Лекція 12. Характеристики, політики

1. *Шаблон класу «Закусочна». Характеристики класів гостей, політики поведінки.*
2. *Характеристики послідовних контейнерів для шаблону виведення в потік.*
3. *Характеристики символів рядка. Рядок, нечутливий до регістра.*
4. *Обмеження конкретизацій класу раціональних чисел за допомогою характеристик.*
5. *Стандартні характеристики ітераторів та інші приклади.*
6. *Метапрограмування. Алгоритмічна повнота шаблонів.*

У цій лекції ми познайомимося з новим (для нас) засобом узагальненого програмування – характеристиками класів. Шаблони характеристик призначено для групування оголошень, які залежать від типу. Такі характеристики допомагають налаштовувати інші шаблони на правильну взаємодію з цим типом. Сказане легше пояснити на прикладі, то ж ним і займемося.

Шаблон класу «Закусочна»

Закусочна, буфет – вид їдальні, в якій продають страви, що готуються швидко і просто. [Академічний тлумачний словник (1970—1980)]. Ми моделюватимемо дуже простий заклад: у ньому подають один напій і одну закуску. При такому скромному асортименті потрібно враховувати вподобання гостя і пропонувати йому улюблені пригощення. Наприклад, дитячий буфет пригощає відвідувачів молоком і печивом, а салун на дикому заході – віскі й беконом.

Алгоритми функціонування обох закладів мають очевидну схожість, яку можна узагальнити шаблоном класу: гостя треба розмістити, подати йому напій, подати закуску. Але тип напою і тип закуски залежать від типу гостя. Можна сказати, що улюблені страви *характеризують* гостя. Як пов'язати тип гостя і тип закуски в програмі? Перше, що спадає на гадку – питати самого гостя. Тоді Клас гостя мав би мати відповідні методи. Таке очевидне рішення має очевидні недоліки. Код класу гостя може бути недоступним для модифікації. Клас закусочної може не мати тієї страви, яку хотів би «чужий» гість (ковбой в дитячому буфеті). І з практичної точки зору це зле: кому сподобається без кінця відповідати на запитання «що Вам запропонувати»?

Було б добре зібрати опис уподобань гостя в окремому класі і пов'язати його з типом гостя. Такий клас (чи шаблон) і називають характеристикою. Механізм оголошення шаблонів і спеціалізацій шаблонів допомагає це зробити без зміни класу гостя чи класу закусочної. Більше того, вибір відповідної характеристики (відповідної спеціалізації) може виконати компілятор ще на етапі трансляції програми.

Починається все з домовленостей, що саме буде оголошено в класі характеристик. У нашому прикладі домовимося, що характеристика міститиме оголошення трьох типів: *place_type* задає улюблене місце гостя, *beverage_type* – тип напою, *snack_type* – закуски. Тоді шаблон закусочної можна оголосити так:

```
template <class Guest, class Traits = GuestTraits<Guest>>
class BearCorner
{
      // характеристики гостя можна довідатися з класу Traits
      typedef typename Traits::place_type     place_type;
      typedef typename Traits::beverage_type  beverage_type;
      typedef typename Traits::snack_type     snack_type;
      Guest    theGuest;
      place_type  place;
      beverage_type bev;
      snack_type  snack;
public:
```

```
	BearCorner(const Guest& g): theGuest(g) {}
	void entertain()
	{
		std::cout<<"Entertaining " << theGuest << " locating " << place
			<< " serving " << bev << " and " << snack << '\n';
	}
};
```

Посилання на гостя екземпляр *BearCorner* отримує в конструкторі (можна було б обслуговувати і колекцію гостей, але це невиправдано ускладнило б оголошення класу), а екземпляри місця, напою і закуски, оголошені як приватні поля закусочної, мають створити конструктори за замовчуванням відповідних класів. Шаблон *BearCorner* «не переймається», що це за класи. Тут він покладається на оголошення в класі *Traits*.

