ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1996
Seria: INFORMATYKA z. 31 Nr kol. 1348

Jarosław JASIŃSKI

PRZYKŁADY UŻYCIA TECHNIK ZORIENTOWANYCH OBIEKTOWO DO ZARZĄDZANIA PAMIĘCIĄ OPERACYJNĄ W JĘZYKU C++

Streszczenie. W artykule przedstawiono niektóre problemy związane z zarządzaniem pamięcią operacyjną, jakie napotkano podczas realizacji pracy dyplomowej „Narzędzia programowe umożliwiające inspekcję zawartości pamięci w środowisku MS Windows". Opisano obiektowo zorientowane techniki programistyczne zastosowane do ich przezwyciężenia. Zwrócono uwagę na korzyści wynikłe z zastosowania struktur danych, takich jak kontenery i inteligentne wskaźniki możliwe do stworzenia tylko w językach obiektowych. Na przykładzie implementacji sterty wskazano na korzyści, jakie daje obiektowe podejście przy implementacji klasycznych struktur danych.

EXAMPLES OF USE OF OBJECT-ORIENTED TECHNIQUES FOR MEMORY MANAGEMENT IN C++ LANGUAGE

Summary. In this article some problems related to memory management encountered in developing master's thesis „Program Tools Enabling Inspection of Memory Content in MS Windows Environment" are presented. Object-oriented techniques used to solve them are described. Advantages of data structures available only in object-oriented languages and object-oriented approach for implementing classic data structures are emphasised.

DIE BEISPIELE VON ANWENDUNG DIE OBJEKT-PROGRAMMIERUNG FÜR SPEICHERVERWALTUNG IN PROGRAMMIERSPRACHE C++

Zusammenfassung. Dieser Artikel presentiert einige Problemen verbunden mit Speicherverwaltung, die während die Realisation von Diplomwerk "Inspektion des Speicher in MS Windows Umgebung" gefunden waren. Die Objekt-orientierte Techniken, die eine Lösung von diese Problemen bilden, sind geschrieben.

1. Wstęp

W ramach pracy [3] został napisany program „WoP" („Wszystko o Pamięci") umożliwiający dokonywanie inspekcji zawartości pamięci w środowisku MS Windows; jako języka programowania użyto C++. Program ten może dostarczyć pełnych informacji o aktualnym stanie wykorzystania pamięci w systemie operacyjnym oraz umożliwia oglądanie jej zawartości. Przez stan wykorzystania pamięci rozumie się uruchomione zadania, załadowane moduły programowe (aplikacje i biblioteki dynamiczne) oraz obszary pamięci zawarte na stercie globalnej systemu i na stertach lokalnych poszczególnych zadań [4]. Możliwe jest obejrzenie zawartości dowolnego obszaru pamięci ze sterty globalnej lub lokalnej. Wymienione elementy stanu wykorzystania pamięci będą w dalszej części artykułu nazywane elementami systemowymi.

Operacje wykonywane przez program „WoP" można podzielić na cztery grupy: zbieranie informacji o elementach systemowych, przechowywanie informacji, prezentowanie informacji użytkownikowi i wykonywanie na nich operacji zleconych przez użytkownika. Operacje, jakie może użytkownik zainicjować, to przede wszystkim sortowanie informacji i ich odświeżenie (tj. anulowanie aktualnych informacji i ponowne ich zebranie). W artykule przedstawione zostaną trzy rozwiązania pochodzące z części programu odpowiedzialnej za przechowywanie informacji o elementach systemowych: użycie obiektów klasy kontenerowej do przechowywania tych informacji, zastosowanie inteligentnych wskaźników jako pośredników w dostępie do struktur danych przechowujących informacje oraz zaimplementowanie struktury danych – sterty – w postaci obiektu.

