Zaawansowane CPP/Wykład 2: Programowanie uogólnione

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wprowadzenie

W poprzednim wykładzie wprowadziłem pojęcia szablonów funkcji i klas. Są to bardzo ważne konstrukcje języka C++ dające programistom bezpośrednie, czyli z poziomu języka, wsparcie dla tworzenia uogólnionych funkcji i typów (nazywanych też funkcjami lub typami parametryzowanymi). Uogólnienie polega na tym, że za jednym zamachem definiujemy całe rodziny klas lub funkcji. Po podstawieniu za parametry konkretnych argumentów szablonu dostajemy już egzemplarz "zwykłego" typu (klasy) lub funkcji (nazywane również instancjami szablonu). Argumenty szablonu mogą reprezentować typy i w ten sposób dostajemy narzędzie umożliwiające pisanie ogólnego kodu parametryzowanego typem używanych w nim zmiennych, typem argumentów wywołania funkcji itp.

Szablony okazały się bardzo silnym narzędziem, których zastosowanie daleko przekracza implementację prostych kontenerów i można spokojnie stwierdzić, że ich prawdziwy potencjał jest ciągle odkrywany. Szablony idealnie wspierają styl programowania nazywany programowaniem uogólnionym. Polega on na generalizowaniu algorytmów i struktur danych tak, aby były w dużej mierze niezależne od typów danych, na których działają lub z których się składają. Mam nadzieję, że po lekturze poprzedniego wykładu Państwo już widzą, że to jest właśnie to, do czego szablony zostały wymyślone. Nie oznacza to, że automatycznie każdy program używajacy szablonów jest od razu programem uogólnionym. Tak jak i w przypadku tworzenia zwykłych (bez szablonów) programów, trzeba się sporo natrudzić, aby uzyskać uniwersalny, łatwy do ponownego wykorzystania kod. Ten wykład ma właśnie za zadanie przekazać Państwu podstawowe wiadomości na temat pisania dobrych programów uogólnionych.

W programowaniu uogólnionym ważną rolę gra pojęcie konceptu. Koncept to asbtrakcyjna definicja rodziny typów. To pojęcie pełni podobną rolę jak interfejs w programowaniu uogólnionym, ale przynależność do tej rodziny jest określona proceduralnie: do konceptu należą typy, które spełniają pewne wymagania. Czyli jeśli coś

kwacze jak kaczka to jest to kaczka, a nie: to jest kaczka jeśli należy do rodziny "kaczowatych". Koncepty omówię w dalszej części tego wykładu.

Co to jest programowanie uogólnione łatwiej jest pokazać na przykładach niż opisać. Niewątpliwie najważniejszą i najbardziej znaną aplikacją programowania ogólnego jest Standardowa Biblioteka Szablonów (STL - Standard Template Library), będąca oficjalną częścią standardu C++. W tych wykladach będę się bardzo często posługiwał przykładami z STL-a, ale szczegółowe nauczenie posługiwania się tą biblioteką **nie** jest celem tego wykładu. Powinni jednak Państwo zrobić to sami. Dlatego zachęcam do analizy przykładów zamieszczonych na wykładzie oraz wykonywanie podanych ćwiczeń.

Drugim znakomitycm źródłem przykladów uogólnionego kodu jest repozytorium bibliotek boost (http://www.boost.org/). Stamtąd też będę podawał przykłady i znów gorąco zachęcam Państwa do zaglądania tam samemu.

Programowanie uogólnione samo w sobie szczególnie obiektowe nie jest, choć oczywiście wymaga możliwości definiowania własnych typów. Oba style programowania: uogólniony i obiektowy można oczywiście stosować razem. Każdy ma swoje charakterystyczne cechy i aby je podkreślić jeszcze raz przypomnę podstawy programowania obiektowego rozumianego jako programowanie z użyciem interfejsów(klas abstrakcyjnych) i funkcji wirtulanych.

Polimorfizm dynamiczny

Sercem programowania obiektowego, oczywiście poza koncepcją klasy i obiektu, jest polimorfizm dynamiczny, czyli możliwość decydowania o tym jaka funkcja zostanie wywołana pod daną nazwą nie w momencie kompilacji (czyli pisania kodu), ale w samym momecie wywołania. Zilustrujemy to na przykładzie. W tym celu skorzystamy z "matki wszystkich przykładów programowania obiektowego", czyli klasy kształtów graficznych:).

Problem jest następujący: nasz program w pewnym momencie musi manipulować kształtami graficznym: rysować, przesuwać, obracać itp. Jest w miarę oczywiste, że każdy kształt będzie posiadał swoją klasę. Następnym krokiem jest ocena które operacje w naszym kodzie wymagają szczególowej znajomości kształtu, a które tylko ogólnych jego własności. Ewidentnie operacja rysowania obiektu należy do tych pierwszych i musi być zdefiniowana w klasie danego kształtu. Mówimy, że "obiekt wie jak się narysować". Często mówi się o tym również jako o ustaleniu odpowiedzialności, czy o podziale obowiązków. Tak więc ustaliliśmy, że do obowiązków obiektu należy umiejętność narysowania się. Jeśli tak, to właściwie cała część kodu manipulującego kształtami nie musi znać szczegółów ich implementacji. Weźmy na przykład fragment aplikacji odpowiedzialny za odświeżanie ekranu. Zakładamy, że wskaźniki do wyświetlanych kształtów są przechowywane w tablicy `shape_table`:

```
for(size_t i=0;i<n;++i)
   shape_table[i]->draw();
```

kod źródłowy

Programista piszący ten kod nie musi wiedziec jakiego typu kształt jest przechowywany w danym elemencie tablicy `shape_table` i jak jest zaimplementowana funkcja `draw`. Istotne jest by każdy obiekt, którego wkaźnik przechowywany jest w tej tablicy posiadał metodę `draw`. Innymi słowy programista korzysta tu tylko ze znajomości i dostępności interfejsu obiektów typu kształt, a resztę wykonuje kompilator, który generuje kod zapewniający wywołanie odpowiedniej funkcji. Aby taki interfejs zdefiniować tworzymy abstrakcyjną klasę obiektów typu kształt:

```
class Shape {
protected:
  long int _x;
  long int _y;
public:
  Shape(long x,long y):_x(x),_y(y){};
```

```
  long get_x() const {return _x;}
  long get_y() const {return _y;}
  virtual void draw() = 0;
  virtual ~Shape() {};
};
```

(Źródło shape.h)

