
PIĄTA JUBILEUSZOWA
WYŻSZA
MIĘDZYNARODOWA
GÓRSKA
SZKOŁA PTI
Szczyrk, 21-25.06.93

V JUBILEUSZOWA WYŻSZA GÓRSKA MIĘDZYNARODOWA SZKOŁA PTI

WSPÓŁCZESNE OBSZARY ZASTOSOWAŃ INFORMATYKI

Szczyrk 21-25 czerwca 1993

Program

PONIEDZIAŁEK 1993.06.21

9^{45} **Otwarcie Szkoły**

10^{00} - 11^{00} **mgr inż. Wojciech Głazek** (DOKP Katowice): *Założenia, projekt i realizacja sieci transmisji danych KOLPAK.*

przerwa na kawę

11^{30} - 12^{15} **mgr inż. Włodzimierz Adamski** (WSK OBR PZL - MIELEC) *Sieci komputerowe w polskim przemyśle lotniczym.*

12^{30} - 13^{15} **mgr inż. Włodzimierz Adamski** (WSK OBR PZL - MIELEC): *Elektroniczna postać dokumentacji konstrukcyjnej i technologicznej.*

15^{00} - 16^{00} **mgr inż. Mariusz Klapper** (Załład Systemów Komputerowych i Oprogramowania "K&MSOFT"): *Praktyczne doświadczenia opracowywania i wdrażania w przedsiębiorstwach kompleksowych systemów informatycznych.*

przerwa na kawę

16^{20} - 17^{20} **dr inż. Jacek Stochlak** (ICL POLAND): *Czy naprawdę dinozaury - rola i miejsce dużych systemów komputerowych w systemach informatycznych przedsiębiorstw.*

17^{45} - 18^{45} **dr Zbigniew A. Nowacki** (AKREM, Łódź): *Szablony klas i listy elastyczne w pakiecie narzędziowym POLY do języka C.*

WTOREK 1993.06.22

9^{00} - 10^{30} **dr Jerzy Skrzypek** (Akademia Ekonomiczna): *Metodologiczne aspekty modelowania procesów gospodarczych (w ujęciu Dynamiki Systemowej).*

przerwa na kawę

11^{00} - 12^{30} **dr inż. Marek Miłosz** (Politechnika Lubelska) *SYMULACJA SYSTEMÓW - Symulacja procesów dyskretnych.*

Zbigniew A. Nowacki
AKREM, Łódź

Szablony klas i listy elastyczne w pakiecie narzędziowym POLY do języka C

1. Wprowadzenie

W pierwszej części tego wykładu chciałbym omówić bibliotekę funkcji POLY wchodzącą, obok zestawu programów POLY-DOS, w skład pakietu narzędziowego POLY do języka C. Natomiast w drugiej części zajmę się bardziej szczegółowo zagadnieniami zasygnalizowanymi w tytule, tj. przedstawię jedno z najważniejszych ostatnio wprowadzonych rozszerzeń języka C - szablony oraz pokażę, w jaki sposób zostały one wykorzystane w pakiecie POLY do implementacji tzw. list elastycznych czyli nowej metody przetwarzania struktur listowych.

2. Biblioteka funkcji POLY dla języka C

Biblioteka POLY dla C powstała z opracowanej wcześniej biblioteki POLY dla CLIPPERA, której większość funkcji była napisana właśnie w języku C. Zostały one zmodyfikowane zgodnie z filozofią C, a ponadto dołączono do nich szereg dodatkowych funkcji, bądź niepotrzebnych w CLIPPERZE, jako że wchodzących tam do podstawowego języka, bądź też niemożliwych w nim do wykorzystania.

Biblioteka może być wykorzystywana zarówno w tradycyjnym C wg standardu ANSI, jak i w C++. W tym drugim przypadku programista ma do dyspozycji wiele dodatkowych udogodnień, których zestaw określany jest mianem POLY++. Wszystkie funkcje dostarczane są dla użytkownika m. in. w postaci źródłowej.

Ponad 300 funkcji biblioteki POLY można podzielić na następujące zasadnicze moduły:

I. Realizacja przerwań BIOS-u.
II. Obsługa ekranu.
III. Obsługa klawiatury.
IV. Obsługa drukarki.
V. Przetwarzanie polskich znaków diakrytycznych.
VI. Komunikacja z rezydentami pakietu.
VII. Manipulacje na łańcuchach znaków.
VIII. Pakowanie i szyfrowanie danych.
IX. Przetwarzanie dat.