2. Implementacja tablic przechowujących informacje o elementach systemu

Informacja o jednym elemencie systemowym jest informacją złożoną, ma wiele składowych. Istnieją cztery grupy elementów systemowych i dla każdej stworzono odpowiadającą jej klasę; obiekty jednej klasy reprezentują elementy z odpowiedniej grupy. Dane o jednym elemencie systemowym są przechowywane w reprezentującym go obiekcie. Każda z grup obiektów tej samej klasy jest przechowywana w jednej tablicy. Użycie do tego celu standardowych tablic dostępnych w języku C++ nie jest możliwe, gdyż w chwili kompilacji programu nie jest znana liczba elementów systemowych, a ponadto liczba ta zmienia się w czasie działania programu. Można by użyć tablic alokowanych dynamicznie na stercie programu, lecz wygodniejszym rozwiązaniem okazało się użycie tablic – obiektów klasy kontenerowej pochodzącej z biblioteki klas zawartej w pakiecie Borland C++ 4.5.

Klasa *TArrayAsVector* jest implementacją struktury danych o nazwie tablica, lecz jednocześnie rozmiar tablicy jest określany dopiero w chwili jej tworzenia i rozmiar ten może być automatycznie zwiększany, jeśli tablica jest wypełniona. Obiekty wstawiane do tablicy przechowywane są w wektorze, tzn. w ciągłym obszarze pamięci. Biblioteka zawiera również klasę *TSArrayAsVector* wyprowadzoną z *TArrayAsVector*. Nową właściwością tablicy tej klasy jest to, że przechowywane przez nią obiekty są posortowane. Obiekt wstawiany do tablicy jest umieszczany w miejscu wynikającym z porównania wartości jego klucza z kluczami obiektów już wstawionych, a nie na końcu tablicy. Aby to umożliwić, wymagane jest, aby w klasach wstawianych obiektów zdefiniowano operator < . Metoda klasy *TSArrayAsVector* używana do dodawania obiektów, *Add*, korzysta z tego operatora do określenia pozycji wstawienia obiektu.

Wśród operacji, jakie można wykonywać na danych o elementach systemowych, jest sortowanie ich według jednego z kluczy. Dlatego też obiekty (reprezentujące elementy systemowe) wstawiane do tablic mają kilka kluczy, według których mogą być sortowane. Przykładowo, dla obiektów przechowujących informacje o zadaniach w systemie MS Windows potencjalnymi kluczami są m.in. uchwyt zadania, uchwyt instancji (uchwyt kopii głównego modułu programowego zadania), nazwa itd. [4]. Ponadto niezbędna jest możliwość wyszukania w tablicy obiektu o aktualnym kluczu takim samym jak klucz obiektu podanego jako parametr. W istniejącej w klasach *TArrayAsVector* i *TSArrayAsVector* w metodzie *Find* przy wyszukiwaniu porównuje się całe obiekty. Dlatego z klasy *TSArrayAsVector* wyprowadzono nową klasę *TSSArrayAsVector* mającą możliwość ustalenia nowego porządku przechowywanych obiektów po zmianie klucza sortowania i możliwość wyszukania w tablicy obiektu mającego żądaną wartość aktualnego klucza. Służą do tego nowe metody o nazwach, odpowiednio, *sort* i *find*. Definicja klasy *TSSArrayAsVector* jest następująca:

```
template <class T>
class TSSArrayAsVector : public TSArrayAsVector<T>
{
  public:
    TSSArrayAsVector( int upper, int lower = 0, int delta = 0 );
    void sort();
    void* find( const T* key )
);
```

W implementacji metody *sort* do wykonania operacji sortowania korzysta się ze standardowej funkcji bibliotecznej *qsort*; funkcja *qsort* sortuje elementy tablicy używając algorytmu quicksort. Jednym z parametrów funkcji *qsort* jest adres funkcji używanej w algorytmie do porównywania elementów. Ze względu na metodę *sort* wymaga się, aby obiekty wstawiane do tablicy miały metodę statyczną o nazwie *compare* i jej adres jest przekazywany funkcji *qsort* jako adres funkcji porównującej. Metoda *compare* musi mieć prototyp zgodny z prototypem funkcji porównującej wymaganej przez *qsort* tj.

*int compare(const void *f1, const void *f2);*

Rolę funkcji porównującej może pełnić tylko metoda statyczna, gdyż w zwykłych metodach każdego obiektu (metodach niestatycznych) oczekiwany jest, oprócz parametrów zadeklarowanych przez programistę, niejawny parametr określający adres obiektu, na rzecz którego metoda została wywołana. W języku C++ parametr ten nazwany jest *this*. Kod, wynikiem którego jest przekazanie tego parametru, jest automatycznie tworzony przez translator każdego języka obiektowego. Funkcja *qsort* jest napisana w języku C i dlatego nie może przekazać wywoływanej przez siebie funkcji porównującej takiego parametru.

