Zapisywanie stanu obiektów

Biblioteka boost::serialization

Zapisywanie obiektów do strumienia bajtów lub znaków wykorzystujemy do przesyłania informacji pomiędzy aplikacjami lub modułami oraz do odtwarzania stanu aplikacji po ponownym jej uruchomieniu. Poniższy tekst omawia bibliotekę, która upraszcza to zadanie.

Dowiesz się:

- Dlaczego przekształca się obiekty do postaci szeregowej
- Jakie są sposoby, inne niż baza danych, trwałego przechowywania stanu aplikacji
- Jak używać biblioteki boost::serialization

Powinieneś wiedzieć:

- Jak pisać proste programy w C++
- Co to są klasy, obiekty, dziedziczenie oraz wskaźniki

W programowaniu obiektowym, proces przekształcania obiektów, tj. instancji określonych klas, w strumień bajtów, z zachowaniem aktualnego stanu obiektu nazywamy serializacją. Procesem odwrotnym do serializacji jest deserializacja. Jest to odczytanie wcześniej zapisanego strumienia danych i odtworzenie na tej podstawie obiektu klasy. Serializacja może być wykorzystywana do:

- Trwałego zapisywania obiektów, w celu późniejszego ich odczytu (nawet po ponownym uruchomieniu aplikacji);
- Przesyłania obiektów do innego modułu (np. napisanego w innym języku programowania), innego procesu (innej aplikacji), innego komputera poprzez sieć;
- Wyświetlania stanu obiektu, np. przy uruchamianiu programu lub znajdowaniu błędów.

Obiekt w postaci szeregowej, czyli obiekt zserializowany może być przechowywany w wielu różnych formatach. Najbardziej popularne to ciąg binarny, ciąg znaków, XML, JSON (patrz ramka).

Wiele popularnych języków posiada wsparcie dla serializacji w rdzeniu języka lub w bibliotece standardowej (C#, Java, Python, PHP, Perl, Ruby). C++ zawiera klasy strumieni oraz konwersje pomiędzy strumieniem bajtów lub znaków, a wbudowanymi typami danych oraz możliwość rozszerzania tego mechanizmu na obiekty klas definiowanych przez użytkownika (patrz ramka). Ponieważ język (i biblioteka standardowa) zawiera wiele dodatkowych konstrukcji (wskaźniki, referencje, kolekcje, sprytne wskaźniki omówione w SDJ 1/2010, itd) wbudowane mechanizmy konwersji obiektów na strumienie bajtów są w praktyce zbyt ubogie. W artykule będziemy pokazywali rozwiązania wykorzystujące bibliotekę boost::serialization. Biblioteki eksperymentalne boost są zbiorem wielu przydatnych rozszerzeń biblioteki standardowej C++. Wiele z nich jest dodawanych do kolejnych wersji standardu, dlatego warto je znać. Innym narzędziem, które warto brać pod uwagę w przedstawionych przypadkach jest Google Protocol Buffers.

Konwersje pomiędzy obiektem a strumieniem znaków (bajtów)

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Przykłady wykorzystują biblioteki boost (www.boost.org). Aby poprawnie je skompilować należy dodać odpowiednie zależności wykorzystywane podczas konsolidacji; dla konsolidatora g++ należy dodać opcje: `-lboost_serialization`; dla konsolidatora Visual Studio (program link) biblioteki boost są dodawane automatycznie. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze.

Przekształcenie obiektu klasy na strumień bajtów powinno uwzględniać następujące zagadnienia:

1. zapis składowych w odpowiedniej kolejności;
2. odpowiednie zapisywanie wskaźników i referencji;
3. zapis rzeczywistych obiektów pomimo posługiwania się wskaźnikiem lub referencją do klasy bazowej;
4. odtwarzanie liczników przy posługiwaniu się sprytnymi wskaźnikami;
5. obsługę kontenerów standardowych.