Klasa ta stanowić będzie klasę bazową dla wszystkich klas opisujących kształty. Klasa `Shape` jest klasą abstrakcyjną, ponieważ zawiera niezaimplementowaną wirtualną czystą fukcję `void draw()`. Kod definiujący konkretne klasy kształtów może wyglądać następująco:

```
class Rectangle: public Shape {
protected:
  long _ur_x;
  long _ur_y;
public:
  Rectangle(long ll_x,long ll_y,long  ur_x,long ur_y):
    Shape(ll_x,ll_y),_ur_x(ur_x-ll_x),_ur_y(ur_y-ll_y) {};
  virtual void draw() {
    std::cerr<<"rectangle : "<<_x<<" "<<_y<<" : ";
    std::cerr<<_ur_x+_x<<" "<<_ur_y+_y<<std::endl;
  }
  long get_ur_x() const {return _ur_x;};
  long get_ur_y() const {return _ur_y;};
};
```

(Źródło rectangle.h)

i

```
class Circle: public Shape {
 protected:
  long _r;
 public:
  Circle(long x, long y,long r) :Shape(x,y), _r(r) {}

  virtual void draw() {
  std::cerr<<"Circle : "<<_x<<" "<<_y<<" : "<<_r<<std::endl;
  }
  long get_r() const {return _r;};
};
```

(Źródło circle.h)

Teraz możemy zdefiniować już funkcję odświeżającą ekran:

```
void draw_shapes(Shape *table[],size_t n) {
  for(size_t i=0;i<n;++i)
    table[i]->draw();
}
```

(Źródło draw.cpp)

Funkcja `draw_shapes` wykorzystuje zachowanie polimorficzne: to która funkcja `draw` zostanie wywołana zależy od tego jaki konkretny kształt jest wskazywany przez element tablicy. Łatwo się o tym przekonać wykonując np. następujący kod

```
int main() {
  Shape *list[4];
  list[0]=new Circle(0,0,100);
  list[1]=new Rectangle(20,20,80,80);
```

```
  list[2]=new Circle(10,10,100);
  list[3]=new Rectangle(20,0,80,10);
  draw_shapes(list,4);
}
```

kod źródłowy

W ten sposób zaimplementowaliśmy podstawowy paradygmat programowania obiektowego: rozdzielenie interfejsu od implementacji za pomocą abstrakcyjnej klasy bazowej i wykorzystanie funkcji wirtualnych. Ważną częścią tego procesu jest więc właśnie odpowiedni wybór interfejsów (klas bazowych).

Polimorfizm statyczny

Patrząc na kod funkcji `draw_shapes` możemy zauważyć, że korzysta on jedynie z własności posiadania przez wskazywane obiekty metody `draw()`. To sygnatura, czyli typ parametru wywołania tej funkcji określa, że musi to być wskaźnik na typ `Shape`. Z poprzedniego wykładu pamiętamy, że możemy zrezygnować z wymuszania typu argumentu wywołania funkcji poprzez użycie szablonu funcji:

```
template<typename T> void draw_template(T table[],size_t n) {
  for(size_t i=0;i<n;++i)
    table[i].draw();
}
```

(Źródło draw_template.h)

Taką funkcję możemy wywołać dla dowolnej tablicy, byle tylko przechowywany typ posiadał metodę `draw`. Mogą to być obiekty typów `Circle` i `Rectangle` (nie `Shape`, obiekty klasy `Shape` nie istnieją!), ale też inne zupełnie z nimi nie związane. Ilustruje to poniższy przykład:

```
class Drawable {
public:
  void draw() {cerr<<"hello world!"<<endl;}
};

int main() {
  Drawable table_d[1]={Drawable()};
  Circle   table_c[2]={Circle(10,10),Circle(0,50)};

  draw_template(table_d,1);
  draw_template(table_c,2);
}
```

kod źródłowy

Korzystając z szablonów uzyskaliśmy więc również pewien efekt zachowania polimorficznego. W przeciwieństwie do poprzedniego przykładu jest to polimorfizm statyczny: to kompilator zadecyduje na podstawie typu tablicy jaką funkcję `draw` wywołać. Oczywiście w rozważanym przypadku to podejście jest całkowicie nieadekwatne, mamy bowiem do czynienia z niejednorodną rodziną kształtów, a wybór konkretnych kształtów dokunuje się podczas wykonywania programu. Podając przykład z szablonami chciałem tylko podkreślić różnice pomiędzy tymi dwoma technikami. Przykłady kiedy to szablony okazują się lepszym rozwiązaniem zostały podane w poprzednim wykładzie.

Polimorfizm statyczny vs. dynamiczny

Jak już wspomniałem każdy styl posiada swoje cechy, które w zależności od okoliczności mogą być postrzegane jako wady lub zalety. Poniżej podaję zebrane głowne właściwości każdego podejścia.

1. Dziedzieczenie i funkcje wirtualne

    1. umożliwia pracę ze zbiorami niejednorodnych obiektów i korzysta z polimorfizmu dynamicznego
    2. wymaga wspólnej hierarchii dziedziczenia
    3. wymusza korzystanie ze wskaźników lub referencji i funkcji wirtualnych
    4. zazwyczaj generuje mniejszy kod.
2. Szablony
    1. implementuje polimorfizm statyczny
    2. bezpiecznie obsługuje jednorodne zbiory obiektów
    3. nie trzeba korzystać ze wskaźników i referencji ani funkcji wirtualnych
    4. nie musimy korzystać ze wspólnej hierarchii dziedziczenia.

Koncepty

Przyjrzyjmy się jeszcze raz deklaracji funkcji `draw_shapes` i `draw_template`. Kiedy programista widzi deklarację:

```
void draw_shapes(Shape *table[])
```

wie, że interfejs wykorzystywany przez funkcję `draw` jest zdefiniowany przez klasę `Shape`. Aby go poznać musi przeczytać kod i dokumentację tej klasy. Natomiast kiedy programista widzi deklarację:

```
template<typename T> void draw_template(T table[],size_t n);
```

to musi prześledzić kod funkcji `draw_templates` aby poznać ograniczenia nałożone na argument szablonu `T`. W tym przypadku nie jest to trudne, ale ogólnie może to być nietrywialne zadanie.

Zamiast jednak definiować ograniczenia i warunki dla każdego szablonu osobno, możemy szukać wspólnych, powtarzających się zestawów warunków. Taki zestaw nazwiemy konceptem i będziemy go traktować jako abstrakcyjną definicję całej rodziny typów, niezależną od konkretnego szablonu. Typ spełniający warunki konceptu nazywamy modelem konceptu lub mówimy, że modeluje ten koncept. Mając wybrany, dobrze przygotowany zestaw konceptów dla danej dziedziny, możemy się nimi posługiwać przy definiowaniu typów i algorytmów uogólnionych.

Koncepty mogą tworzyć hierachie analogiczne do hierarachii dziedziecznia. Mówimy, że koncept `A` jest bardziej wyspecjalizowany niż `B` (`A` is-refinement-of `B`), jeśli zestaw ograniczeń konceptu `B` zawiera się w zestwie ograniczeń konceptu `A`. Będę też używał określenia `A` jest "uszczegółowieniem" `B`.