Podobnie jak metoda *sort* napisana jest metoda *find*. Do wykonania wyszukiwania wykorzystuje ona funkcję biblioteczną *bsearch*. Jako funkcja porównująca również jest używana statyczna metoda *compare*.

Jak widać w deklaracji klasy *TSSArrayAsVector*, jest to klasa szablonowa [1] (i właściwie jej prawidłowa, pełna nazwa to *template <class T> TSSArrayAsVector*). Oznacza to, że można z niej wygenerować klasę o opisanych wyżej własnościach przechowującą obiekty prawie całkowicie dowolnego typu (muszą posiadać operatory < i == oraz metodę statyczną *compare*). Generowanie nie wymaga żadnego nakładu pracy ze strony programisty. Trzeba jedynie tworząc tablicę typu szablonowego użyć w deklaracji jej typu odpowiedniego parametru. Parametr ten jest nazwą typu, który ma być w tablicy przechowywany. Przykładowo definicja

TSSArrayAsVector<TGlobalEntry> tabZadan;

tworzy tablicę *tabZadan*, która może przechowywać obiekty klasy *TGlobalEntry*.

Wyższość użycia klasy kontenerowej nad użyciem zwykłej tablicy polega na:

a) zabezpieczeniu przed użyciem operacji na niewłaściwych danych oraz przed niepowołanym dostępem do danych, a to dzięki zgrupowaniu danych i operacji na nich

w jednej paczce – obiekcie (mechanizm klas) i uniemożliwieniu bezpośredniego dostępu do danych (kapsułkowanie),

b) umożliwieniu tworzenia tablic o dynamicznym, zmiennym rozmiarze; wyższość nad użyciem tablicy alokowanej dynamicznie polega na posiadaniu przez tablice kontenerowe własności automatycznego zwiększania rozmiaru bez potrzeby jego kontroli przez programistę,

c) wyeliminowaniu konieczności dostosowywania implementacji tablicy do przechowywania danych nowych typów; możliwe jest automatyczne generowanie klas dla obiektów-tablic przechowujących dane dowolnego typu, w którym są zdefiniowane pewne operacje,

d) zwiększeniu bezpieczeństwa użycia tablic – niezgodności typów podczas wykonywania operacji na tablicy mogą być wykryte na etapie kompilacji programu.

Oprócz cechy a) właściwej wyłącznie językom zorientowanym obiektowo pozostałe mogą być w jakiś sposób osiągnięte przy tradycyjnym rozwiązaniu problemu przechowywania danych, czyli przez użycie tablicy dynamicznej. O tym, że jest to mniej wygodne, wspomniano w punkcie b), natomiast niemożliwe jest połączenie w tym przypadku cech c) i d). Aby móc przechowywać w tablicy elementy dowolnego typu, procedury operujące na nich, musiałaby założyć, że odwoływano by się do nich za pomocą wskaźników do danych nieznanego typu *void* *. Nie ma wtedy jednak żadnej kontroli typów przechowywanych elementów i tym samym tracona jest cecha d). Próba zapewnienia cechy d) powoduje konieczność wielokrotnej implementacji tablicy.

W językach zorientowanych obiektowo jeszcze jednym sposobem umożliwienia przechowywania w tablicy obiektów dowolnego typu jest wykorzystanie zgodności typu klasy z jej dowolną klasą bazową. Należy zdefiniować klasę bazową zawierającą wymagane aktualnie elementarne operacje i stworzyć tablicę mogącą przechowywać wskaźniki (czy referencje) do obiektów tej klasy. Klasy wszystkich obiektów, które mają być przechowywane w tablicy, muszą być wyprowadzone z tej klasy bazowej. Przy wyjmowaniu elementu z tablicy konieczne jest wykonywanie konwersji do jego rzeczywistego typu. W tym rozwiązaniu istnieje kontrola typów, lecz jest ona tylko częściowa. Istnienie kilku klas mających wspólny korzeń jest potencjalnym powodem błędu polegającego na wstawieniu do tablicy lub usunięciu z niej obiektu niewłaściwej klasy. Przykładowo, jeśli istnieją dwie klasy B i C, obie wyprowadzone z klasy bazowej A, to możliwe jest wstawienie do tablicy wskaźnika do B, a wyjęcie wskaźnika do C:

```
B* pb;
C* pc;
...
tab[2] = pb;
...
pc = (C*)tab[2];  // b³¹d niewykrywalny przez kompilator
```