Pierwszy punkt może być realizowany przy pomocy biblioteki standardowej, mamy konwersje pomiędzy ciągiem znaków a typami wbudowanymi, możemy takie konwersje tworzyć dla typów użytkownika. Odpowiednie traktowanie wskaźników (i referencji) jest bardziej skomplikowane, niż zapis adresu do strumienia znaków czy bajtów, ponieważ zazwyczaj chodzi nam o zapis obiektu, który jest wskazywany. Jeżeli ten sam obiekt jest wskazywany przez wiele wskaźników, to chcemy, aby serializować tylko jedną kopię. Dodatkowo czasami posługujemy się wskaźnikiem (lub referencją) do klasy bazowej, zaś obiekt jest klasy pochodnej (stosujemy polimorfizm czasu wykonania). Wtedy, gdy chcemy taki wskaźnik (lub referencję) zserializować, to chodzi nam o wskazywany obiekt odpowiedniego typu. Dla sprytnych wskaźników chcielibyśmy zapisywać wskazywane obiekty oraz informacje pomocnicze (licznik odniesień). Wygodnie byłoby także mieć udogodnienia, które pozwalają w prosty sposób zapisywać tablice obiektów i kontenery standardowe.

Popularne formaty przechowywania zserializowanych obiektów

Binarny – zwykły ciąg zer i jedynek, najbardziej oszczędny pamięciowo spośród formatów tutaj wymienionych, nieczytelny dla człowieka.
Tekstowy - podobny do binarnego, oszczędny pamięciowo, czytelny dla człowieka, jeżeli struktury obiektów są proste.
JSON (*JavaScript Object Notation*) - format tekstowy, pozwala przechowywać drzewiaste struktury obiektów, czytelny dla człowieka, bardziej oszczędny pamięciowo niż XML. Przykład:

```
{ "osoba": {
     "imie" : "Donald",
     "nazwisko" : "Knuth",
     "wiek" : 73
   }
}
```

XML (e*Xtensible Markup Language*)- format tekstowy, pozwala przechowywać drzewiaste struktury obiektów, czytelny dla człowieka, posiada możliwość sprawdzania poprawności (walidacja), zajmuje dużo pamięci. Przykład:

```
<osoba>
   <imie>Donald</imie>
   <nazwisko>Knuth</nazwisko>
   <wiek>73</wiek>
</osoba>
```

Klasy strumieni w C++

Strumienie wchodzą w skład standardowej biblioteki C++ i są odpowiedzialne za obsługę wejścia i wyjścia. Strumienie dostarczają konwersji bezpiecznych ze względu na typ pomiędzy plikiem, napisem, standardowym wejściem/wyjściem dla typów wbudowanych oraz dla typów standardowych (np. `std::string`). Istnieje możliwość dodawania konwersji dla typów użytkownika, należy zdefiniować odpowiednie operatory. Hierarchia klas strumieni została pokazana na Rysunku 1. Każda z przedstawionych tam klas jest szablonem, którego parametrem jest typ znaku

Boost::serialization

Biblioteka boost::serialization wspiera przekształcanie obiektów na postać szeregową (ciąg bajtów lub znaków). Wykorzystuje ona szablony i meta-programowanie (patrz SDJ 11/2010 oraz 12/2010), jest wydajna i prosta w użyciu. Rozdzielona jest tam warstwa przekształcania obiektów na postać szeregową (nagłówki `boost/serialization`) oraz warstwa formatowania archiwum (nagłówki `boost/archive`), przy czym istnieje wsparcie dla postaci binarnej, tekstowej i XML, do zapisu można wykorzystywać strumienie dla znaków typu `char` bądź `wchar_t`.

Przykład użycia serializacji pokazano na Listingu 1. Obiekty klasy autonomicznej `Foo` mogą być zapisywane i odczytywane, odpowiednie konwersje są zawarte w metodzie `serialize`. Jest to metoda szablonowa, która podaje sposób zapisu i odczytu z archiwum. Wykorzystywane są operatory `>>`, `<<` oraz `&`, dwa pierwsze oznaczają odpowiednio odczyt z archiwum oraz zapis do archiwum, operator & oznacza odczyt bądź zapis, w zależności od aktualnie wybranej operacji. Na Listingu 1 wykorzystano właśnie ten operator serializując składowe. Jeżeli zdefiniujemy metodę `serialize`, to możemy wykorzystać archiwum, tak jak podano na Listingu 1 w funkcji `main`. Zapisujemy tam postać szeregową obiektu `f` w strumieniu napisowym `oss`, następnie napis zawierający tę postać wykorzystujemy, aby odtworzyć obiekt. Na przedstawionym wydruku używamy archiwum tekstowego (`text_oarchive`, `text_iarchive`).