X. Przetwarzanie list.
XI. Pełnoekranowe wprowadzanie danych.
XII. Organizacja menu.
XIII. Funkcje wywoływane przez makra języka NewOrder.
XIV. Obsługa okienek.
XV. Obsługa baz danych (upgrade w przygotowaniu).

Funkcje modułów II, III oraz IV obsługują urządzenia zewnętrzne przy pomocy przerwań BIOS-u wywoływanych za pośrednictwem należącej do modułu I funkcji *_intpoly*, bądź też odwołując się bezpośrednio do zmiennych BIOS-u przy użyciu makrodefinicji *FARPOINT*. Ta pierwsza realizuje swoje zadania albo przy pomocy standardowej funkcji *int86*, albo (po wywołaniu funkcji *_vdirect*) poprzez symulację operacji BIOS-u bezpośrednim działaniem na pamięci ekranu. Ponadto *_intpoly* jest jedyną funkcją z biblioteki POLY odwołującą się do funkcji niezdefiniowanych w standardzie ANSI i niedostępnych w systemie UNIX (konkretnie do *int86*). Zatem w celu przeniesienia biblioteki na komputery lub systemy operacyjne różne od IBM PC i DOS, wystarczy zmodyfikować funkcję *_intpoly* i przedefiniować *FARPOINT*. (Oczywiście potrzebna jest w tym celu dobra znajomość BIOS-u dla IBM PC oraz BIOS-u i systemu operacyjnego komputera docelowego).

Dla potrzeb biblioteki POLY zostało zdefiniowanych pięć następujących typów listowych:

```
typedef struct anylist
    {struct anylist *prev; void *any; } *anylist;          // (1)
typedef struct wordlist {struct wordlist *prev; int word;} *wordlist;
typedef struct numlist {struct numlist *prev;long num;} *numlist;
typedef struct namelist {struct namelist *prev; char* name;}
*namelist;
typedef struct keylist {struct keylist *prev;int key;intfun funkcja;}
*keylist;
```

Programista może deklarować własne listy powyższych typów, a w przypadku konieczności użycia innych typów listowych może wzorując się na wersjach źródłowych napisać własne wersje funkcji je obsługujących. Nie jest to jednak, jak zobaczymy dalej, potrzebne w POLY++ z szablonami.

Warto dodać, że przetwarzanie list w pakiecie POLY zostało zrealizowane w oparciu o pewną sztuczkę programistyczną, dzięki czemu listy te dają znacznie większe możliwości niż listy implementowane w standardowych bibliotekach. Oczywiście istnieje tu również ściśle określony algorytm, w pełni widoczny dla użytkownika POLY, ale algorytm ten posiada pewną charakterystyczną cechę: dość trudno go sformułować zajmując się listami od strony teoretycznej, zaś znacznie łatwiej pisząc programy przetwarzające listy, najlepiej w języku C.

Pełnoekranowe wprowadzanie danych wykorzystujące obiekty *GET* i tzw. formaty izometryczne zrealizowane jest w module XI. Obiekty *GET* dostępne są już w bibliotece POLY używanej w zwykłym C, ale POLY++ umożliwia realizację pełnoekranowego strumienia danych przy pomocy obiektów klasy *Getstream*, wg przykładowego schematu:

```
Getstream gs;
gs << Get(...)
<< Get(...)
```

```
...
<< Get(...)
>> SCREEN;
```

Funkcje *Get* zwracają wskaźniki do obiektów *GET* podających dla każdej danej następujące informacje: komentarz wprowadzający, format, nazwa funkcji sprawdzającej poprawność, nazwa funkcji zezwolenia na wprowadzanie i inne.

Ważne miejsce w bibliotece POLY zajmują funkcje z modułu XII umożliwiające zorganizowanie menu dwuwymiarowego, rolowanego lub nie, z dowolną ilością kolumn i wierszy. Dwuwymiarowość jest często naturalną strukturą opcji, a zawsze pozwala zaoszczędzić powierzchnię ekranu. Dzięki wprowadzeniu obiektów *PROMPT*, menu może mieć w razie konieczności dowolny nieregularny kształt. Istnieje również możliwość konstrukcji menu logicznego polegającego na wyborze nie jednej opcji, lecz pewnego podzbioru zbioru opcji prezentowanych. Termin "menu logiczne" pochodzi stąd, że taki wybór musi być zapisany w tablicy o elementach traktowanych jako wartości logiczne i funkcja *_logmenu* zwraca tablicę określającą dokonany wybór. Ponadto w POLY++ zdefiniowany jest operator << umożliwiający konstrukcję menu strumieniowego.

