* στόχευε έναν κόμβο μέσα στον οποίο υπάρχει το αντικείμενο που μας ενδιαφέρει ενώ τώρα στοχεύει το ίδιο το αντικείμενο.

Ο δημιουργός είναι ολόϊδιος:

```
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
```

Αλλά τώρα θα χρειστούμε –η πρώτη σημαντική διαφορά– και *καταστροφέα που θα ανακυκλώνει τον στόχο*:

```
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

Για την *getPNode()* υπάρχουν δυο διαφορές (με βάση αυτά που ισχύουν για το *auto_ptr*):

- Η πιο σημαντική διαφορά: είναι "`public`".
- Η λιγότερο σημαντική: θα της αλλάξουμε το όνομα σε "*get*".

```
   K* get() const {  return ap;  }
```

Στην επιφόρτωση του "`operator*`" και του "`operator->`" έχουμε πρόβλημα: Όλες οι μέθοδοι της *auto_ptr* έχουν προδιαγραφή εξαιρέσεων "`throw()`". Το C++03 δίνει τις εξής προδιαγραφές για την `operator*()` του *auto_ptr* που ακολουθούμε:

```
X& operator*() const throw();
```

Προϋπόθεση: `get() != 0`

Επιστρέφει: `*get()`

Για την `operator->()` μας λέει:

```
X* operator->() const throw();
```

Επιστρέφει: `get()`

Οι συναρτήσεις που γράψαμε για τον προσεγγιστή μπορεί να ρίξουν εξαίρεση! Εδώ τι θα κάνουμε; Θα ακολουθήσουμε το πρότυπο:

```
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
```

Έτσι, αν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα αντικείμενο *AutoPtr* που έχει *ap* `==` 0 θα έχεις –όταν εκτελείται η `operator*()`– το πρόβλημα που έχεις όταν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα σύνηθες μηδενικό βέλος. Στην περίπτωση του "`->`" η εκτέλεση της `operator->()` γίνεται κανονικώς και το πρόβλημα θα παρουσιαστεί όταν θα δράσει ο "`->`" στο αποτέλεσμα της συνάρτησης.

Ας πούμε ότι έχουμε δηλώσει:

```
   AutoPtr<string> aps( new string("abc") );
```

Οι εντολές

```
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;
```

θα δώσουν:

```
abc
qwerty
```

Μέχρι εδώ έχουμε:

```
template < typename K >
class AutoPtr
{
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

```
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
private:
   K* ap;
}; // AutoPtr
```

Τι έχουμε κερδίσει με αυτά; Αν έχουμε:

```
void f( int v )
{
   int* p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
   delete p;
} // f
```

παρ' όλο που φροντίσαμε να βάλουμε τη "`delete p`", αν ριχτεί εξαίρεση θα έχουμε διαρροή μνήμης. Αν όμως βάλουμε:

```
void f( int v )
{
   AutoPtr<int> p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
} // f
```

τα πράγματα αλλάζουν: Είτε η εκτέλεση της συνάρτησης τελειώσει κανονικώς είτε διακοπεί λόγω ρίψης εξαίρεσης η καταστροφή της *p* ανακυκλώνει και τη μνήμη που κατέχει η *p*. Όπως καταλαβαίνεις, δεν χρειαζόμαστε πια τον `delete`, αφού έχουμε αυτόματη ανακύκλωση.

Θα δούμε τώρα δύο χαρακτηριστικές μεθόδους της *AutoPtr*. Η πρώτη είναι η *release()*. Το πρότυπο μας λέει για τη *release()* της *auto_ptr*:

```
X* release() throw();
```

Επιστρέφει: `get()`

Απαίτηση: το αντικείμενο (`*this`) έχει το μηδενικό βέλος (`NULL`).