Listing 2 zawiera podobną funkcjonalność, ale wykorzystuje archiwum XML. W tym wypadku koniecz-

Listing 1. *Serializacja i deserializacja za pomocą biblioteki boost::serialization*

```
class Foo { //przykładowa klasa
public:
   Foo() : id_(0), name_("") { }
   Foo(int id, const std::string& name) : id_(id), name_(name) { }
   ~Foo() { }
   template<class Archive> void serialize(Archive & ar, const unsigned int /* file_version */){
      ar & id_;
      ar & name_;
   }
private:
   int id_;
   std::string name_;
};

int main() {
   std::ostringstream oss;
   {
      Foo f(1,"ALA"); //utworzenie obiektu
      boost::archive::text_oarchive oa(oss); //utworzenie archiwum
      oa << f; //zapis obiektu f w strumieniu oss - serializacja
   }
   std::cout << oss.str() << std::endl; //wydruk archiwum
   std::istringstream iss(oss.str());
   {
      boost::archive::text_iarchive ia(iss);
      Foo f;
      ia >> f; //obiekt f jest odtworzony na podstawie postaci szeregowej - deserializacja
      cout << f << endl;
   }
   return 0;
}
```

Listing 2. *Serializacja przykładowego obiektu do XML; wymagane jest podanie nazw tagów*

```
class Foo {
public:
   Foo() : id_(0), name_("") { }
   Foo(int id, const std::string& name) : id_(id), name_(name) { }
   ~Foo() { }
private:
   friend class boost::serialization::access;    // dostęp do metody serialize dla biblioteki serialization
   template<class Archive> void serialize(Archive & ar, const unsigned int /* file_version */){
      ar & boost::serialization::make_nvp("id", id_);
      ar & boost::serialization::make_nvp("name", name_);
   }
private:
   int id_;
   std::string name_;
};
```

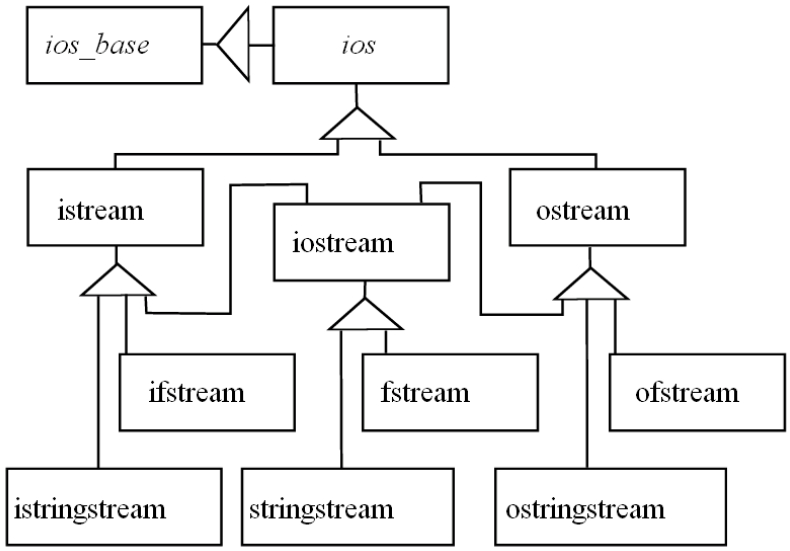
ne jest dostarczenie nazwy odpowiednich tagów (co robi funkcja pomocnicza `make_nvp`). Jeżeli nie chcemy tych nazw wymyślać, to można posłużyć się makrodefinicją `BOOST_SERIALIZATION_NVP`, która nazwę taga XML pobiera z nazwy składowej. Przykładowy XML produkowany przez Listing 2 jest pokazany na Listingu 3.

Zazwyczaj nie chcemy, aby funkcje związane z serializacją były wołane jawnie przez inne obiekty aplikacji, dlatego umieszczamy je w sekcji prywatnej. Ponieważ są one przeznaczone dla biblioteki boost::serialization, dodajemy możliwość ich wołania stosując zaprzyjaźnioną klasę `boost::serialization::access`. Pokazano to na Listingu 2.