W module XIII zostały zgrupowane funkcje, które nie powinny być przez programistę wywoływane bezpośrednio, a jedynie poprzez makra należące do tzw. języka NewOrder. Język ten wykorzystuje preprocesor C w celu maksymalnego ułatwienia procesu tworzenia helpu kontekstowego, sformatowanego wprowadzania danych oraz wielopoziomowych menu powiązanych ze strukturą programu. Jest on trochę podobny do języka NewOrder dla CLIPPERA 5, aczkolwiek wiele różnic zostało wymuszonych przez odmienne możliwości preprocesorów.

A oto przykładowy fragment programu w języku NewOrder (a właściwie NewOrder++ ze względu na użycie operatora <<):

```
#include "NEWORDER.H"
void PROTOTYPE(help,(void))
PROTOTYPE(fun,(char* par))
void cfun(char*);
GENESIS
    _pldef(0);
    ASSIGN(EXTEND(F1),help)
    fun(ARGUMENT(1));
STOP(0)
FUNCTION(fun,(char* par))
    static DATE dat={1978,4,23};
    int log=TAK;
    double d=-1;
    static char t[]="ALFA21";
    MAINMENU(2,2)
  THISMENU << Prompt(2,4,"Kopiowanie")
        << Prompt(2,24,"baza Danych")
        << Prompt(4,4,"Indeksy")
        << Prompt(4,24,"kaSowanie");
          SELECT
SUBMENU(3,2,1,1)
THISMENU << Prompt(12,15,"Przeszukiwanie")
<< "Testowanie"
<< "Kopiowanie"
<< FREE
```

```
<< par
<< "Wyniki";
SELECT
BEGIN(WITHOUT,CHOICE)
cout << "BEZ WYBORU\n";
END
ENDMENU
BEGIN(0,0)
cout << _p("WYBRANO OPCJE` 0,0\n");
END
BEGIN(OTHER,WISE)
cout << "WYBRANO INNE\n";
END
ENDMENU
    GETSTREAM
                                                        <<
Get(4,0,"Kalendarz",it,CALENDAR,&dat,"DD/MM/RRRR",
            INVERSE,sprawdz_dat,0,TAK)
        << Get(6,40,"Warunek",it,LOGICAL,&log,".....T.....")
        << Get(12,0,_p("Wielkos`c`"),it,NUMERIC,&d,
            "-9,999,990.09")
        << Get(12,40,"Tekst",it,TEXT,&t[0],"XxxxCC");
    READ
STOP(1)
void FUNCTION(help,(void))
    SAVE_SCREEN
    cfun(CONTEXT(1));
    REST_SCREEN
FINISH
```

Jak widać, komendy języka NewOrder mogą dowolnie przeplatać się z instrukcjami i deklaracjami C.

Podejście okienkowe pozwala na tworzenie programów i procedur niezależnych od parametrów aktualnie stosowanego okna. Przy pomocy funkcji *_okno* i pokrewnych (należących do modułu XIV) możliwe jest utworzenie listy okien, przy czym okno znajdujące się w danym momencie na końcu listy nazywane jest oknem aktywnym. Podejście okienkowe nie ogranicza się tylko do funkcji biblioteki POLY, ale może być wykorzystywane w wywołaniach funkcji użytkownika oraz funkcji z innych bibliotek; wszędzie tam, gdzie jako parametry stosuje się współrzędne okna lub kursora.

W czasie przygotowywania tego wykładu funkcje modułu XV były testowane. Przewiduje się, że upgrade pakietu POLY zawierający moduł obsługi baz danych wejdzie do sprzedaży na jesieni 1993 r.

3. Szablony - najnowsze rozszerzenie języka C

Pojęcie szablonów zwanych również sparametryzowanymi typami zostało wprowadzone do standardu Cfront 3.0 czyli najnowszego wydania obiektowo zorientowanego nadzbioru języka C opracowanego przez Bjarne Stroustrupa z firmy AT&T Bell Laboratories. Pierwsza kompletna komercyjna implementacja szablonów pojawiła się w kompilatorze Borland C++ 3.1 dostępnym również w Polsce.