Алгоритм обслуговування гостя описує метод *entertain*. Його зумисно максимально спрощено. Він виводить на консоль інформацію про гостя та обрані місце і страви. Очевидно, що класи гостей, напоїв і закусок повинні реалізувати оператор виведення в потік. Ось приклади таких класів:

```
// напої та закуски для гостей
class Milk
{ public:
    friend ostream& operator<<(ostream& os, const Milk&)
    {    return os << "Milk";    }
};
class CondensedMilk
{ public:
    friend ostream& operator<<(ostream& os, const CondensedMilk&)
    {    return os << "CondensedMilk";    }
};
class Water
{ public:
    friend ostream& operator<<(ostream& os, const Water&)
    {    return os << "Water";    }
};

class Honey
{ public:
    friend ostream& operator<<(ostream& os, const Honey&)
    {    return os << "Honey";    }
};
class Cookies
{ public:
    friend ostream& operator<<(ostream& os, const Cookies&)
    {    return os << "Cookies";    }
};
class Bread
{ public:
    friend ostream& operator<<(ostream& os, const Bread&)
    {    return os << "Bread";    }
};
class Meat
{ public:
    friend ostream& operator<<(ostream& os, const Meat&)
    {    return os << "Meat";    }
};

// Місця, де можна розташувати гостей
class Table
{ public:
    friend ostream& operator<<(ostream& os, const Table&)
    {    return os << "at the table";    }
};
```

```
class Carpet // килим, диван
{ public:
    friend ostream& operator<<(ostream& os, const Carpet&)
    {    return os << "on the carpet";    }
};
class Cage // клітка
{ public:
    friend ostream& operator<<(ostream& os, const Cage&)
    {    return os << "in the cage";    }
};
```

Усі вони дуже прості і дуже схожі, відрізняються хіба що рядком літер в операторі виведення в потік. Ми не будували тут ніяких ієрархій, хоча наслідування могло б бути. Можна було додати унікальної поведінки кожному з класів, але для демонстрації задуманого вистачить і того, що є. Класи гостей будуть дещо цікавішими (хоча б для того, щоб зовні відрізнятися від напоїв і закусок).

```
// Гостями закладів харчування бувають Медведі, хлопці, хижаки :)
// Їх описують відповідні класи, ніяк не пов'язані наслідуванням
class Bear  // Медвідь має якийсь вік
{     int age;
public:
      Bear(int y=5):age(y) {}
      friend ostream& operator<<(ostream& os, const Bear& B)
      {     return os << "Bear " << B.age << " years old";       }
};
class Boy  // Хлопець має ім'я
{     string name;
public:
      Boy(string n="Alex"):name(n) {}
      friend ostream& operator<<(ostream& os, const Boy& B)
      {     return os << "Boy " << B.name;      }
};
class Predator  // хижаків відрізняють за біологічним видом
{     string species;
public:
      Predator(string s="Volf"):species(s) {}
      friend ostream& operator<<(ostream& os, const Predator& P)
      {     return os << P.species;       }
};
```

А тепер – найцікавіше: класи, що пов’язують одних з другими. Основою для оголошення шаблонів характеристик стане порожній шаблон

```
template <class Guest> class GuestTraits;
/* place_type, beverage_type, snack_type */
```

Він задає ім’я *GuestTraits* і вказує, що шаблон залежить від одного параметра-типу. Використати такий шаблон для оголошення *BearCorner* не вдасться, бо він нічого не описує. Порожній *GuestTraits* потрібен для того, щоб для кожного типу гостя визначити спеціалізацію:

```
template<> class GuestTraits<Bear>
{ // характеристики медведя, як гостя:
  // місце - килим, напій - згущене молоко, закуска - мед
public:
      typedef Carpet place_type;
      typedef CondensedMilk beverage_type;
      typedef Honey snack_type;
};
```

```
template<> class GuestTraits<Boy>
{ // характеристики хлопчика, як гостя:
  // місце - за столом, напій - молоко, закуска - печиво
public:
      typedef Table place_type;
      typedef Milk beverage_type;
      typedef Cookies snack_type;
};
template<> class GuestTraits<Predator>
{ // характеристики хижака, як гостя:
  // місце - клітка, напій - вода, закуска - м'ясо
public:
      typedef Cage place_type;
      typedef Water beverage_type;
      typedef Meat snack_type;
};
```

Спеціалізації дотримуються домовленості і визначають типи *place_type*, *beverage_type*, *snack_type*. Тепер створення екземпляра закусочної виглядає дуже просто:

```
      // різноманітні гості закусочних
      Boy boy("Mathew");
      Bear A(3);
      // швидке створення закусочних відповідно до типу гостя
      BearCorner<Boy>  C1(boy);
      BearCorner<Bear> C2(A);
```

Розглянемо докладніше процес створення *C1*.