Δηλαδή: επιστρέφει την τιμή του «κρυμμένου» βέλους αλλά επιπλέον το μηδενίζει.

```
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
```

Ας δούμε τη διαφορά της από τη *get()* με ένα παράδειγμα. Μετά τη δήλωση

```
   auto_ptr< Date >  apd( new Date(2010, 11, 26) );
```

οι εντολές:

```
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
```

δίνουν:

```
26.11.2010
26.11.2010
26.11.2010
it is NULL
```

Η τελευταία γραμμή δείχνει ότι, μετά την *apd.release()*, το βέλος που κρύβεται μέσα στο *apd* μηδενίζεται. Βέβαια η τιμή του πέρασε στο *pd1*, όπως φαίνεται από την προτελευταία

γραμμή. Με τη "`Date* pd0(apd.get())`" αντιγράφεται η τιμή του βέλους του *apd* στο *pd0* χωρίς να αλλάζει, όπως δείχνουν οι δύο πρώτες γραμμές.

Η δεύτερη μέθοδος είναι η *reset()*:

```
void reset( X* p=0 ) throw();
```

Ενέργεια: Αν `get() != p` τότε `delete get()`.
Απαίτηση: το αντικείμενο (`*this`) έχει το βέλος *p*.
Με αυτήν αλλάζουμε το κρυμμένο βέλος. Αλλά προσοχή: *αν το παλιό βέλος έχει την κυριότητα κάποιου αντικειμένου αυτό ανακυκλώνεται πριν από την αλλαγή της τιμής του βέλους.*

```
   void reset( K* p=0 )
   {  if ( ap != p ) {  delete ap;  ap = p;  }  }
```

Πρόσεξε ότι αν δώσουμε:

```
   api1.reset();
```

ανακυκλώνεται το αντικείμενο που κατέχει το *api1*. Δηλαδή, μπορούμε έτσι να ανακυκλώνουμε τη μνήμη όποτε θέλουμε.

Ας δούμε τώρα τον «δημιουργό αντιγραφής» και τον τελεστή εκχώρησης.

Το C++03 μας λέει:

```
auto_ptr( auto_ptr& a ) throw();
```

Ενέργεια: Εκτελείται η κλήση "`a.release()`".
Απαίτηση: Το `*this` κρατάει το βέλος που επιστρέφει η `a.release()`.

Για το *AutoPtr* γράφουμε:

```
   AutoPtr( AutoPtr& other ) {  ap = other.release();  }
```

που ισοδυναμεί με:

```
   AutoPtr( AutoPtr& other ) {  ap = other.ap;  other.ap = 0;  }
```

Τι δημιουργός αντιγραφής είναι αυτός; Η μοναδική εντολή φέρνει στο *ap* την τιμή του *other.ap* αλλά, επί πλέον, μηδενίζει το *other.ap*· για να γίνεται αυτό *δεν βάλαμε* "`const`" στην παράμετρο. Στην πραγματικότητα εδώ έχουμε έναν **δημιουργό μεταβίβασης** (move constructor).[13]

Για τον τελεστή εκχώρησης οι προδιαγραφές είναι:

```
auto_ptr& operator=( auto_ptr& a ) throw();
```

Προϋπόθεση: Μπορεί να εκτελεσθεί η "`delete get()`".
Ενέργεια: `reset( a.release() )`.
Επιστρέφει: `*this`.

Για το *AutoPtr* μπορούμε να γράψουμε:

```
   AutoPtr& operator=( AutoPtr& rhs )
   {  reset( rhs.release() );  return *this;  }
```

που είναι ισοδύναμο με:

```
   AutoPtr& operator=( AutoPtr& rhs )
   {  if ( ap != rhs.ap )
      {  delete ap;  ap = rhs.ap;  rhs.ap = 0;  }
      return *this;  }
```

Τώρα μπορείς να καταλάβεις γιατί πολλές φορές γράφαμε για «*αντιγραφικό τελεστή εκχώρησης*»· όπως βλέπεις εδώ έχουμε έναν *μεταβιβαστικό* τελεστή εκχώρησης. Αφού η *reset()* βάζει ως νέο βέλος αυτό που επιστρέφει η `rhs.release()` το βέλος του *rhs* δεν αντιγράφεται αλλά μηδενίζεται. Για να είναι αυτό δυνατό η παράμετρος δεν έχει "`const`".