Pojęcie konceptu pełni więc przy programowaniu za pomocą szablonów podobną rolę jak pojęcie interfejsu przy programowaniu za pomocą abstrakcyjnych klas bazowych i polimorfizmu dynamicznego. W przeciwieństwie do interfejsu jest to jednak pojęcie bardziej "ulotne", bo nie narzucamy go za pomocą formalnej definicji klasy abstrakcyjnej. Koncepty definiujemy poprzez mniej lub bardziej ścisłe wypisanie nakładanych przez nie ograniczeń. Ograniczenia te mogą zawierać między innymi:

1. Prawidłowe wyrażenia. Zestaw wyrażeń języka C++, które muszą się poprawnie kompilować.
2. Typy stowarzyszone. Ewentualne dodatkowe typy występujące w prawidłowych wyrażeniach.
3. Semantyka: zanczenie wyrażeń. Jednym ze sposobów określanie semantyki jest podawanie niezmienników, czyli wyrażeń, które dla danego konceptu są zawsze prawdziwe.
4. Złożoność algorytmów. Gwarancje co do czasu i innych zasobów potrzebnych do wykonania danego wyrażenia.

Programowanie uogólnione polega więc na wyszukiwaniu konceptów na tyle ogólnych, aby pasowały do dużej liczby typów i na tyle szczegółowych, aby zezwalały na wydajną implementację.

Definiowanie konceptów

Weźmy za przykład szablon funkcji `max` z poprzedniego wykładu

```
template<typename T> max(T a,T b) {return (a>b)?a:b;}
```

i zastanówmy się, jakie koncepty możemy odkryć w tak prostym kodzie.

Zacznijmy od gramatyki. Jakie warunki musi spełniać typ `T`, aby podstawienie go jako argument szablonu `max` dawało poprawne wyrażenie? Oczywistym warunkiem jest, że dla tego typu musi być zdefiniowany operator porównania `bool operator>(...)`. Specjalnie nie wyspecyfikowałem sygnatury tego operatora. Nie ma np. znaczenia jak parametry są przekazywane, co więcej `operator>(...)` może być zdefiniowany jako składowa klasy i posiadać tylko jeden jawny argument. Ważne jest to, że jeśli `x` i `y` są obiektami typu `T` to wyrażenie:

```
x>y
```

jest poprawne (skompiluje się).

Łatwiej jest przeoczyć fakt, że ponieważ argumenty wywołania są zwracane i przekazywane przez wartość, to typ `T` musi posiadać konstruktor kopiujący. Oznacza to, że jeśli `x` i `y` są obiektami typu `T` to wyrażenia:

```
T(x);
T x(y);
T x = y;
```

są poprawne.

PRZYKŁAD 2.1

Spełnienie obydwu tych warunków zapewni nam poprawność gramatyczną wywołania szablonu z danym typem, tzn. kod się skompiluje.

A co z poprawnością semantyczną? Mogłoby sie wydawać, że jest bez znaczenia jak zdefiniujemy `operator>(...)`. Koncept typu `T` jest jednak częścią kontraktu dla funkcji `max`. Kontraktu zawieranego pomiędzy twórcą tego wielce skomplikowanego kodu, a jego użytkownikiem. Kontrakt stanowi, że jeżeli użytkownik dostarczy do funkcji argumenty o typach zgodnych z konceptem i o wartościach spełniających być może inne warunki wstępne, to twórca funkcji gwarantuje, że zwróci ona poprawny wynik.

Zastanówny się więc jak zdefiniować poprawność dla funkcji maksimum. Z definicji maksimum żaden element argument funkcji `max` nie może być większy od wyniku, czyli wyrażenie

$$!(a > max(a,b)) \wedge !(b > max(a,b)) \quad (2.1)$$

musi być zawsze prawdziwe. Jasne jest, że jeśli dla jakiegoś typu `X` zdefiniujemy operator porównania tak, aby zwracał zawsze prawdę

```
bool operator>(const X &a,const X &b) {return 1;}
```

lub aby był równoważny operatorowi równości:

```
bool operator>(const X &a,const X &b) {return a==b;}
```

to wyrażenie 2.1 nie może być prawdziwe dla żadnej wartości `a` i `b`. Aby funkcja `max` mogła spełnić swój warunek końcowy musimy narzucić pewne ograniczenia semantyczne na `operator>()`. Te warunki to żądanie, aby relacja większości definiowana przez ten operator byłą relacją porządku częściowego, a więc aby spełnione było

$(x > x) = false$ i $(x > y) \wedge (y > z) => (x > z)$

To rozumowanie możnaby ciągnąć dalej i zauważyć, że nawet z tym ograniczeniem uzyskamy nieintuicyjne wyniki w przypadku, gdy obiekty `a` i `b` będą nieporównywalne, tzn. $!(a > b)$ i $!(b > a)$.

Poprawność semantyczną konstruktora kopiującego jest trudniej zdefiniować, ograniczymy się więc tylko do stwierdzenia, że wykonanie operacji 2.1 powoduje powstanie kopii obiektu `x` (cokolwiek by to nie znaczyło).

Comparable i Assignable

Reasumując, dostajemy zbiór warunków, które musi spełniać typ `T`, aby móc go podstawić do szablonu funkcji `max`. Czy to oznacza, że zdefiniowaliśmy już poprawny koncept? Żeby się o tym przekonać spróbujmy go nazwać. Narzuca się nazwa w stylu `Comparable`, ale wtedy łatwo zauważyć, że istnienie konstruktora kopiującego nie ma z tym nic wspólnego. Próbujemy upchnąc dwa niezależne pojęcia do jednego worka. Co więcej bardzo łatwo jest zrezygnować z konieczności posiadania konstruktora kopiujacego, zmieniając deklarację `max` na:

```
template<typename T> const T& max(const T&,const T&);
```

Teraz argumenty i wartość zwracana przekazywane są przez referencję i nie ma potrzeby kopiowania obiektów.

Logiczne jest więc wydzielenie dwu konceptów: jednego definiującego typy porównywalne, drugiego - typy "kopiowalne". Dalej możemy zauważyć, że istnienie operatora > automatycznie pozwala na zdefiniowanie operatora < poprzez:

```
bool operator<(const T& a,const T&b) {return b>a;};
```

Podobnie istnienie konstruktora kopiującego jest blisko związane z istnieniem operatora przypisania.

Tak więc dochodzimy do dwu konceptów: `Comparable` reprezentującego typy, których obiekty można porównywać za pomocą operatorów < i > oraz `Assignable` reprezentujacego typy, których obiekty możemy kopiować i przypisywać do siebie. Taką zabawę można kontynuować, pytając np. co z operatorem porównania `operator==()`?, co z konstruktorem defaultowym? itd. Widać więc, że koncepty to sprawa subietywna, ale to żadna nowość. Wybór używanych abstrakcji jest zawsze sprawą mniej lub bardziej subiektywną i silnie zależną od rozpatrywanego problemu. O tym czy dwa pojęcia włączymy do jednego konceptu czy nie decyduje np. odpowiedź na pytanie czy prawdopodobne jest użycie kiedykolwiek któregoś z tych pojęć osobno?