3. Inteligentne wskaźniki

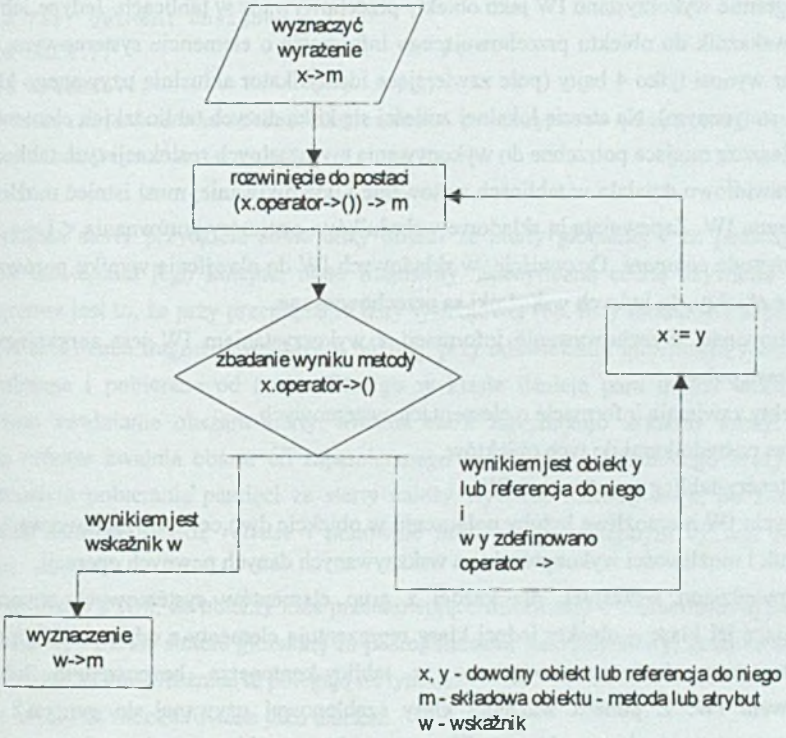
Architektura systemu MS Windows (wersje 3.x) była projektowana w czasie, gdy standardem były komputery z procesorem 80286 i małą ilością pamięci. Jest to system prawie całkowicie 16-bitowy, czego konsekwencją jest to, że pamięć jest dostępna w postaci segmentów wielkości 64 KB, a operacje na danych są wykonywane na liczbach 16-bitowych. Organizacja taka, przy średnich i dużych ilościach danych, stwarza problemy związane z koniecznością manipulowania w programie wieloma segmentami pamięci. Jest ona również mniej efektywna w porównaniu z organizacją 32-bitową.

Na etapie tworzenia programu „WoP" ujawniło się szereg problemów z zarządzaniem danymi o elementach systemowych środowiska MS Windows, problemów charakterystycznych dla programów 16-bitowych. Okazuje się, że niemożliwe jest przechowywanie bezpośrednio w tablicach-kontenerach niektórych grup obiektów zawierających te informacje, ponieważ:

- ich łączny rozmiar może łatwo przekroczyć rozmiar sterty lokalnej programu, czyli w praktyce ok. 35 - 40 KB (np. przy próbie zapamiętania informacji o stercie globalnej potrzeba miejsca na 1500 - 2000 elementów po 36 bajtów każdy),
- konieczna jest czasami realokacja obszaru pamięci przydzielonego tablicy w celu dodania nowych elementów; realokacja oznacza przydzielenie drugiego, większego obszaru i przepisanie tam zawartości obszaru poprzednio zajmowanego, a więc dane nie mogłyby zajmować całej, i tak już niezbyt dużej, sterty lokalnej.