Όπως βλέπεις, όλες οι μέθοδοι έχουν τα χαρακτηριστικά που πρέπει για να γλυτώσουμε από τα προβλήματα που παραθέσαμε στην αρχή:

- Δεν επιτρέπονται αντίγραφα βελών.

[13] Στο C++11 ο δημιουργός μεταβίβασης ορίζεται με πιο συγκεκριμένο τρόπο και έχει την ίδια θέση με τον συνήθη δημιουργό αντιγραφής.

- Όταν αλλάζουμε την τιμή βέλους που έχει την κυριότητα κάποιου δυναμικού αντικειμενου ανακυκλώνεται και το αντικείμενο.
- Κάθε φορά που ανακυκλώνεται το δυναμικό αντικείμενο μηδενίζεται η τιμή του βέλους που το είχε.

Υπάρχει όμως μια εξαίρεση: η *get()*!

Να λοιπόν πώς θα είναι το **AutoPtr.h**:

```
#ifndef _AUTOPTR_H
#define _AUTOPTR_H

template < typename K >
class AutoPtr
{
public:
   explicit AutoPtr( K* p=0 )  {  ap = p;  }
   AutoPtr( AutoPtr& other ) {  ap = other.release();  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
   AutoPtr& operator=( AutoPtr& rhs )
   {  reset( rhs.release() );  return *this;  }
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
   void reset( K* p=0 )
   {  if ( ap != p ) {  delete ap;  ap = p;  }  }
private:
   K* ap;
}; // AutoPtr

#endif // _AUTOPTR_H
```

Το παρακάτω πρόγραμμα περιέχει όλες τις δοκιμές που είδαμε παραπάνω:

```
#include <iostream>
#include <string>

#include "AutoPtr.h"
#include "Date.cpp"

using namespace std;

int main()
{
   AutoPtr< string > aps( new string("abc") );
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;

   AutoPtr< Date >  apd( new Date(2010, 11, 26) );
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
}
```

Πρόσεξε το εξής: Ενώ έχουμε δύο "**new**" δεν υπάρχουν "**delete**" διότι δεν χρειάζονται την ανακύκλωση των δυναμικών αντικειμένων τη φροντίζουν οι καταστροφείς. Είδαμε λοιπόν έναν τρόπο για να εφοδιάσουμε τη C++ με μηχανισμό «**αποκομιδής απορριμάτων**» (garbage collection). Βέβαια δεν είναι πλήρης, αφού δεν καλύπτει και δυναμικούς πίνακες αλλά είναι ένα πολύ καλό εργαλείο.

25.8.1 *std*: : *auto_ptr*

Αν στο παραπάνω πρόγραμμα με τα παραδείγματα αλλάξεις:

- το "`#include "AutoPtr.h"`" σε "`#include <memory>`" και
- το "`AutoPtr`" σε "`auto_ptr`"

θα πάρεις τα ίδια αποτελέσματα· αυτήν τη φορά όμως με χρήση του περιγράμματος (*std*::)*auto_ptr* που μας δίνει η C++.

Δες την υλοποίηση του `auto_ptr` στο `memory` της C++ που χρησιμοποιείς και πιθανότατα θα δεις ότι είναι σαν αυτήν του `AutoPtr` που δώσαμε πιο πάνω. Τώρα που είδες πώς και γιατί γράφηκε αυτή η παραμετρική κλάση ας δούμε πώς μπορείς να τη χρησιμοποιείς.

1. Μπορείς να δηλώσεις μεταβλητές ως εξής:

```
auto_ptr<int> api1( new int(375) );
auto_ptr<string> aps1( new string("F.C.Barcelona") );
auto_ptr<int> api2;
auto_ptr<string> aps2( aps1 );
```

- Η πρώτη δήλωση έχει ως αποτέλεσμα: η *api1* να έχει κυριότητα μιας δυναμικής μεταβλητής (τύπου `int`) με έχει τιμή 375.
- Η δεύτερη δήλωση έχει αποτέλεσμα: η *aps1* κατέχει μια δυναμική μεταβλητή (τύπου *string*) που έχει τιμή `"F.C.Barcelona"`.
- Το αποτέλεσμα της τρίτης δήλωσης: η *api2* να δεν κατέχει κάποια δυναμική μεταβλητή.
- Η τέταρτη δήλωση έχει ως αποτέλεσμα η *aps2* να πάρει την κυριότητα της δυναμικής μεταβλητής (τύπου `std::string`) που έχει τιμή `"F.C.Barcelona"` και ήταν αρχικώς στην κατοχή της *aps1*. Η *aps1* δεν κατέχει πια κάποια δυναμική μεταβλητή.