Tak więc zanim zaczniemy defniować koncepty musimy ustalić w jakim kontekście je rozpatrujemy. Na tym wykladzie kontekstem jest STL i oba wprowadzone koncepty są wzorowane na koncetach z STL-a. Należy jednak nadmienić, że pojęcie konceptu nie pojawia się wprost w definicji stadardu C++. Najlepiej koncepty STL przedstawione są na stronach firmy SGI (http://www.sgi.com/tech/stl/) dokąd Państwa odsyłam.

STL

Standardowa Biblioteka Szablonów (STL) to doskonałe narzędzie programistyczne zawarte w standardzie C++. Stanowi ona również znakomity, niejako sztandarowy, przykład programowania uogólnionego. Na tę bibliotekę można patrzeć więc dwojako: jako rozszerzenie języka C++ o dodatkowe funkcje lub jako na zbiór konceptów stanowiących podstawę do projetowania programów uogólnionych. Ja chciałbym podkreślić tutaj ten drugi aspekt, podkreślając jednak, że dobre poznanie możliwości STL-a może bardzo ułatwić Państwu prace programistyczne.

Biblioteka składa się zasadniczo z dwu części: uogólnionych kontenerów i uogólnionych algorytmów. Trzecią cześcią, niejako sklejającą te dwie, są iteratory.

Kontenery to obiekty służące do przechowywania innych obiektów. Kontenery w STL są jednorodne, tzn. mogą przechowywać tylko zbiory (kolekcje) obiektów tego samego typu. Kluczem do efektywnego programowania

uogólnionego jest jednak sprawa ujednolicenia dostępu do zawartości kontenera. Rozważmy dla przykładu dwa typowe kontenery `vector` i `list`, implementujące odpowiednio "inteligentną" tablicę oraz listę dwukierunkową. Naturalnym sposobem dostępu do tablicy jest indeksowanie:

```
std::vector<int> v(10);
v[8]=1;
```

a listy przeglądamy po kolei, przesuwając się o jeden element w przód czy w tył

```
Uwaga! To nie jest kod STL-owy !!!
lista<int> l;
l.reset(); ustawia element bieżacy na początek listy
for(int i=0;i<8;i++)
     l.next(); przesuwa element bieżący o jeden element do przodu

l.current()=1; zwraca referencję do elementu bieżącego
```

Widać, że w takim sformułowaniu praktycznie nie jest możliwe napisanie ogólnego kodu np. dodającego wszystkie elementy kontenera czy wyszukującego jakiś element w kontenerze. Ponadto opisany sposób dostępu do listy ogranicza nas do korzystania z jednego bieżącego elementu na raz.

Rozwiązaniem tego problemu zastosowanym w STL jest koncept iteratora, który definiuje abstrakcyjny interfejs dostępu do elementów kontenera. W STL iterator posiada semantykę wskaźnika, w szczególności może być zwykłym wskaźnikiem, choć normalnie jest to wskaźnik inteligentny. Każdy kontener posiada zestaw funkcji zwracających iteratory do swojego początku i na swój koniec. Korzystając z nich można listę przeglądać następująco

```
std::list<int> l;
tu jakoś inicjalizujemy liste
for(list<int>::iterator it=l.begin();it!=l.end();it++) {
     każdy kontener definiuje typ stowarzyszony nazwany iterator
  cout<<*it<<endl;
     korzystamy z iteratorów jak ze zwykłych wskaźników
  }
}
```

Przykładowy ogólny algorytm oparty o iteratory może wyglądać w ten sposób:

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init) {
T total=init;
for( ; first != last;++first)
   total+= *first;
return total;
}
```

(Źródło: accumulate.cpp)

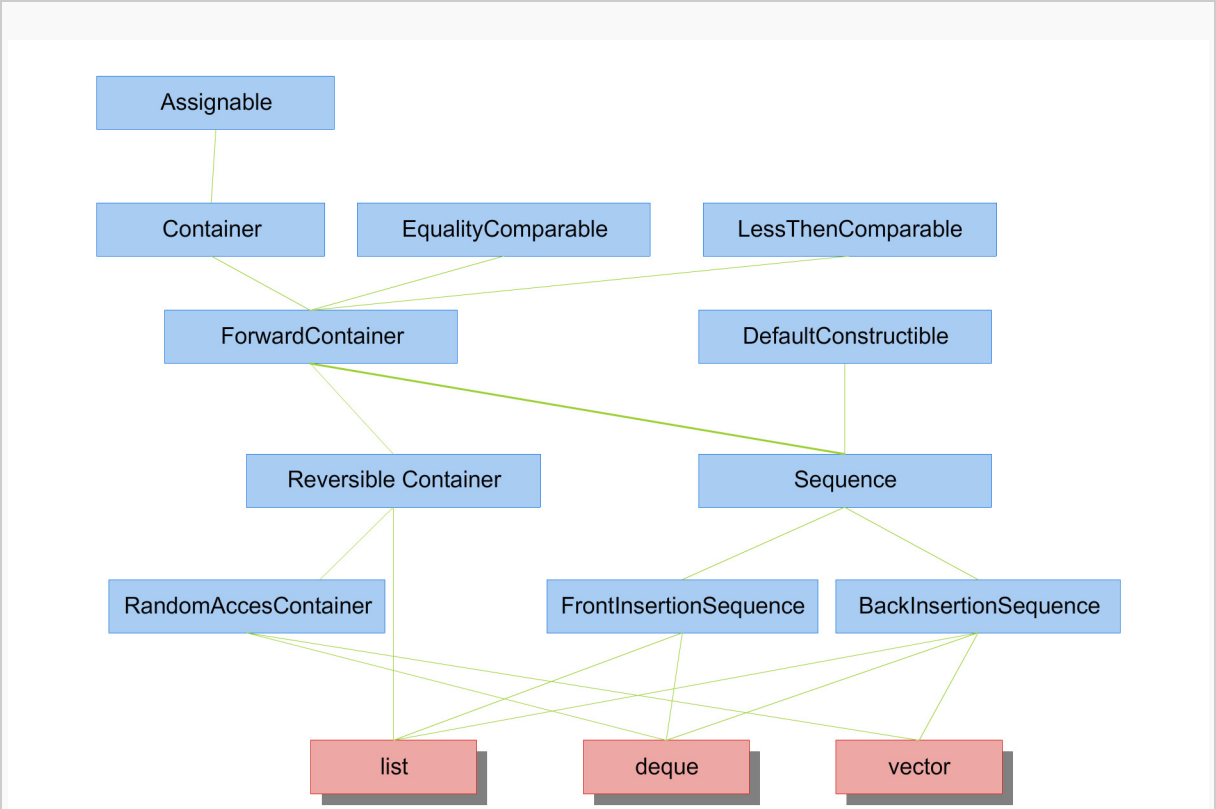
Oczywiście nie da się zignorować fundamentalnych różnic pomiędzy listą a wektorem. Dlatego np. iterator wektora zezwala na konstrukcje `it[i]`, a iterator listy już nie. Oznacza to, że algorytm, który działa dla iteratorów wektora (np. `sort`), nie musi działać dla iteratora listy. W języku konceptów oznacza to, że `std::vector<T>::iterator` jest modelem konceptu bardziej wyspecjalizowanego niż koncept, którego modelem jest `std::list<T>::iterator`. Zobaczymy to w następnej części tego wykładu.

Kontenery

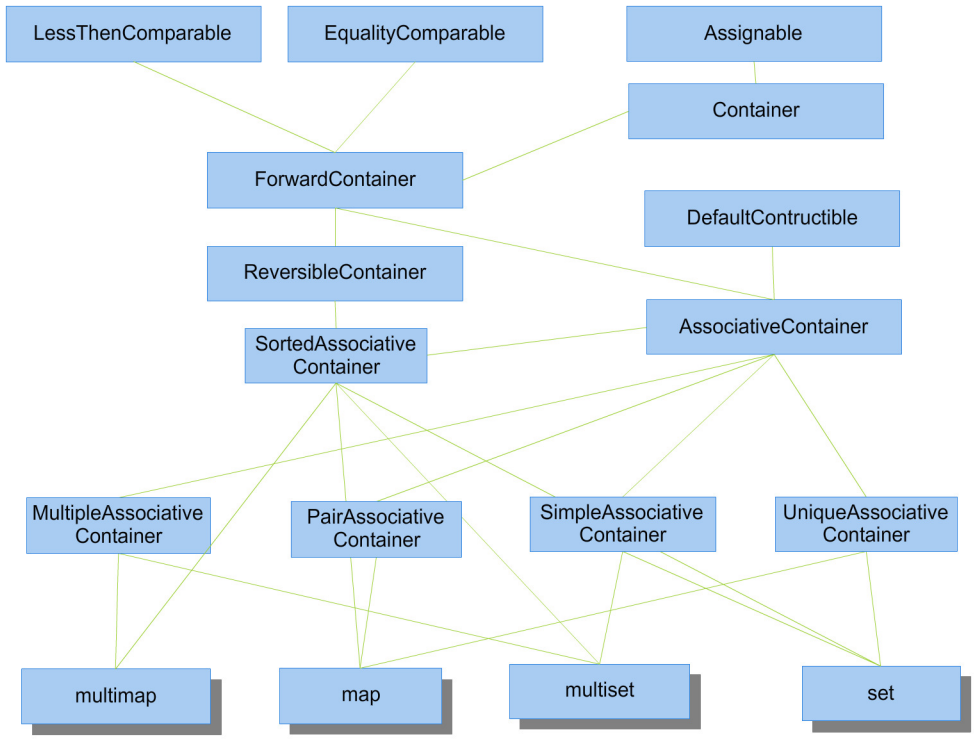
Standard C++ definiuje dwa zestawy kontenerów wchodzące w skład STL:

1. Sekwencje czyli pojemniki, w których kolejność elementów jest ustalana przez korzystającego z pojemnika (klienta) są to:
    1. `vector` (http://www.sgi.com/tech/stl/Vector.html),
    2. `deque` (http://www.sgi.com/tech/stl/Deque.html),
    3. `list` (http://www.sgi.com/tech/stl/List.html).
2. Kontenery asocjacyjne, czyli pojemniki, w których klient nie ma kontroli nad kolejnością elementów, są to:
    1. `set` (http://www.sgi.com/tech/stl/set.html),
    2. `map` (http://www.sgi.com/tech/stl/Map.html),
    3. `multiset` (http://www.sgi.com/tech/stl/multiset.html),
    4. `multimap` (http://www.sgi.com/tech/stl/Multimap.html).

Ponadto różni dostawcy oferują dodatkowe pojemniki. Na uwagę zasługuje znakomita darmowa implementacja STL firmy Silicon Graphics, która miedy innymi wchodzi w skład pakietu g++ i dostarcza dodatkowo takich kontenerów jak: lista jednokierunkowa `slist` oraz tablice haszujące `hash_set` czy `hash_map` (zob. STL (http://www.sgi.com/tech/stl/)). Hierachię konceptów kontenerów typu sekwencji przedstawia rysunek 2.1, a kontenerów asocjacyjnych rysunek 2.2.



Rysunek 2.1. Hierarchia konceptów dla pojemników typu sekwencyjnego.



Rysunek 2.2. Hierarchia konceptów dla pojemników typu asocjacyjnego.

Nie będę tu omawiał tych wszystkich konceptów. Ich szczegółowe opisy znajdują się na stronie http://www.sgi.com/tech/stl. Tutaj chciałbym tylko dodać parę luźnych komentarzy.

Po pierwsze, rodzi się pytanie czy taka skomplikowana taksonomia jest potrzebna? W końcu patrząc na rysunki widać, że konceptów jest dużo więcej niż typów kontenerów. Rzeczywiście, do posługiwania się biblioteką w zasadzie wystarczy zaznajomić się z opisami kontenerów i hierarchią iteratorów (zob. rysunek 2.3). Podane klasyfikacje przydają się dopiero kiedy dodajemy własne elementy do biblioteki. Dobierając do implemetacji najbardziej ogólny koncept spełniający nasze wymagania zwiększamy potencjał ponownego użycia naszego kodu z innymi komponentami biblioteki, czy kodem innych developerów.

Kontenery z STL są właścicielami swoich elementów, zniszczenie kontenera powoduje zniszczenie jego elementów. Wszytkie operacje wkładania elementów do kontenera używają przekazywania przez wartość, czyli kopiują wkładany obiekt. Jeżeli chcemy, aby czas życia elementów kontenera był dłuższy od czasu życia kontenera, należy użyć wskaźników.

Kontenery różnią się nie tylko rodzajem iteratorów, jaki implementują, ale również rodzajem operacji, które można wykonać bez unieważnienia istniejących iteratorów. Pokażę to na przykładzie:

```
std::vector<int>::iterator it;
int i;

std::vector<int> v(1);
std::vector<int> buff(100); staramy się zająć pamięć za v

v[0]=0;
it=v.begin();
i=(*it); OK, przypisuje i=0
for(int i=0;i<10;++i)
  v.push_back(i);
    ponieważ przekraczamy koniec wektora, kontener zaalokuje dodatkową pamięć. Może
    się to wiązać z koniecznością przeniesienia zawartości wektora v w inne miejsce
    pamięci. To spowoduje, że wskaźnik it przestanie pokazywać na początek wektora v

std::cerr<<(*it)<<std::endl ;                  niezdefiniowane
```

```
std::cerr<<"iterator nieprawidlowy"<<std::endl;
for(;it != v.end(); ++it)  potencjalnie nieskończona pętla
  std::cerr<<*it<<std::endl;
;

std::cerr<<"iterator prawidlowy"<<std::endl;
for(it=v.begin();it != v.end(); ++it)
std::cerr<<*it<<std::endl;
;
```

(Źródło: invalid.cpp)

Bardzo Państwa na ten problem uczulam. Efekt działania powyższego kodu jest gorzej niż zły: jest niezdefiniowany!, tzn. będzie zależał od implementacji kompilatora, od zadeklarownych wcześniej zmiennych itp. Proszę np. spróbować wykomentować linijkę

```
std::vector<int> buff(100); staramy się zająć pamięć za v
```

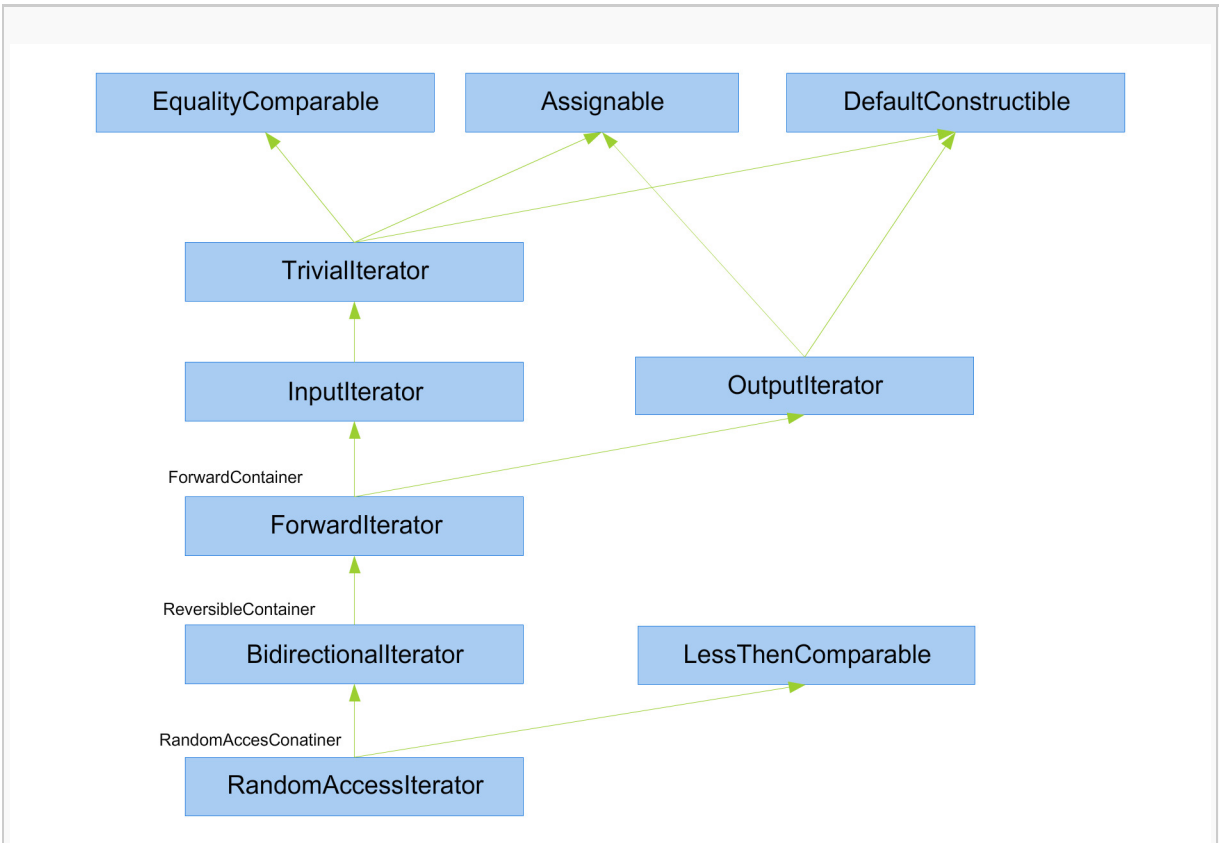
i porównać wynik działania programu. Może się również zdarzyć, że program zadziała poprawnie (wbrew pozorom jest to najgorsza możliwa sytuacja!).

Ważne są gwarancje złożoności metod kontenera. Ewidentnie każdy rodzaj kontenera może dostarczyć każdego rodzaju operacji, różny będzie jednak czas ich wykonywania. I tak rząd O(1) jest gwarantowany w operacji indeksowania wektora. Natomiast operacja dodania elementu w środku wektora jest rzędu O(N). Z listą jest odwrotnie i dlatego listy w STL nie posiadają operacji indeksowania.

Nie wszystkie własności kontenerów są zdefiniowane w konceptach. Każdy kontener może definiować dodatkowe metody właściwe tylko dla niego.

Iteratory

Iteratory to koncept, który uogólnia pojęcie wskaźnika. Hierarchię konceptów iteratorów przedstawia rysunek 2.3. Zaznaczono na nim również które koncepty kontenerów wymagają danego modelu iteratora.

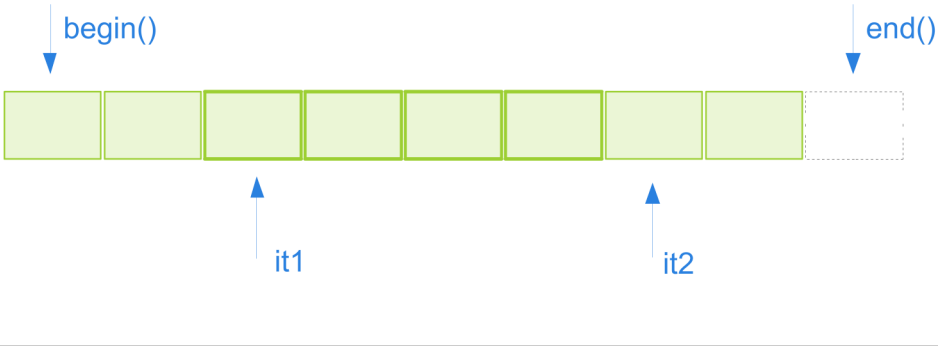


Rysunek 2.3. Hierarchia konceptów dla iteratorów.

Najprostsze iteratory pojawiające sie w STL-u to iteratory wejściowe i wyjściowe. Wprawdzie żaden kontener nie posiada iteratorów tego typu, ale iteratory wejściowe, umożliwiające tylko jednoprzebiegowe odczytanie wartości kontenera, są częstym wymaganiem dla argumentów algorytmów nie zmieniających elementów kontenera (non mutable algorithms).

Należy pamiętać, że iterator nie wie na jaki kontener wskazuje, czyli poprzez iterator nie ma dostępu do interfejsu kontenera.

Iteratory pozwalają na określanie zakresu elementów w kontenerze poprzez podanie iteratora wskazującego na początek i na pierwszy element poza końcem zakresu. Zakres oznaczamy poprzez [`it1`,`it2`) (zob. rysunek 2.4).



Rysunek 2.4. Zakres.

Z tego powodu dozwolona jest instrukcja pobrania adresu pierwszego elementu poza końcem tablicy.