Najogólniej mówiąc, jeśli tylko stwierdzimy konieczność napisania szeregu bardzo podobnych procedur, powinniśmy pomyśleć o szablonach. Jednym z najprostszych, lecz mimo to interesujących przykładów jest funkcja *swap(x,y),* której zadaniem jest zamiana zawartości dwóch parametrów *x* i *y* identycznego typu. Konieczność wykonania operacji tego rodzaju jest często spotykana w informatyce, wiadomo jak ją przeprowadzić, ale dotychczas trzeba było taką funkcję pisać oddzielnie dla każdego typu.

Koncepcja szablonów rozwiązuje powyższy problem raz na zawsze. Aby ją zastosować, trzeba przede wszystkim zadeklarować szablon przy pomocy deklaracji postaci

```
Template<argument_1,...,argument_k>                    // (2)
```

gdzie *k >= 1*, a każdy argument szablonu może być jednej z następujących postaci

```
Class nazwa                                            // (3)
nazwa_typu nazwa1                                      // (4)
nazwa_typu nazwa1 = wyrażenie stałe                    // (5)
```

Bezpośrednio po deklaracji szablonu powinna wystąpić deklaracja lub prototyp funkcji, bądź też definicja klasy, w których można używać identyfikatorów *nazwa* z argumentów postaci (3) dla oznaczenia dowolnych typów (oczywiście ten sam identyfikator użyty wielokrotnie reprezentuje ten sam typ), zaś identyfikatorów *nazwa1* z argumentów (4) lub (5) dla oznaczenia wielkości stałych (argumenty te są nazywane *pozatypowymi*).

Deklaracja funkcji poprzedzona deklaracją szablonu określana jest mianem *szablonu funkcyjnego* lub *funkcji ogólnej (ang. generic function)*. W szablonie funkcyjnym nie wolno stosować argumentów pozatypowych. Natomiast konkretny przypadek użycia szablonu funkcyjnego nazywa się *funkcją szablonową* i formalnie rzecz biorąc nie różni się niczym od wywołania zwykłej funkcji; kompilator sam zadba o to, aby został wykorzystany odpowiedni szablon.

Powracając do naszego przykładu funkcji *swap*, możemy zastosować następujący szablon funkcyjny:

```
template <class T>
void swap(T & x, T & y)
{
   T robocza;
    robocza=x;
    x=y;
    y=robocza;
};
```

Dla każdego typu *T* takiego, że w programie znajduje się przynajmniej jedno wywołanie postaci *swap(x,y)*, gdzie *x* i *y* są typu *T*, kompilator wygeneruje dokładnie jeden egzemplarz funkcji szablonowej przeprowadzającej żądaną wymianę. Zatem jest to dokładnie to, co musieliśmy do tej pory robić "ręcznie". Widać też, że bez straty efektywności można teraz wykonać w C wiele rzeczy możliwych dotąd tylko w językach takich jak CLIPPER, gdzie typ zmiennej jest znany dopiero w czasie realizacji programu.

Klasa, której definicja poprzedzona jest deklaracją szablonu nazywa się *szablonem klas* albo *generatorem klas* albo też *klasą ogólną (ang. generic class)*. Jeśli *X* jest nazwą takiej klasy, to użycie jej konkretnej wersji musi mieć postać

```
X<T1, ..., Tm>                                    // (6)
```

gdzie *T1, ..., Tm* powinny być aktualnymi argumentami odpowiadającym formalnym argumentom z (2). Dokładniej mówiąc, pod argumenty postaci (3) należy podstawić konkretne typy, zaś pod argumenty pozatypowe - stałe wyrażenia typu zgodnego z podanym w (4) lub (5). Ilość argumentów aktualnych może być mniejsza od ilości argumentów formalnych tylko wtedy, gdy wszystkie brakujące argumenty były postaci (5); występujące w nich wyrażenia są traktowane jako wartości domyślne. Zaleca się zastępowanie (6) odpowiednią nazwą wprowadzoną przez *typedef*.