Jedynym rozwiązaniem problemu zbyt małej sterty lokalnej jest umieszczać obiekty z danymi o elementach systemowych na stercie globalnej, a w kontenerach przechowywać tylko wskaźniki do nich. Istnieją w bibliotece kontenery przeznaczone do przechowywania wskaźników, lecz mogą to być tylko wskaźniki domyślnego dla aktualnego modelu pamięci rodzaju. W modelu pamięci medium (używanym powszechnie przy tworzeniu programów pracujących w środowisku MS Windows) są to wskaźniki bliskie, natomiast do odwoływania się do danych na stercie globalnej używane są wskaźniki dalekie. Dlatego zdecydowano się

zastosować szczebel pośredni – inteligentne wskaźniki (ang. smart pointers) w dalszej części artykułu nazywane IW.



Rys. 1.Algorytm interpretowania metody operatorowej *operator* ->
Fig. 1.The algorithm of a interpretation of the operator method *operator* ->

IW są to obiekty, który mają następujące cechy [2]:

- mają zdefiniowany operator selekcji ->,
- operator selekcji jako wynik zwraca wskaźnik na dowolny typ lub na obiekt również zawierający operator selekcji (lub referencję do takiego obiektu),
- oprócz udostępniania wskazywanych danych wykonują dodatkowe operacje.

Algorytm interpretacji metody operatorowej *operator->* pokazany jest na rys. 1. Polega on na rekursywnym wywoływaniu metody *operator->* na rzecz obiektu będącego wynikiem poprzedniego wywołania tej metody aż do chwili, gdy wynikiem tym będzie zwykły wskaźnik. Pierwsze wywołanie odbywa się na rzecz obiektu podanego jawnie. Najczęściej jednym z pól IW jest zwykły wskaźnik. Wyrażenie *x->m* jest interpretowane jako *(x.operator->())->m*. Widać więc, że składnia użycia IW i wskaźników zwykłych jest

identyczna. Jednocześnie IW są też zwykłymi obiektami. Na przykład możliwe jest w (automatycznie wywoływanym!) destruktorze zwalnianie danych, do których wskaźnik jest jednym z atrybutów IW.

W programie wykorzystano IW jako obiekty przechowywane w tablicach. Jedyne ich pole to daleki wskaźnik do obiektu przechowującego informacje o elemencie systemowym, więc ich rozmiar wynosi tylko 4 bajty (pole zawierające identyfikator aktualnie używanego klucza jest polem statycznym). Na stercie lokalnej zmieści się kilka dużych tablic takich elementów i pozostaje jeszcze miejsce potrzebne do wykonywania ewentualnych realokacji tych tablic.

Aby prawidłowo działało w tablicach sortowanie i wyszukiwanie, musi istnieć możliwość porównywania IW. Zapewniają ją składowe wskaźników: operatory porównania < i == oraz statyczna metoda *compare*. Oczywiście, w składowych IW do określenia wyniku porównania są używane obiekty, do których wskaźniki są przechowywane.

Podsumowując, przechowywanie informacji z wykorzystaniem IW jest zorganizowane w sposób następujący:

- obiekty zawierają informacje o elementach systemowych,
- IW są pośrednikami do tych obiektów,
- kontenery-tablice przechowują IW.

Bez użycia IW niemożliwe byłoby połączenie w obiekcie dwu cech, tj. zachowywania się jak wskaźnik i możliwości wykonywania na wskazywanych danych pewnych operacji.

Jak powiedziano wcześniej, dla każdej z grup elementów systemowych stworzono odpowiadającą jej klasę – obiekty jednej klasy reprezentują elementy z odpowiedniej grupy. Obiekty te mogą być przechowywane w tablicy-kontenerze bezpośrednio lub za pośrednictwem IW. Z punktu widzenia klasy szablonowej używanej do generacji klas obiektów-kontenerów te dwa warianty są identyczne i klasa szablonowa nie wymaga żadnych modyfikacji. Jest to możliwe, gdyż zarówno obiekty reprezentujące elementy systemowe, jak i IW mają metody wymagane przez kontener; cześć interfejsu obiektów i cześć interfejsu IW jest taka sama. Metody z tych części interfejsów, metody o takich samych prototypach, nie muszą oczywiście mieć takich samych definicji.