Odtworzenie obiektu jest możliwe nawet po zamknięciu aplikacji, jeżeli zserializowany obiekt zapiszemy do pliku (używając strumienia `ofstream`), to możemy go odczytać (za pomocą strumienia `ifstream`).

Zapis wskaźników

W języku C++ możemy posługiwać się uchwytami do obiektu (najprostsze uchwyty to wskaźniki i referencje), pozwala to m.in. na dostęp do tego samego obiektu z różnych miejsc, robienie „płytkiej kopii" (patrz SDJ 3/2010). Przy serializacji wskaźnika (lub innego uchwytu) nie jest istotna jego wartość (przechowywany adres), ale wskazywany obiekt oraz fakt, że ta sama wartość wskaźnika oznacza ten sam obiekt, wskaźnik jest tutaj identyfikatorem. Biblioteka boost::serialization przekształcając uchwyt (wskaźnik, referencja itp.) w strumień bajtów lub znaków zapisuje wskazywany obiekt, a nie adres. Dodatkowo adresy zapisywanych obiektów są rejestrowane i traktuje się je jak identyfikatory obiektów, więc jeżeli serializujemy złożoną strukturę obiektów, to każdy obiekt zostanie zapisany tylko raz, pomimo tego, że może istnieć do niego odnośnik (uchwyt, wskaźnik) w różnych miejscach.



Rysunek 1: *Klasy ios_base oraz ios dostarczają cech wspólnych dla różnych rodzajów strumieni, m.in. sposób formatowania, stan strumienia, itp., klasa istream jest klasą bazową dla strumieni wejściowych (obiekt typu istream to np. cin), klasa ostream jest klasą bazową dla strumieni wyjściowych (obiekt cout, crr jest typu ostram), iostram jest klasą bazową dla strumieni które są jednocześnie wejściowe i wyjściowe (zastosowano tutaj dziedziczenie wielobazowe). Klasy ifstram, ofstram, fstream reprezentują strumień związany z plikiem, natomiast istringstream, ostringstream, stringstream to strumienie związane z napisem (obiektem typu string).*

Odtwarzając złożony obiekt (deserializując) wskazywane obiekty będą tworzone w innym miejscu pamięci, ale powiązania (wskaźniki) będą prawidłowe, tzn. będą wskazywały na odpowiednie obiekty. Przykład pokazany na Listingu 4 przedstawia zapis obiektów typu Mapping do strumienia bajtów. Obiekt `Mapping` zawiera listę obiektów `Obj` (składowa `l_`) oraz wektor wskaźników do tych obiektów (składowa `v_`). Po odtworzeniu obiektu elementy w liście będą przechowywane prawdopodobnie w innym miejscu, pod innymi adresami, ale powiązanie będzie prawidłowe, inne adresy będą także w wektorze.

Listing 4 pokazuje użycie wbudowanych udogodnień do zapisu kolekcji: boost::serialization pozwala zapisywać kontenery standardowe (obiekty typu `vector`, `list`, `deque`, `set`, `map`, `multiset`, `multimap`).