Προσοχή όμως!

α) *Τα παρακάτω απαγορεύονται*:

```
int  k( 255 );
auto_ptr<int> api0( &k );
```

Δεν μπορείς να δώσεις στην κατοχή ενός έξυπνου βέλους μια μεταβλητή που δεν είναι δυναμική. Γιατί; Διότι η ώρα του "`delete`" θα έλθει οπωσδήποτε! Δυστυχώς, ο μεταγλωττιστής δεν θα σε αποτρέψει! Όλα θα είναι στη δική σου την ευθύνη!

β) Κατ' αρχήν μπορείς να συνεχίσεις να χειρίζεσαι μια δυναμική μεταβλητή με «παραδοσιακό» βέλος και μετά την κατοχή της από ένα έξυπνο βέλος. Π.χ. οι παρακάτω:

```
int* pi1( new int(511) );
auto_ptr<int> api( pi1 );
cout << *api << endl;
*pi1 = 375;
cout << *api << endl;
int* pi2( api.get() );
*pi2 = 101;
cout << *api << endl;
```

δίνουν:

```
511
375
101
```

Πάντως *κάτι τέτοιο είναι εξαιρετικά επικίνδυνο*!

2. Μπορείς να ελέγξεις την τιμή του κρυμμένου βέλους παίρνοντάς το με τη μέθοδο *get()*. Μπορείς να κάνεις αποπαραπομπή με το "`*`" όπως και με τα απλά βέλη.

Μετά τις παραπάνω δηλώσεις οι παρακάτω εντολές

```
if ( api1.get() != 0 ) cout << *api1 << endl;
                  else cout << " api1 NULL" << endl;
if ( aps1.get() != 0 ) cout << *aps1 << endl;
                  else cout << " aps1 NULL" << endl;
if ( api2.get() != 0 ) cout << *api2 << endl;
                  else cout << " api2 NULL" << endl;
```

```
   if ( aps2.get() != 0 ) cout << *aps2 << endl;
                      else cout << " aps2 NULL" << endl;
```

θα δώσουν:

```
375
 aps1 NULL
 api2 NULL
F.C.Barcelona
```

3. Να ξαναγυρίσουμε στις ιδιαιτερότητες του «δημιουργού αντιγραφής» που, όπως είπαμε, είναι *δημιουργός μεταβίβασης*. Μετά την τέταρτη περίπτωση του σημείου 1, δες και αυτό:

```
   cout << *api1 << endl;
   cout << *(f(api1)) << endl;
   if ( api1.get() != 0 ) cout << *api1 << endl;
                      else cout << " api1 NULL" << endl;
```

όπου:

```
auto_ptr<int> f( auto_ptr<int> x )
{
   *x += 1;
   return x;
} // f
```

Αποτέλεσμα:

```
375
376
 api1 NULL
```

Αρχικώς η *api1* κατέχει βέλος προς δυναμική μεταβλητή που έχει τιμή 375. Όταν καλείται η *f()*, η *api1* περνάει ως παράμετρος τιμής. Αυτό σημαίνει ότι χρησιμοποιείται ο δημιουργός αντιγραφής με αποτέλεσμα η *api1* να χάνει την κατοχή του βέλους προς τη δυναμική μεταβλητή. Την κατοχή έχει πια η τοπική μεταβλητή της συνάρτησης *x*.

Με την εκτέλεση της `return` και η *x* χάνει την κατοχή του βέλους, αφού και πάλι εκτελείται ο δημιουργός αντιγραφής, αλλά αυτό είναι κάτι που δεν μπορούμε να το δούμε ούτε και μας ενδιαφέρει!