4. Listy elastyczne

Aczkolwiek nie jestem w stanie zaprezentować na tym wykładzie całego algorytmu przetwarzania list w bibliotece POLY, to jednak przedstawiam tu jego najistotniejszą część, a mianowicie strukturę danych tworzących listę. Z drugiej strony ktoś może się poczuć rozczarowany, ponieważ struktura ta jest niezwykle prosta (ale właśnie dlatego to działa). Definicja podstawowego szablonu klas o nazwie *polylist* w pliku POLYTEMP.H zawiera tylko dwie składowe danych i zaczyna się następującym fragmentem:

```
template <class T>
class polylist
{
anylist lista;
static T* (*copy)(T*);
 ...
```

Dalej występują już tylko definicje operatorów i składowych funkcji. Najistotniejszą składową danych jest oczywiście lista, która jest typu *anylist* dokładnie takiego, jaki został zdefiniowany przez (1), z tym że w C++ musimy użyć

```
typedef struct Anylist {Anylist *prev; void *any; } Anylist;
typedef Anylist *anylist;
```

Pomimo formalnych różnic powyższa definicja znaczy to samo co (1), a zatem funkcje obsługi list typu *anylist* zawarte w zwykłym POLY są również wykorzystywane w POLY++.

Aby przetwarzać listy w POLY++, należy włączyć do programu plik POLYTEMP.H oraz zadeklarować używane typy listowe przy pomocy makro

```
POLYLIST(typ_listy, typ_obiektu, kopia)
```

gdzie *typ_listy* jest unikalną (dotychczas nieużywaną) nazwą typu, *typ_obiektu* jest nazwą dowolnego już istniejącego typu, zaś *kopia* jest nazwą funkcji o podanym wcześniej prototypie

```
*typ_obiektu kopia(*typ_obiektu);
```

lub nazwą specjalną *NOCOPY*. Od tego momentu w programie mogą występować deklaracje takie jak, przykładowo,

```
typ_listy a, b, c;
```

która powoduje utworzenie trzech początkowo pustych list *a*, *b* i *c* złożonych z wskaźników do obiektów typu *typ_obiektu*. Parametr *kopia* określa, czy obiekty dołączane do listy mają być kopiowane (wyjaśniamy to dokładniej dalej).

W programie makro *POLYLIST* może wystąpić dowolnie wiele razy i utworzone w ten sposób różne typy listowe mogą być wykorzystywane jednocześnie. Jak łatwo można się domyśleć, każde makro *POLYLIST(x,y,z)* m.in. nadaje przy pomocy *typedef* nazwę *x* dla typu

polylist<y> (tj. konkretyzacji typu *polylist* wg (6)), zaś *z* służy do inicjacji członu statycznego *copy*.

Po zadeklarowaniu zmiennych listowych można poddawać je działaniu kilku operatorów. I tak operator

```
list <= pointer
```

dodaje wskaźnik *pointer* na końcu listy list zwracając referencję do list, co umożliwia wykonywanie operacji wielokrotnych, np.,

```
POLYLIST(textlist, char, NOCOPY)
 ...
textlist li;
li <= "CD" <= "EF" <= "GH" <= "IJ";
```

W przypadku, gdy parametr kopia makra *POLYLIST* jest różny od *NOCOPY*, to do listy list w (2) dodawany jest nie *pointer*, ale *kopia(pointer)*. Na ogół oczekuje się przy tym, że funkcja *kopia* skopiuje obiekt wskazywany przez *pointer* do nowego obszaru przydzielonego przy pomocy standardowego operatora *new* i zwróci adres kopii. Zatem ten wariant może być interpretowany jako wstawianie do listy obiektów zamiast wskaźników.

Należy podkreślić, iż dzięki wykorzystaniu szablonów istnieje tu pełna kontrola typów. Na przykład sekwencja

```
POLYLIST(intlist, int, NOCOPY)
 ...
intlist inli;
inli <= "CD";
```

spowoduje zasygnalizowanie błędu już na etapie kompilacji. Zatem chociaż te listy sprowadzają się, jak widzieliśmy, do list typu *anylist*, programista może uważać, iż dysponuje listami nieskończenie wielu rodzajów.

Unarne operatory + i –– zastosowane do listy zwracają ostatni (czyli ostatnio wprowadzony) wskaźnik na liście. Jedyną różnicą między nimi jest to, że + pozostawia listę bez zmiany, a –– usuwa ostatni wskaźnik z listy. W przypadku, gdy lista jest pusta, oba operatory zwracają 0.