Listing 3. *Przykład napisu wygenerowanego podczas serializacji obiektu z Listingu 2*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
<Foo class_id="0" tracking_level="0" version="0">
   <id>1</id>
   <name>ALA</name>
</Foo>
```

Dodatkowo przykład ten pokazuje użycie algorytmów z biblioteki standardowej oraz nienazwanych obiektów funkcyjnych tworzonych przez `boost::bind` (patrz SDJ 1/2010). Szablon `bind` jest częścią standardu C++11, umieszczono go w standardowej przestrzeni nazw.

Podczas deserializacji wskaźnika po raz pierwszy tworzony jest obiekt na stercie i jego adres jest wiązany z identyfikatorem, przechowywanym w strumieniu bajtów. Każda kolejna deserializacja tego wskaźnika powoduje ustawienie zapamiętanego adresu. Ten mechanizm pozwala odtworzyć strukturę obiektów powiązanych wskaźnikami. Dodatkowo biblioteka dostarcza funkcji, które serializują/deserializują sprytne wskaźniki z licznikiem odniesień `boost::shared_ptr` (SDJ 1/2010), które także zostały dołączone do biblioteki standardowej (C++11).

Klasa, która będzie deserializowana powinna dostarczać konstruktora bezparametrowego. Składowe stałe, czyli składowe, dla których nie możemy zmieniać wartości po utworzeniu obiektu, możemy zapisywać do archiwum bez dodatkowych zabiegów, natomiast odczyt wymaga jawnego rzutowania (`const_cast`). Jawne rzutowanie jest wymagane, ponieważ zmieniamy wartość składowej dla już utworzonego obiektu. Ze względu na konieczność rzutowania składowych stałych, zapis (serializacja) musi różnić się od odczytu (deserializacja), więc dostarczamy dwóch metod: `save` i `load`. Przykład pokazano na Listingu 5. Metoda `serialize` zawiera wołanie funkcji, która uruchomi metodę `load` przy odczycie oraz metodę `save` przy zapisie. Metoda `save` jest stała (nie zmienia obiektu serializowanego), co pozwala wykrywać niektóre błędy. Metoda `load` zmienia stan obiektu.

Listing 4. *Serializacja wskaźników i kolekcji wskaźników*

```
class Obj { //przykładowa klasa
public:
   Obj() : id_(0) {    }
   Obj(int i) : id_(i) {    }
   Obj(const Obj& o) : id_(o.id_) {    }
   ~Obj() {    }
   bool operator==(const Obj& o) const { return o.id_ == id_; }
   template<class Archive>
   void serialize(Archive & ar, const unsigned int /* file_version */){
      ar & id_;
   }
private:
   int id_;
};
typedef std::list<Obj> Objs;
typedef Obj* IddObj;
IddObj getIdd(const Obj& o) { return const_cast<Obj*>(&o); }
class Mapping {//klasa przechowuje obiekty oraz powiązania pomiędzy nimi (wskaźniki)
public:
   Mapping() {}
   Mapping(const Objs& l) : l_(l) {
      //tworzy powiązania na podstawie obiektów
      std::transform( l_.begin(), l_.end(), std::back_inserter(v_), boost::bind(getIdd, _1) );
}
   template<class Archive>
   void serialize(Archive & ar, const unsigned int /* file_version */){
      ar & l_; //zapisuje lub odczytuje listę
      ar & v_; //zapisuje lub odczytuje wektor
   }
private:
   Objs l_;
   std::vector<IddObj> v_;
};
```

Rozdzielenie odczytu i zapisu możemy wykorzystać do przechowywania obiektów zarządzanych przez sprytne wskaźniki. W przykładzie pokazanym na Listingu 6, pokazano serializację i deserializację klasy, która wykorzystuje sprytne wskaźniki typu `auto_ptr`. Wskaźniki takie wchodzą one w skład biblioteki standardowej C++ (C++03), zapewniają one automatyczne niszczenie obiektów utworzonych na stercie w przypadku, gdy tylko jeden wskaźnik wskazuje na obiekt, patrz SDJ 11/2009. Serializacja `auto_ptr` oznacza serializację zwykłego wskaźnika, czyli za pierwszy razem zapis wskazywanego obiektu, zaś zapis identyfikatora, gdy ten sam wskaźnik zostanie napotkany po raz kolejny. Odczyt (deserializacja) obiektu typu `auto_ptr` to odtworzenie zwykłego wskaźnika, czyli utworzenie wskazywanego obiektu na stercie lub pobranie adresu z modułu rejestrującego identyfikatory obiektów, a następnie ustawienie stanu tego obiektu (metoda `reset`).

Zamiana identyfikatora obiektu przechowywanego w archiwum (strumieniu bajtów) a adresem obiektu na stercie, stosowana do odtwarzania struktur obiektów powiązanych wskaźnikami pozwala odtwarzać struktury obiektów powiązanych wskaźnikami. Należy jednak zdawać sobie sprawę, że istnieją przypadki, gdy potrzebna jest interwencja programisty. Automatyczny mechanizm zawodzi, jeżeli obiekty są przechowywane w innym miejscu pamięci niż odczytywany obiekt. Taka sytuacja ma miejsce, gdy stosujemy kolekcje standardowe, które przechowują kopie obiektów. Odpowiedni przykład pokazano na Listingu 7. Mapowanie pomiędzy identyfikatorem obiektu a adresem wymaga wykorzystania metody `reset_object_address`. Jeżeli będziemy używać funkcji `load` pokazanej na tym listingu, to odtwarzając złożony obiekt (deserializując) wskazywane obiekty będą tworzone w innym miejscu pamięci, ale powiązania (wskaźniki) będą prawidłowe, tzn. będą wskazywały na odpowiednie obiekty. Wbudowane udogodnienia do odczytu/zapisu kolekcji korzystają z przedstawionej metody, dlatego przykład na Listingu 4 jest poprawny.

Listing 8 zawiera przykład serializacji i deserializacji listy list obiektów oraz kontenera wskaźników do tych obiektów. Udało się uniknąć stosowania metody `reset_object_address` stosując wbudowane metody dla kontenerów, ponieważ wczytujemy obiekty do elementów kontenera (bez kopiowania). Niestety, zapis/odczyt listy za pomocą funkcji dostarczanych przez boost::serialization nie działa poprawnie w tym przypadku.

Serializacja obiektów klas pochodnych

Jeżeli serializujemy/deserializujemy klasę pochodną (klasę z hierarchii klas), to nie zaleca się jawnie wołać metody `serializable, load`, czy `save` z klasy bazowej, lepiej użyć `boost::serialization::base_object`. Odpowiedni przykład pokazano na Listingu 9. Biblioteka boost::serialization (za pomocą meta-programowania,

Listing 5. *Przykład zapisu/odczytu stałego wskaźnika (składowa `ptr_`). Wymagane jest rzut na stały obiekt przy odczycie, więc metody odczytu i zapisu są rozdzielone.*