4. Μπορείς να αλλάξεις την τιμή μιας τέτοιας μεταβλητής είτε με τη *reset()* είτε με τον (μεταβιβαστικό) τελεστή εκχώρησης "`=`".

Ας θυμίσουμε την επικεφαλίδα της *reset()*:

```
   void reset( K* p = 0 )
```

Δηλαδή, η *reset()* περιμένει όρισμα που είναι σύνηθες βέλος. Έτσι, οι εντολές:

```
   api1.reset( new int(1023) );
   cout << *api1 << endl;
   cout << *aps2 << endl;
   aps2.reset( new string("Kavala F.C.") );
   cout << *aps2 << endl;
```

θα δώσουν:

```
1023
F.C.Barcelona
Kavala F.C.
```

Όπως στη δήλωση, έτσι και στη *reset()*,

α) *δεν επιτρέπεται να περάσεις βέλος προς μια μη δυναμική μεταβλητή,*

β) παρ' όλο που μπορείς να χειρίζεσαι μια δυναμική μεταβλητή και με το απλό βέλος, *από τη στιγμή που το πέρασες σε ένα έξυπνο βέλος μην το ξαναπειράξεις* (το απλό βέλος).

Ο τελεστής "`=`" δουλεύει ως εξής:

```
   auto_ptr& operator=( auto_ptr& rhs ) throw()
   {  reset( rhs.release() );  return *this;  }
```

Πρόσεξε ότι η παράμετρος είναι αναφοράς. Επομένως, μπορείς να μεταβιβάσεις την κατοχή βέλους από άλλη μεταβλητή-έξυπνο-βέλος. Μπορείς λοιπόν να βάλεις:

```
double* pd2( new double(7.5) );
auto_ptr<double> apd2( pd2 );
auto_ptr<double> apd3;
apd3 = apd2;
```

αλλά όχι:

```
apd3 = auto_ptr<double>( new double(7.5) );
```

ούτε:

```
apd3 = f( pd2 );
```

Φυσικά, το σπουδαιότερο που πρέπει να θυμάσαι είναι ότι με *τον τελεστή αυτόν γίνεται μεταβίβαση τιμής και όχι εκχώρηση όπως την ξέρουμε*. Με την:

```
apd3 = apd2;
```

η *apd3* γίνεται κάτοχος του βέλους που είχε η *apd2* αλλά η *apd2* χάνει την κατοχή και καθαρίζεται (*apd2*.*ap* = 0).

5. Με τη *reset()* μπορείς να ανακυκλώσεις τη δυναμική μνήμη που διαχειρίζεται ένα έξυπνο βέλος (περίπου ό,τι κάνεις με τον "`delete`" για τα συνήθη βέλη). Π.χ.:

```
apd3.reset();
```

6. Με τη *release()* μπορείς να ξαναπάρεις τον έλεγχο από το έξυπνο βέλος:

```
double* pd( apd3.release() );
```

25.8.2 Συνελόντι Ειπείν...

Τα έξυπνα βέλη είναι πολύ καλά εργαλεία για να γράφεις πιο σίγουρα προγράμματα με λιγότερα λάθη. Η πιο απλή περίπτωση είναι τα βέλη που παράγονται από το περίγραμμα `auto_ptr`, που βρίσκεται στην πάγια βιβλιοθήκη της C++.

Αν προσπαθήσεις να τα μάθεις και να τα χρησιμοποιείς, τα σημεία που πρέπει να προσέξεις είναι τα εξής:

- Ο δημιουργός αντιγραφής και ο "=" κάνουν μεταβίβαση τιμής και όχι συνήθη αντιγραφή.
- Ο "`delete`" συνήθως δεν χρειάζεται αλλά όπου είναι απαραίτητος αντικαθίσταται από κλήση της "`reset()`".

Μια σοβαρή έλλειψη του *auto_ptr* είναι ότι δεν μπορεί να χειρισθεί δυναμικούς πίνακες.