```
double x[10];
double *end=&x[10];
//zwykłe wskaźniki mogą być użyte jako iteratory
std::cout<<accumulate(x,end,0)<<endl; <i>suma elementów tablicy</i>
```

Każdy kontener posiada motody `begin()` i `end()`, zwracające iterator na początek i "poza koniec". Typowa pętla obsługi kontenera wygląda więc następująco:

```
typedef vector<int>::iterator iterator;
vector<it> v(100);
for(iterator it=v.begin();it!=v.end();++it) {
  ...
}
```

(Źródło: accumulate.cpp)

Proszę zwrócić uwagę na wykorzystanie operatora `!=` do sprawdzenia końca zakresu. Tylko iteratory o dostępie swobodnym mogą być porównywane za pomocą operatora `operator<()`. Reszta jest tylko `EqualityComparable`.

Algorytmy

Algorytmy działają na zakresach elementów kontenera definiowanych przez dwa iteratory, a nie na kontenerach. Umożliwia to jednolity dostęp do różnych kontenerów. Takie podejście ma też inne konsekwencje, jak już

pisałem iterator nie wie z jakiego kontenera pochodzi, w szczególności oznacza to, że algorytmy ogólne nie mogą usuwać elementów z kontenera.

Oczywiście część algorytmów, np. `sort`, wymaga bardziej wyrafinowanych iteratorów, nie dostarczanych przez każdy kontener. Wiele jednak jednoprzebiegowych algorytmów zadawala się iteratorami wejściowymi.

Poza iteratorami uogólnione algorytmy wykorzystują obiekty funkcyjne czyli funktory. Obiekt funkcyjny to koncept będący uogólnieniem pojęcia fukcji, czyli coś do czego można zastosować składnię wywołania funkcji. W C++ mogą to być funkcje, wskaźniki do funkcji oraz obiekty klas, w których zdefiniowano `operator()(...)`.

Funktory w STL są podzielone ze względu na liczbę argumentów wywołania. `Generator` nie przyjmuje żadnego argumentu, `UnaryFunction` posiada jeden argument, a `BinaryFunction` - dwa argumenty wywołania. Ważną podklasą są funkcje zwracające wartość typu `bool`, nazywane predykatami. Rozróżniamy więc `UnaryPredicate` i `BinaryPredicate`.

Żeby zilustrować użycie algorytmów i funktorów rozważmy następujący przykład. Najpierw definiujemy funktor, który posłuży nam do generowania sekwencji obiektów:

```
template<typename T> class SequenceGen {
private:
  T _start;
  T _step;
public:
  SequenceGen(T start = T(),T step = 1 ):
  _start(start),_step(step){};

  T operator()() {T tmp=_start; _start+=_step; return tmp;}
};
```

(Źródło: bind.cpp)

Za pomocą obiektu klasy `SequenceGen` możemy wypełnić wektor sekwencją 20 pierwszych nieparzystych liczb całkowitych:

```
const size_t n = 20 ;
vector<int> v(n);
generate_n(v.begin(),n,SequenceGen<int>(1,2));
```

(Źródło: bind.cpp)

Standardowy algorytm

```
template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first,
                          Size n, Generator gen);
```

służy właśnie do wypełniania kontenerów za pomocą `n` kolejnych wyników wywołania funktora `gen`. Powyższy kod ilustruje typowy sposób opisu algorytmów w STL. Nazwy parametrów szablonu odpowiadają nazwom konceptów, które muszą modelować.

W tak wypełnionym kontenerze poszukamy pierwszego elementu większego od czterech (powinno to być pięć). Służy do tego algorytm

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first,
```

```
                    InputIterator last,
                    Predicate pred);
```

Który przeszukuje zakres `[first,last)` do napotkania pierwszego elementu, dla którego predykat `pred` jest prawdziwy i zwraca iterator do tego elementu. Jeśli takiego elementu nie ma, to `find_if` zwraca `last`. Do zakończenia programu potrzebujemy jeszcze predykatu, który testuje czy dana wartość jest większa od czterech. Zamiast go implementować skorzystamy z adaptera funkcji `bind2nd`. Ta funkcja przyjmuje funktor dwuargumentowy (`AdaptableBinaryFunction`) `F(T,U)` i jakąś wartość `x` typu `U` i zwraca funktor jednoparametrowy `F(T,x)`. Korzystając z predefiniowanego predykatu `greater` możemy napisać:

```
vector<int>::iterator it= find_if(v.begin(),v.end(),
                                  bind2nd(greater<int>(),4));
if(it!=v.end())
  cout<<*it<<endl;
else
  cout<<"nie znaleziono zadanego elementu";
}
```

(Źródło: bind.cpp)

STL wprowadza więc do C++ elementy programowania funkcyjnego.

Debugowanie

Sprawdzanie konceptów

Programowanie uogólnione korzysta istotnie z pojęcia konceptu. Koncept opisuje abstrakcyjne typy danych (czy funkcji), które mogą być użyte jako argumenty danego szablonu. Definiowanie konceptu polega tylko na jego opisie. C++ nie posiada żadnego mechanizmu pozwalającego na bardziej formalną definicję. Co za tym idzie, nie można też automatycznie sprawdzać czy nasz typ modeluje żądany koncept.

Oczywiście kompilator podczas konkretyzacji szablonu sprawdza syntaktyczną zgodność przekazanego typu z wymaganiami szablonu. Nie jest to jednak idealne narzędzie diagnostyczne. Po pierwsze, komunikat o błędzie może być bardzo zawiły i na pewno nie będzie się odnosił do nazwy konceptu. Po drugie, może się okazać, że szablon, który konkretyzujemy nie wykorzystuje wszystkich możliwych wyrażeń konceptu. Zresztą idea konceptu polega na rozdzieleniu definicji abstrakcyjnego typu od definicji szablonu, którego ten typ może być argumentem. Rozwiazaniem jest napisanie własnego zestawu szablonów, których jedynem zadaniem jest sprawdzanie zgodności przekazanych argumentów szablonu z definiowanym przez ten szablon konceptem. Niestety, można w ten sposób sprawdzać tylko zgodność syntaktyczną.