Również i w tym przypadku użycie szablonów daje nam istotne korzyści polegające na tym, iż zwracane wskaźniki są odpowiedniego typu. Przykładowo, sekwencja

```
POLYLIST(textlist, char, kopiuj)
 ...
textlist li;
...
cout << --li;
```

spowoduje wyprowadzenie tekstu, podczas gdy przy bezpośrednim korzystaniu z list typu *anylist* niezbędny byłby dodatkowy operator rzutowania.

Usuwanie wskaźników z listy może być także przeprowadzone przy pomocy operatora –=. Wyrażenie

```
list -= number
```

kasuje *number* ostatnich wskaźników z listy *list*. Dodatkowo, jeśli została określona funkcja kopiująca (parametr *kopia* był różny od *NOCOPY*), to do wszystkich usuwanych wskaźników stosowany jest operator *delete*. Operator –= zwraca referencję do list.

Operator = służy do kopiowania list. Ściśle mówiąc, wyrażenie

```
list1 = list2
```

powoduje skasowanie wszystkich wskaźników listy *list1*, a następnie przyporządkowanie do zmiennej *list1* kopii listy *list2*. Jeśli funkcja kopiująca została określona, to będzie ona

wykorzystana do kopiowania obiektów wskazywanych przez wskaźniki listy *list2*, a ponadto obiekty wskazywane przez pierwotne wskaźniki z *list1* będą kasowane operatorem *delete*.

Zachowywane są wszystkie intuicyjne aspekty operatora =. Przykładowo, we fragmencie programu

```
POLYLIST(textlist,char,copy)
 ...
textlist a;
 ...
textlist b = a, c;
a = 0;
c = a = b;
```

lista *b* już w momencie zadeklarowania zostaje utworzona jako dokładna kopia listy *a*, następnie lista *a* jest czyszczona, aż wreszcie listy *c* i *a* stają się dokładnymi acz różnymi kopiami listy *b*.

Chociaż funkcja *Len* zwraca długość listy, nie powinno się jej używać do testów na to, czy lista jest pusta. Wynika to z faktu, iż *Len* po prostu czyta całą listę zliczając jej elementy. Jednakże operator ! został zdefiniowany tak, aby wyrażenie !*list* było różne od zera tylko wtedy, gdy list jest puste. Tym samym w warunkach można stosować wyrażenie !!*list*, ale nie można równoważnego pod względem logicznym wyrażenia *list* (dlaczego?).

Omówimy teraz ostatni, ale najważniejszy operator list elastycznych, oznaczany symbolem |. Warto w tym miejscu przypomnieć, że oparte na teoretycznych algorytmach funkcje zawarte w bibliotekach standardowych w celu przetworzenia elementów znajdujących się wewnątrz listy wykorzystują tzw. iteratory. Aczkolwiek producenci oprogramowania starają się pisać funkcje obsługujące iteratory w sposób jak najbardziej ogólny, to jednak zawsze muszą wystąpić pewne ograniczenia wynikające z faktu, iż iteratory są dodatkowymi obiektami nie dającymi się opisać w terminach samej listy. Otóż operator | umożliwia przetwarzanie list bez pośrednictwa iteratorów, dając programiście możliwość pełnej kontroli i optymalizacji tego procesu.

Wyrażenie

```
list | number
```

gdzie *list* jest listą, a *number* liczbą całkowitą nieujemną, definiuje się jako adres podlisty listy *list* powstałej przez pominięcie *number* ostatnich (ostatnio wprowadzonych) elementów. Należy podkreślić, że operator | zwraca tylko adres, a więc żadna nowa lista nie jest tworzona.