Πάντως, στο C++11 αποθαρρύνεται η χρήση του *auto_ptr* (που μπορεί να μην υπάρχει σε επόμενες τυποποιήσεις) αφού δίνεται ένα νέο περίγραμμα κλάσης για έξυπνα βέλη, το *unique_ptr*, με πολύ περισσότερες δυνατότητες.

Το C++11 δίνει ακόμη δύο περιγράμματα κλάσεων για έξυπνα βέλη, τα *shared_ptr* και *weak_ptr*, που είναι περίπου αυτά που υπάρχουν στη βιβλιοθήκη Boost με τα ίδια ονόματα.[14]

25.9 Ένα Χρήσιμο Περίγραμμα Κλάσης

Βάζοντας στο πρόγραμμά σου "`#include <limits>`" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*::*numeric_limits*. Η δήλωσή του είναι:

```
template< typename T >
class numeric_limits
{
public:
   static const bool is_specialized = false;
   static T min() throw();
   static T max() throw();
   static const int digits = 0;
   static const int digits10 = 0;
```

[14] `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`

```
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 0;
    static T epsilon() throw();
    static T round_error() throw();
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();
    static const bool is_iec559 = false;
    static const bool is_bounded = false;
    static const bool is_modulo = false;
    static const bool traps = false;
    static const bool tinyness_before = false;
    static const float_round_style round_style = round_toward_zero;
};
```

Μα τι κλάσεις θα βγάλουμε από εδώ; *Τα πάντα είναι "*`static`*"!* Θα το καταλάβεις από τα παραδείγματα που θα δεις στη συνέχεια.

Τα ονόματα των μελών περιγράφουν αρκετά καλά το νόημά τους. Για μερικά από αυτά μπορεί να πρέπει να γυρίσεις στο Κεφ. 17. Στο `limits` υπάρχουν και εξειδικεύσεις για τους βασικούς τύπους `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double` και `long double`.

Ορισμένα μέλη –όπως *min_exponent*, *min_exponent10*, *max_exponent*, *max_exponent10* κλπ και οι συναρτήσεις *epsilon()*, *round_error()* κλπ– έχουν νόημα μόνο για τύπους κινητής υποδιαστολής, δηλαδή τους `float`, `double` και `long double` (και άλλους που τυχόν δίνει κάποια υλοποίηση). Αν κάποιο μέλος δεν έχει νόημα για κάποιον τύπο τότε στην εξειδίκευση αφήνουμε τιμή "`0`" ή "`false`"όπως φαίνεται στην παραπάνω δήλωση.

Με χρήση των κατάλληλων εξειδικεύσεων μπορούμε να ξαναγράψουμε το προγραμματάκι (§2.5):

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
    cout << "INT_MIN = " << INT_MIN << "    INT_MAX = "
         << INT_MAX << endl;
}
```

ως εξής:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
    cout << "INT_MIN = " << numeric_limits<int>::min()
         << "    INT_MAX = " << numeric_limits<int>::max() << endl;
}
```

Δηλαδή:

- Αντί για "`#include <climits>`" βάλαμε "`#include <limits>`".
- Αντί για "`INT_MIN`" βάλαμε "`numeric_limits<int>::min()`".

- Αντί για "`INT_MAX`" βάλαμε "`numeric_limits<int>::max()`".

Ο πρώτος τρόπος γραφής είναι η κληρονομιά από τη C. Ο δεύτερος τρόπος, με το πολύ γράψιμο, είναι «καθαρή C++».

Παρομοίως, το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << DBL_MIN
        << "    DBL_MAX = " << DBL_MAX << endl;
   cout << "DBL_EPSILON = " << DBL_EPSILON << endl;
}
```

θα γίνει:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << numeric_limits<double>::min()
        << "    DBL_MAX = " << numeric_limits<double>::max()
        << endl;
   cout << "DBL_EPSILON = " << numeric_limits<double>::epsilon()
        << endl;
}
```

Από εδώ και πέρα θα προτιμούμε να γράφουμε πιο πολλά και να είμαστε «πιο C++» εκτός από την εξής περίπτωση: όταν ελέγχουμε τα αποτελέσματα συναρτήσεων της C, όπως αυτά που είδαμε στην §24.1.3. Εκεί θα πρέπει να βάζουμε τις προκαθορισμένες σταθερές της C.