4. Sterta

Jak już zostało powiedziane w p. 3, obiekty przechowujące informacje o elementach systemowych są umieszczane na stercie globalnej. Aby każdy z nich nie wymagał osobnego segmentu, stworzono specjalną klasę *THeapManager*. Obiekt tej klasy pełni rolę zarządcy prostej sterty. Deklaracja klasy *THeapManager* jest następująca:

```
class THeapManager
{
public:
  void far* getMem( unsigned s );
  void mark();
  void release();
  // deklaracje konstruktora, destruktora i  atrybutów przechowujących
  //informacje o stanie sterty
};
```

Zarządca sterty przydziela sobie duży obszar ze sterty globalnej i za pomocą metody *getMem* udostępnia jego kolejne, małe fragmenty. Specyficzną cechą używania tej sterty w programie jest to, że przy przeglądaniu listy systemowej (np. listy modułów) pobierana jest odpowiednia liczba fragmentów sterty, a później, przy odświeżaniu informacji, wszystkie one są zwalniane i pobierane od nowa. Dlatego w klasie istnieje para metod umożliwiająca efektywne zwalnianie obszaru sterty. Metoda *mark* zapamiętuje aktualny szczyt sterty, a metoda *release* zwalnia obszar od zapamiętanego punktu aż do aktualnego szczytu. Przed rozpoczęciem pobierania pamięci ze sterty należy wywołać metodę *mark*, po zakończeniu używania pamięci metodę *release* i ponownie *mark* przed następnym cyklem pobierania pamięci.

W związku z tym, że obiekty klas przechowujące informacje o elementach systemowych są przechowywane na stercie globalnej za pośrednictwem zarządcy sterty, klasy te odróżniają się od zwykłych klas. Różnice te polegają na tym,że:

- klasy są zadeklarowane jako dalekie:

```
class far TGlobalEntry  // słowo kluczowe far po słowie class powoduje
                        // traktowanie klasy jako dalekiej
{
    void far* operator new( size_t size, THeapManager& heapManager );
    void operator delete( void far* object );
    // deklaracje pozostałych metod i atrybutów
};
```

  dzięki czemu prawidłowo działa wywoływanie metod za pomocą dalekich wskaźników do obiektów,
- w klasie jest zdefiniowany operator *new*; operator ten oprócz zwykłego parametru, czyli rozmiaru przydzielanej pamięci, wymaga referencji do zarządcy sterty klasy *THeapManager* i od niego pobiera obszar pamięci dla tworzonego obiektu,
- w klasie jest zdefiniowany operator *delete*; deklaracja operatora składa się tylko z pustego bloku; wywołanie operatora *delete* nie zwalnia przydzielonej pamięci, gdyż

zakłada się, że będzie to wykonane za pomocą metody *release* zarządcy sterty; definicja operatora *delete* jedynie zasłania domyślny, globalny operator *delete*.

Innym rozwiązaniem problemu przydzielania pamięci dla obiektów tworzonych na stercie globalnej byłoby użycie standardowej funkcji *farmalloc*. W implementacji funkcji *farmalloc* w pakiecie Borland C++ cały segment pamięci (64 KB) pobierany jest bezpośrednio od systemu operacyjnego i w nim są sukcesywnie alokowane kolejne, żądane obszary, aż do jego wyczerpania. Wyjątkiem są żądania alokowania obszarów pamięci większych niż 4 KB, ponieważ na ich poczet są specjalnie pobierane od systemu segmenty żądanej wielkości.

Użycie funkcji *farmalloc* do przydzielania pamięci dla obiektów tworzonych na stercie globalnej byłoby możliwe, lecz zastosowanie własnego zarządcy sterty, choć funkcjonalnie identyczne, jest efektywniejsze ze względu na specyficzny przebieg alokowania i zwalniania obszarów pamięci, tzn. naprzemienne pobieranie fragmentów sterty i zwalniania całych ich grup.