```
class Goo;

class Foo {
   //pozostałe metody klasy
private:
   friend class boost::serialization::access;
   template<class Archive> void save(Archive & ar, const unsigned int /* file_version */) const {
      ar << boost::serialization::make_nvp("Goo", ptr_ );
   }
   template<class Archive> void load(Archive & ar, const unsigned int /* file_version */) {
      Goo* p;
      ar >> boost::serialization::make_nvp("Goo", p); //odczyt wskazywanego obiektu
      ptr_ = const_cast<const Goo*>(p); //modyfikacja stałej składowej
   }
   template<class Archive> void serialize( Archive &ar, const unsigned int file_version ){
      boost::serialization::split_member(ar, *this, file_version);
   }
private:
   const Goo* ptr_;
};
```

patrz SDJ 12/2009) odczytuje rzeczywisty typ obiektu i woła odpowiednią metodę do zapisu/odczytu, nawet gdy posługujemy się wskaźnikiem do klasy bazowej, zaś obiekt jest klasy pochodnej.

Biblioteka automatycznie rejestruje każdy typ, który po raz pierwszy jest serializowany / deserializowany. Jeżeli obiekt danego typu jest używany ponownie, to wykorzystuje się wcześniejsze rejestracje. Posługując się wskaźnikami i hierarchią klas, czasami istnieje potrzeba ręcznej rejestracji odpowiedniej klasy pochodnej przy odczycie obiektu. W tym celu dostarczono funkcję `register_type`, patrz przykład na Listingu 10. Innym sposobem, który możemy wykorzystać w tym przypadku, jest zapis pustych obiektów klasy pochodnej.

Nieinwazyjna serializacja/deserializacja

Nie musimy modyfikować interfejsu klas, których obiekty będziemy serializować i deserializować. Zamiast dodawać metody `serializable` (lub `load` i `save`) możemy dodać funkcje, które będą wołane podczas zapisu i odczytu obiektów. Przykład takiego rozwiązania jest pokazany na Listingu 11. Serializowana klasa musi dostarczać metod, które pozwalają odczytać i zapisać jej stan, dlatego nie zawsze jest to wygodne

Listing 6. *Serializacja obiektów, które posiadają składowe typu auto_ptr*