Skoro mamy już adres, to natychmiast przychodzi na myśl operator wyłuskania. Oczywiście, zgodnie z podstawowymi regułami C, również unarny operator * nie tworzy żadnego nowego bytu. Zatem *(*list* | *number*) jest listą będącą fizyczną częścią listy *list* (identyczną z *list* w przypadku *number* = 0). W konsekwencji widzimy natychmiast, w jaki sposób odczytać zawartość elementu listy, za którym znajduje się *number* innych elementów. Mianowicie, wystarczy ewaluować wyrażenie

```
+ * (list | number)                                    // (7)
```

Rzecz jasna, czas ewaluacji zależy od parametru *number*. Jednak podróżując po liście od elementu do elementu nie musimy za każdym razem odwoływać się do wyrażenia (7). Możemy natomiast znacznie zoptymalizować przetwarzanie, jeśli zauważymy, iż podlista podlisty jest znów podlistą całej listy, a ściśle mówiąc zachodzi następujący wzór:

```
*(list | number1) | number2 = list | number1 + number2
```

Wynika stąd, że optymalna pętla wyprowadzająca elementy listy na standardowe wyjście może wyglądać następująco:

```
textlist li, *a;
 ...
```

```
for(a = li|0; !! *a; a = *a|1) cout << + *a ;                    // (8)
```

Tutaj wyrażenia z operatorem | wykonywane są z maksymalną prędkością (oczywiście zamiast *li*|0 można by było napisać &*li*). Ponieważ zmiany adresu o 1 element zdarzają się bardzo często, wprowadzono specjalną funkcję *Inc* działającą na adresach list. Jeśli *adr* jest zmienną typu "wskaźnik do listy", to *Inc(adr)* wykonuje podstawienie *adr* = **adr* | 1 oraz wyłuskuje i zwraca listę poprzednio wskazywaną przez *adr*. Stosując *Inc* pętla (8) może być zapisana następująco:

```
a = li|0; while(!! *a) cout << +Inc(a);                    // (9)
```

Odczytywanie elementów jest proste z intuicyjnego punktu widzenia. Jednak skoro podlista jest normalną listą, to przy pomocy operatorów –– i –= można kasować elementy znajdujące się na końcu podlisty. Powstaje zatem pytanie: Co w tym przypadku stanie się z całą listą? Otóż nie ma powodów do obaw; w liście nie powstanie dziura. Po prostu można w ten sposób kasować środkowe elementy listy.

Podobne uwagi dotyczą dodawania elementów do podlisty. Przykładowa instrukcja

```
*(li | k) <= "A" <= "B" <= "C"
```

spowoduje wstawienie trzech elementów do środka listy *li* (po elemencie *"C"* będzie *k* już przedtem istniejących elementów). Zatem lista *li* zostanie, obrazowo mówiąc, rozepchnięta niczym rajstopy elastyczne w celu przyjęcia nowych elementów. Ta właśnie cecha list implementowanych w pakiecie POLY uzasadnia nazwę listy elastyczne.

Oczywiście w tym samym wyrażeniu można usunąć pewne elementy, a na ich miejsce wstawić nowe. Taki sposób pracy jest nawet zalecany, ponieważ pozycja wprowadzenia zmian znajdowana jest tylko raz. W razie potrzeby adres odpowiedniej podlisty może być zapamiętany w zmiennej wskaźnikowej.

Warto tu zauważyć, że listy wykorzystujące pojęcie iteratora (implementowane w standardowych bibliotekach) nie są elastyczne. W przypadku list jednokierunkowych nowe elementy mogą być dodawane na tylko końcu listy, zaś dla list dwukierunkowych - na końcu lub początku.

Na podlistach list elastycznych można wykonywać naprawdę dowolne operacje. Wszystkie poniższe przykłady są poprawne i mogą być stosowane w sposób celowy.

```
*(li|k) = 0;              // lista li będzie teraz zawierać
                          // co najwyżej k elementów
*(li|n) = li2;            // konkatenacja list
*(li|k) = *(li|n);
li <= (--*(li|3));
```

Ostatnia operacja powoduje przestawienie elementów na liście. Jeśli została określona funkcja kopiująca, to takie wyrażenie dobrze jest umieszczać wewnątrz makra *REFER*, jak to pokazuje poniższy przykład:

```
REFER(textlist, li <= (-- *(li | 3) ) )
```

Wszystkie operatory umieszczone tym sposobem w makro *REFER* wykonywane są bez kopiowania i kasowania obiektów, a więc przestawienie zostanie wykonane szybciej.

Jak już stwierdziliśmy wcześniej, wyniki działania operatora | mogą być przechowywane w zmiennych wskaźnikowych. Natomiast po wyłuskaniu nie powinny być one przesyłane operatorem = do innych zmiennych listowych, ponieważ w ten sposób otrzymujemy kopie podlist (chyba, że jest to specjalnie wymagane). Niemniej jednak można napisać, przykładowo,

```
{
 textlist &b = *(li | 4);
 ...
}
```

Tutaj *b* będzie w dalszym ciągu podlistą listy *li* i wszystkie zmiany *b* zostaną odzwierciedlone w *li*.