Klasa implementująca stertę zachowuje się tak samo jak typ wbudowany:

- istnieją standardowe metody tworzenia i usuwania zmiennych tego typu,
- typ ten jest znany kompilatorowi i jest przez niego odróżniany od innych typów,
- do zmiennych tego typu stosuje się zwykłe reguły zasięgu i metody przekazywania parametrów.

Użycie do implementacji sterty techniki modularyzacji może polegać na stworzeniu grupy funkcji umożliwiających tworzenie stert, usuwanie ich i wykonywanie operacji na nich. Wynikiem utworzenia jest identyfikator sterty – wartość pewnego typu. Wartość ta używana jest we wszystkich następnych operacjach na stercie. Typ tej wartości można utworzyć korzystając z deklaracji *typedef*, np. :

```
typedef int id_heap;
```

Tego rodzaju typy ewidentnie różnią się od typów wbudowanych i nie mają ich, wymienionych wcześniej, zalet. Zmiennej tego typu nie można traktować jak sterty, ale tylko jako identyfikator sterty.

Stworzenie klas implementujących pewien rodzaj sterty daje możliwość łatwego tworzenia wielu stert jednocześnie. Zamknięcie w jednym obiekcie informacji o położeniu i stanie sterty oraz operacji na niej czyni istniejące jednocześnie sterty całkowicie od siebie odizolowanymi.

5. Podsumowanie

W artykule przedstawiono rozwiązania trzech problemów związanych z zarządzaniem pamięcią operacyjną, jakie napotkano podczas realizacji pracy [3]; pochodzą one z części

programu odpowiedzialnej za przechowywanie informacji. Problemy te to użycie obiektów klasy kontenerowej do przechowywania innych obiektów, zastosowanie IW pośredniczących w dostępie do struktur danych przechowujących informacje oraz zaimplementowanie struktury danych – sterty – w postaci obiektu.

Klasy kontenerowe są strukturami danych dostępnymi w bibliotekach większości obecnych na rynku kompilatorów języka C++. Są wygodniejsze w użyciu od zwykłych tablic dzięki swemu dynamicznemu rozmiarowi oraz dzięki posiadaniu dużej liczby metod ułatwiających wykonywanie operacji na przechowywanych danych. Ponadto, dzięki możliwości wykorzystania dziedziczenia, są bardzo elastyczne i łatwo dają się dostosować do specyficznych wymogów.

Inteligentne wskaźniki rozszerzają pojęcie wskaźnika pozwalając na automatyzację niektórych operacji na nim (jak np. jego usuwanie).

Implementacja sterty w postaci obiektu zwiększa bezpieczeństwo jej używania, pozwala na łatwe tworzenie wielu jej egzemplarzy, upodabnia ją do wbudowanych typów danych czyniąc jej użycie jednolite z użyciem innych zmiennych czy struktur danych.

W przedstawionych rozwiązaniach szeroko korzysta się z możliwości, jakie stwarza technika obiektowa i język C++. Porównując je z możliwymi rozwiązaniami zgodnymi z paradygmatem programowania strukturalnego pokazano, jakie można odnieść korzyści przy stosowaniu udogodnień wprowadzonych przez technikę obiektową, takich jak klasy, dziedziczenie, metody operatorowe, przeciążanie metod, szablony i kapsułkowanie.

LITERATURA

[1] Stroustrup B.: Język C++. Wydawnictwa Naukowo-Techniczne, Warszawa 1994.
[2] Ellis M. A., Stroustrup B.: The Annotated C++ Reference Manual. Addison-Wesley Reading, MA, 1990.
[3] Jasiński J.: Inspekcja zawartości pamięci środowisku MS Windows. Praca magisterska, Politechnika Śląska, Instytut Informatyki, Gliwice 1995.
[4] Klein M.: Przewodnik po bibliotekach DLL i sposobach zarządzania pamięcią dla programujących w środowisku MS Windows. Intersoftland / Prentice Hall, Warszawa, Polska / Hemel Hempstead, England 1994.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 4 marca 1996 r.

Abstract

We present three solutions related to memory management. Facilities provided by object-oriented\ technology and C++ language such as classes, inheritance, overloading, encapsulation and templates are used in them. We describe a container class used to storing objects, smart pointers as intermediary between container and objects with data and implementation of data structure called heap as object. We compare advantages of OO solutions to traditional ones.