Idea tworzenia takich szablonów jest prosta (zob. http://www.boost.org/libs/concept_check/concept_check.htm) : dla każdego konceptu tworzymy szablon zawierający funkcję `constraints()`, która zawiera wszystkie możliwe poprawne wyrażenia dla danego konceptu. Np. dla konceptu `Comparable` możemy zdefiniować:

```
template<typename T> struct ComparableConcept {
void constraints() {
  require_boolean_expr( a > b);
  require_boolean_expr( a < b);
};
T a,b;
};
```

(Źródło: concept_check.cpp)

Szablon `require_boolean_expr`

```
template <class TT>
 void require_boolean_expr(const TT& t) {
```

```
    bool x = t;
    ignore_unused_variable_warning(x);
    używa zmiennej x aby kompilator nie generował ostrzeżenia
}
```

(Źródło: concept_check.cpp)

sprawdza czy jego argument, a więc wartość zwracana przez operatory, może być konwertowana na `bool`.

Zwracam uwagę, że nie możemy w kodzie szablonu `Comparable` użyć defaultowego konstruktora, bo nie jest on wymagany. Dlatego zmienne `a` i `b` nie były zdefiniowane wewnątrz funkcji `constraints()`, tylko jako pola składowe klasy. Ponieważ nie tworzymy żadnej instancji tej klasy, to nie będą wywoływane konstruktory, a więc kompilator nie będzie generował ich kodu.

Teraz potrzebujemy jeszcze sposobu, aby skompilować, ale nie wywołać, funkcję `ComparableConcept<T>::constraints()`. Możemy tego dokonać pobierając adres funkcji i przypisując go do wskaźnika. Kompilator skompiluje kod funkcji, ale jej nie wykona. Dodatkowo najprawdopodobniej kompilator optymalizujący usunie to przypisanie jako nieużywany kod, ale dopiero po kompilacji (no chyba, że jest bardzo, ale to bardzo inteligentny). Dla wygody opakujemy tę konstrukcję w szablon funkcji:

```
template <class Concept>
inline void function_requires() {
  void (Concept::*x)() = &Concept::constraints;
  ignore_unused_variable_warning(x);
}
```

(Źródło: concept_check.cpp)

Możemy teraz używać szablonu `Comparable` w następujący sposób:

```
main() {
 function_requires<ComparableConcept<int> >();
 function_requires<ComparableConcept<std::complex> >(); błąd
}
```

(Źródło: concept_check.cpp)

Bardziej skomplikowane koncepty możemy sprawdzać korzystając z klas sprawdzających dla innych konceptów, np:

```
template <class Container>
struct Mutable_ContainerConcept
{
  typedef typename Container::value_type value_type;
  typedef typename Container::reference reference;
  typedef typename Container::iterator iterator;
  typedef typename Container::pointer pointer;

  void constraints() {
    sprawdzamy czy spełnia wymagania konceptu Container
    function_requires< ContainerConcept<Container> >();
    function_requires< AssignableConcept<value_type> >();
    function_requires< InputIteratorConcept<iterator> >();

    i = c.begin();
    i = c.end();
    c.swap(c2);
  }
  iterator i;
```

```
  Container c, c2;
};
```

Biblioteka `boost`, skąd wzięty został ten przykład, posiada implementację szablonów dla każdego konceptu z STL (http://www.boost.org/libs/concept_check/concept_check.htm). Hierachia, którą można tam odczytać, różni się trochę od tej, którą wcześniej zaprezentowałem i która jest opisana w http://www.sgi.com/tech/stl/. Główna różnica to wprowadzenie rozróżnienia pomiędzy kontenerami, które umożliwiaja modyfikację swoich elementów (`MutableContainer`) i tych, które na to nie pozwalają (`Container`).

Archeotypy

Klasy sprawdzające koncepty służą do pomocy w implementacji typów będących modelami danego konceptu. Możemy jednak mieć sytuację odwrotną: implementujemy jakiś algorytm ogólny i chcemy się dowiedzieć jaki koncept jest wymagany dla parametrów szablonu? Chcemy wybrać jak najogólniejszy koncept, który jeszcze pozwala na poprawne działanie algorytmu. Pomóc mogą nam w tym archeotypy. Są to klasy, które dokładnie implementują interfejs danego konceptu. Opierając się na http://www.boost.org/libs/concept_check/concept_check.htm, przedstawię teraz implementację archeotypu dla konceptu `Comparable`.

Koncept `Comparable` nie wymaga posiadania konstruktora defaultowego, konstruktora kopiujacego oraz operatora przypisania, dlatego w naszym archeotypie zdefiniujemy je jako prywatne:

```
class comparable_archetype {
private:
  comparable_archetype() {};
  comparable_archetype(const comparable_archetype &) {};
  comparable_archetype &operator=(const comparable_archetype &) {
    return *this;};
public:
  comparable_archetype(dummy_argument) {};
};
```

(Źródło: archeotype.cpp)

Aby móc tworzyć obiekty typu `comparable_archetype` dodaliśmy niestandardowy konstruktor z argumentem sztucznego typu:

```
class dummy_argument {};
```

używanego tylko na tę okazję (jego nazwa powinna być unikatowa).

Operator `operator<()` nie musi zwracać wartości typu `bool`, a jedynie wartość typu konwertowalnego na `bool`, dlatego tworzymy taki typ:

```
struct boolean_archetype  {
  operator const bool() const {return true;}
};
```

i podajemy go jako typ zwracany przez operatory porównania

```
boolean_archetype operator<(const comparable_archetype &,
                            const comparable_archetype &){
    return boolean_archetype();
};
boolean_archetype  operator>(const comparable_archetype &,
                             const comparable_archetype &){
    return boolean_archetype();
};
```

(Źródło: archeotype.cpp)

Teraz możemy już przetestować nasz szablon `max`.

```
template<typename T>
const T &max(const T &a,const T &b) {return (a>b)?a:b;}

main() {
comparable_archetype ca(dummy_argument());
max(ca,ca);
}
```

(Źródło: archeotype.cpp)

Poprawna kompilacja tego kodu przekonuje nas, że koncept `Comparable` jest wystarczajacy, przynajmniej syntaktycznie. Proszę zwrócić uwagę, że jeśli użyjemy orginalnego szablonu

```
template<typename T> T max(T a,T b) {return (a>b)?a:b;}
```

(Źródło: archeotype.cpp)

to kod się nie skompiluje, bo zabraknie konstruktora kopiujacego.

Większość konceptów jest uszczegółowieniem innych konceptów. Implementacja archeotypów w biblitece boost (http://www.boost.org/libs/concept_check/concept_check.htm) zezwala na takie konstrukcje i gorąco zachęcam do zapoznania się z nią.

Źródło: "http://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_2:_Programowanie_uog%C3%B3lnione"