```
class Goo;

class Foo {
public:
   //pozostałe metody klasy
private:
   friend class boost::serialization::access;

   template<class Archive> void save(Archive & ar, const unsigned int /* file_version */) const {
      const Goo* const p = ptr_.get();
      ar << p; //zapisuje wskaźnik, za pierwszym razem obiekt, później identyfikator obiektu
   }
   template<class Archive> void load(Archive & ar, const unsigned int /* file_version */) {
      Goo* p;
      ar >> p; //odtwarza wskaźnik (tworzy obiekt lub odczytuje nowy adres obiektu)
      ptr_.reset(t);
   }
   template<class Archive> void serialize( Archive &ar, const unsigned int file_version ){
      boost::serialization::split_member(ar, *this, file_version);
   }
private:
   std::auto_ptr<Goo> ptr_;
};
```

Listing 7. *Modyfikacja powiązania pomiędzy identyfikatorem a adresem obiektu*

```
template<class Archive>
void load(boost::basic_iarchive & ar, const unsigned int version ) {
   int size;
   ar >> size; //potrzebne do implementacji pętli
   for(int i =0; i < size; ++i) {
      Obj o;
      ar >> o; //teraz obiekt będzie mia adres &o;
      l_.push_back(o); //ale  chcemy, by obiekt miał adres &l_.back()
      ar.reset_object_address( &l_.back(), &o ); //rejestracja innego adresu
   }
}
```

rozwiązanie. Jeżeli jednak możemy stan obiektu modyfikować z zewnątrz, to możemy także obiekty takie serializować/deserializować za pomocą biblioteki serialization, tworzymy funkcje `serialize` lub `load` i `save` z odpowiednimi argumentami. Proszę zwrócić uwagę, że funkcje te umieszczamy w przestrzeni nazw boost::serialization.

Wykorzystywanie archiwum w różnych wersjach aplikacji

Przy serializacji mamy dostępny dodatkowy parametr - wersję archiwum. Dotychczas nie używaliśmy tego parametru, ale jeżeli chcemy pozwolić na serializację/deserializację obiektów w różnych wersjach naszej aplikacji, np. wczytywać wcześniej utworzone strumienie zawierające obiekty, to parametr ten może okazać się pomocny. Pozwala to m.in. na częściową zmianę struktury obiektów bez utraty kompatybilności wstecz.

Jeżeli nie podejmujemy dodatkowych działań, to numer wersji jest ustawiany na 0. Możemy to zmienić dla danego typu (danej klasy) za pomocą makrodefinicji

```
BOOST_CLASS_VERSION(Foo, 1)
```

Listing 8. *Serializacja listy list, potrzebne są dodatkowe zabiegi, aby adresy były odtwarzane prawidłowo*

```
class Foo;
typedef std::list<Foo> List;
typedef std::list<List> LList;
typedef std::list<const Foo*> ListId;

class Goo {
private:
   template<class Archive>
   void save(Archive & ar, const unsigned int /* file_version */) const {
      //ar << boost::serialization::make_nvp("list", l_);
      unsigned int size = l_.size();
      ar << boost::serialization::make_nvp("size",size);
      for(LList::const_iterator i = l_.begin(); i != l_.end(); ++i) {
         const List& l = *i;
         ar << boost::serialization::make_nvp("list",l);
      }
      ar << boost::serialization::make_nvp("id", id_);
   }
   template<class Archive>
   void load(Archive & ar, const unsigned int /* file_version */) {
      unsigned int size;
      ar >> boost::serialization::make_nvp("size", size );
      l_.clear();
      for(; size > 0; --size) {
         l_.push_back( List() );
         List& l = l_.back();
         ar >> boost::serialization::make_nvp("list", l ); //deserializacja utworzonego elementu listy
      }
      id_.clear();
      std::list<Foo*> idl;
      ar >> boost::serialization::make_nvp("id", idl); //odczyt listy wskaźników
      for(std::list<Obj*>::const_iterator i = idl.begin(); i != idl.end(); ++i) {
         id_.push_back( *i ); //automatycznie rzutowane na stałe wskaźniki
      }
   }
   LList l_;
   ListId id_;
};
```

Listing 9. *Serializacja /deserializacja obiektów klas pochodnych.*