Οι κλάσεις που προκύπτουν από τις εξειδικεύσεις της *numeric_limits* είναι παραδείγματα **κλάσεων χαρακτηριστικών** (traits) και η εισαγωγή τους δεν έγινε μόνο για να γράφουμε προγράμματα «πιο C++». Θα δούμε τη χρησιμότητά τους με ένα

Παράδειγμα

Ας πούμε ότι θέλουμε να γράψουμε ένα

```
template < typename T >
class MyClass
{
   void f( ... )
   { . . . }
}
```

που θα μας δίνει στιγμιότυπα για ακέραιους τύπους.

Στη μέθοδο *f* θέλουμε τη μέγιστη τιμή του τύπου *T*. Με αυτά που ξέρουμε μέχρι τώρα θα πρέπει να γράψουμε μια πολλαπλή `if`:

```
if ( T είναι ο int )  {  T mt( INT_MAX ); ... }
else if ( T είναι ο long )  {  T mt( LONG_MAX ); ... }
else . . .
```

Οι συνθήκες "`T είναι ο int`" και οι άλλες παρόμοιες μεταφράζονται σε C++ με τη βοήθεια του `typeinfo`.

Χρησιμοποιώντας το περίγραμμα που είδαμε πιο πάνω τα πράγματα απλουστεύονται πάρα πολύ:

```
T mt( numeric_limits<T>::max() ); ...
```

☜☜☜

Όπως καταλαβαίνεις, οι κλάσεις χαρακτηριστικών είναι πολύτιμο εργαλείο όταν έχουμε να γράψουμε περιγράμματα.

Παρατήρηση:▶

Αξίζει να πούμε κάτι ακόμη, σχετικώς με το πρώτο μας παράδειγμα, το περίγραμμα *BStringT*: Η C++ μας δίνει το περίγραμμα:

```
template< typename charT > struct char_traits
```

και δύο εξειδικεύσεις του:

```
template<> struct char_traits< char >;
template<> struct char_traits< wchar_t >;
```

Με τη χρήση τους θα μπορούσαμε να γράψουμε το *BStringT* πιο εύκολα (και να το κάνουμε πιο αποδοτικό.)◀

25.10 Ανακεφαλαίωση

Με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων με όλες τις λειτουργίες της που να έχει ως παράμετρο τον τύπο του *περιεχομένου*. Η υλοποίηση της δομής για συγκεκριμένο τύπο προκύπτει ως στιγμιότυπο του περιγράμματος.

Οι πιθανές ιδιαιτερότητες που παρουσιάζονται στα στιγμιότυπα για ορισμένους τύπους αντιμετωπίζονται με *εξειδικεύσεις* ή *μερικές εξειδικεύσεις*.

Ενδιαφέρον έχουν οι κληρονομιές περιγραμμάτων:

- Μια κλάση μπορεί να κληρονομεί στιγμιότυπο περιγράμματος κλάσης.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί κάποια κλάση ώστε να χρησιμοποιεί εργαλεία που έχει η κλάση.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί άλλο περίγραμμα κλάσης.

Δύο περιγράμματα κλάσεων που «περιτυλίγουν» βέλη ώστε να εμπλουτίσουν ή/και να αλλάξουν τις ιδιότητές τους είναι: οι *προσεγγιστές* (που ήδη ξέραμε) και τα *έξυπνα βέλη* (μια σύνοψη στην §25.8.2).

Οι *κλάσεις χαρακτηριστικών* όπως η *numeric_limits* είναι πολύ χρήσιμες για να γράφεις περιγράμματα κλάσεων.

Ασκήσεις

25-1 «Μάζεψε» όλα τα περιγράμματα συναρτήσεων για διαχείριση ψηφιοπινάκων σε περίγραμμα κλάσης *Bitmap*.