Pętle (8) i (9) odczytywały elementy listy w porządku odwrotnym do tego, w jakim elementy te zostały do listy wstawione. Nie znaczy to jednak wcale, że listy elastyczne są tylko listami typu LIFO. Poniższy przykład wskazuje, jak należy wstawiać elementy do listy, aby takie pętle jak (8) i (9) odczytały elementy listy w tej samej kolejności, w jakiej te elementy zostały do listy wstawione.

```
textlist li, *a;
a = li|0;
 ...
a= *a <= p | 1;                                    // (10)
 ...
```

Dla list FIFO, wyrażenie (10) powinno być stosowane zamiast zwykłego

```
li <= p;                                           //
(11)
```

Oczywiście to działa dlatego, że (10) powoduje zawsze wstawienie nowego elementu na początku listy. Zauważmy, iż dzięki wykorzystaniu zmiennej adresowej *a*, (10) działa niemal tak samo szybko jak (11).

Listy elastyczne wydają się być, na pierwszy rzut oka, listami jednokierunkowymi. Prawdą jest (zob. (1)), iż struktury list elastycznych zawierają tylko jeden wskaźnik (*prev*) do innej struktury listy, co rzekomo wyklucza możliwość poruszania się w dwóch kierunkach. Dlatego jedną z najciekawszych własności list elastycznych jest fakt, że to wrażenie jest fałszywe. Pokażemy bowiem za chwilę, w jaki sposób zorganizować poruszanie się w dwóch kierunkach na liście elastycznej. Ważną rolę, przynajmniej dla elegancji wyrażenia algorytmu, znów odgrywają tu szablony klas.

Poglądowo mówiąc, zastosowaną tu metodę można porównać ze strategią wytrawnego trapera, który poruszając się po prerii zostawia sobie tylko wiadome znaki ułatwiające powrót. Skoro w makro *POLYLIST* można wstawiać dowolne typy, więc nic nie stoi na przeszkodzie, aby jako argument aktualny szablonu klas wykorzystać typ, który sam został utworzony przy pomocy szablonu klas. Rzeczywiście, kompilator Borlanda przyjmuje to bez zastrzeżeń i możemy napisać przykładowe deklaracje:

```
POLYLIST(textlist, char, kopia)
POLYLIST(overlist, textlist, NOCOPY)
textlist li,*a;
overlist ov,*av;
```

Zatem lista *ov* może być nazwana "listą list" (określenie to brzmi groźnie, ale w rzeczywistości *ov* jest zwykłą listą wskaźników). Teraz wystarczy tylko poruszając się po liście *li*, przy użyciu pętelek takich jak (8) lub (9), zapełniać listę *ov* instrukcją *ov* <= *a*. To zabezpieczy nam powrót przy pomocy takich pętelek jak

```
// w przeciwnym kierunku
for(av = ov|0; !! *av; Inc(av)) cout << +* +* av ;
// kasowanie w przeciwnym kierunku
while(!! ov) cout << + * -- ov ;
```

Należy podkreślić, że lista *ov* nie może być zastąpiona listą typu *textlist*, ale odwrotnie uporządkowaną, bowiem wtedy poruszając się do tyłu nie wiedzielibyśmy, jak wrócić na listę podstawową. Innymi słowy, zastosowane tu podejście wymaga posiadania umiejętności stosowania adresów podlist. Z drugiej strony, jest ono bardziej efektywne od klasycznych list dwukierunkowych, ponieważ nie jest potrzebna ciągła aktualizacja dwóch wskaźników (tworzenie listy list inicjujemy dopiero wtedy, gdy może być potrzebne poruszanie się w dwóch kierunkach).

5. Zakończenie

Istnieje szeroki wachlarz problemów, w których zastosowanie list wydaje się być celowe. Listy mogą zmniejszyć wykorzystanie pamięci, a jednocześnie pozwalają wyeliminować ograniczenia nakładane w wyniku beztroskiego stosowania tablic. Sądzę, że listy elastyczne z pakietu POLY tworzą bardzo efektywny i wygodny w użyciu mechanizm, który może być z powodzeniem wykorzystany w takich przypadkach.