```
class Base {
public:
   Base() : id_(0) { }
   Base(const Base& o) : id_(o.id_) { }
   virtual ~Base() { }
private:
   friend class boost::serialization::access;

   template<class Archive> void serialize( Archive &ar, const unsigned int /* file_version */ ){
      ar & boost::serialization::make_nvp("id_", id_);
   }
private:
   int id_;
};

class Der : public Base {
public:
   Der() : der_(0) {    }
   Der(const Der& o) : der_(o.der_) {    }
   virtual ~Der() {    }
private:
   friend class boost::serialization::access;

   template<class Archive> void serialize( Archive &ar, const unsigned int file_version ){
      ar & boost::serialization::make_nvp("base", boost::serialization::base_object<Base>(*this) );
      ar & boost::serialization::make_nvp("der", der_);
   }
private:
   int der_;
};
```

Listing 10. *Przy zapisie/odczycie wskaźników do obiektów polimorficznych archiwum musi zarejestrować odpowiedni typ. Jeżeli nie jest to robione automatycznie, trzeba wołać funkcję register_type, aby poprawnie zapisać i odczytać taki wskaźnik.*

```
//Wykorzystano definicje klasy Base i Derived pokazane na Listingu 9
class Foo {
public:
   Base() : id_(0) { }
   Base(const Base& o) : id_(o.id_) { }
   virtual ~Base() { }
private:
   friend class boost::serialization::access;
   template<class Archive>
   void serialize( Archive &ar, const unsigned int file_version ){
      ar.template register_type<Der>(); //rejestruje typ Der
      ar & boost::serialization::make_nvp("ptr", ptr_);
   }
private:
   Base* ptr_;
};
```

Listing 11. *Serializacja/deserializacja klasy bez zmiany jej kodu*

```
struct Person {//klasa pozwala modyfikować składowe
   string name;
   int age;
};

namespace boost {
   namespace serialization {
      template<class Archive>
      void serialize(Archive & ar, Person & p, const unsigned int /* version */ ) {
         ar & p.name;
         ar & p.age;
      }
   } // namespace serialization
} // namespace boost
```

gdzie pierwszym argumentem jest nazwa klasy, zaś kolejnym - numer wersji. W takim przypadku obiekty danej klasy będą zachowywane z dostarczonym numerem wersji. Numer wersji przy odczycie (metoda lub funkcja `load`) jest zgodny z numerem zapisanym podczas serializacji. Możemy tutaj tworzyć obiekty w różny sposób, w zależności od numeru wersji.

Podsumowanie

Serializacja upraszcza proces przesyłania informacji pomiędzy modułami lub aplikacjami.

Jeżeli stosujemy serializację do trwałego zapisywania obiektów, to alternatywą jest wykorzystanie systemu zarządzania bazą danych (DBMS). System ten zapewnia trwałość oraz szereg innych właściwości (np. transakcje), ale ponieważ najbardziej popularnym modelem danych jest model relacyjny, zaś języki programowania najczęściej wspierają podejście obiektowe, wymagana jest konwersja obiektów na rekordy. Należy więc dostarczyć odpowiedni kod, np. używając biblioteki umożliwiającej stosowanie „zanurzonego SQL" (embeded SQL), albo wspierając się bibliotekami do mapowania relacyjno - obiektowego (ORM - Object Relational Mapping). Dodatkowy nakład pracy nie zawsze jest uzasadniony, jeżeli chcemy tylko przechować obiekt i później odtworzyć jego stan. Dodatkowo, gdy format jest czytelny dla człowieka (JSON, XML, czasami tekstowy), to można ręcznie modyfikować archiwa (pliki zawierające serializowane obiekty), lub wykorzystywać je do diagnozowania stanu aplikacji lub wyszukiwania błędów.

ROBERT NOWAK
Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji wykorzystujących algorytmy sztucznej inteligencji i fuzji danych. Autor biblioteki faif.sourceforge.net. Programuje w C++ od ponad 15 lat.
Kontakt z autorem:rno@o2.pl

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost, w tym boost::serialization;
- *http://code.google.com/p/protobuf/* – dokumentacja Google Protocol Buffers (alternatywne rozwiązanie).

Więcej w książce

Omówienie współcześnie stosowanych technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, opisano w książce Robert Nowak, Andrzej Pająk ,,Język C++: mechanizmy, wzorce, biblioteki'', BTC 2010.