1. Компілятор знаходить оголошення шаблона *BearCorner* і підставляє тип *Boy* замість параметра *Guest*.
2. Другий параметр шаблона *BearCorner* не задано, тому компілятор використовує його значення за замовчуванням *Traits = GuestTraits<Boy>*. Тепер треба знайти відповідне оголошення.
3. Компілятор знаходить загальний шаблон *GuestTraits* і його спеціалізацію *GuestTraits<Boy>*. Відомо, що за таких умов компілятор віддає перевагу спеціалізації. Отож, ім'я *place_type* стає синонімом типу *Table*, ім'я *beverage_type* – типу *Milk*, а *snack_type* – типу *Coockies*.
4. Сконструйовано оголошення класу *BearCorner<Boy>*. Умовно його можна зобразити так

```
class BearCorner<Boy>
{     Boy theGuest;     Table place;      Milk bev;   Cookies snack;
public:
      BearCorner(const Boy& g): theGuest(g) {}
      void entertain()
      {     std::cout<<"Entertaining " << theGuest << " locating " << place
                  << " serving " << bev << " and " << snack << '\n';   }
};
```

5. Створено екземпляр *C1* класу *BearCorner<Boy>*, що містить посилання на *boy* і власні екземпляри класів *Table*, *Milk*, *Cookies*.
6. Тепер можна використовувати *C1*, наприклад: *C1.entertain();* – отримаємо:

```
Entertaining Boy Mathew locating at the table serving Milk and Cookies
```

Процес створення *C2* відбувається за схожим сценарієм, тільки компілятор вибирає іншу спеціалізацію шаблона характеристик – *GuestTraits<Bear>* – і використовує оголошені там типи для побудови класу *BearCorner<Bear>*.

Треба відзначити дві важливі риси оголошеної сім'ї шаблонів *GuestTraits*. По-перше, вона нагадує ієрархію класів. Тільки наслідування класів тут нема. Є уточнення концепцій:

кожна спеціалізація уточнює концепцію, задекларовану загальним шаблоном. Говорять про *статичний поліморфізм* оголошень. Поліморфізм, бо маємо різні оголошення під одним іменем. Статичний, бо розв'язується він на етапі трансляції (нагадаємо, що поліморфізм повідомлень – динамічний, вирішується на етапі виконання завдяки пізньому зв'язуванню методів). По-друге, вибір спеціалізації компілятором нагадує роботу інструкції *switch*. Це не тільки зовнішня схожість: спеціалізації шаблонів можна використовувати для організації галужень.

Використання нестандартних характеристик

Цікаво, що характеристики класу гостя можна описувати не тільки шаблоном, а й звичайним класом, створеним під нестандартні умови використання. Справді, домовленість передбачала оголошення трьох типів: місця, напою, закуски. Ніде, ні в домовленості, ні в синтаксисі, не сказано, що *GuestTraits* – шаблон. Ми використовували спеціалізації шаблона тільки для автоматизації вибору класу характеристик. Якщо ми згідні задавати характеристики явно, то можна обійтися звичайним класом.

```
class HelthTraits
{ public:
      typedef Carpet place_type;
      typedef Water beverage_type;
      typedef Bread snack_type;
};

      Boy yogi;
      BearCorner<Boy, HelthTraits> C3(yogi);
      C3.entertain();
```

Додаткове налаштування поведінки

У деяких випадках буває зручно пов'язати певну *функціональність* з аргументами шаблонів, щоб прикладний програміст міг легко змінити поведінку шаблона. Клас, що описує способи поведінки шаблона, називають *політикою*.

Нова версія класу «Закусочна» має різні способи обслуговування клієнтів. Спосіб задає клас політики. Домовимося, що кожен клас політики має статичний метод *doAction*.

```
// класи ПОЛІТИК інкапсулюють функціональність
class Feed
{ public:
      static const char * doAction() { return "Feeding"; }
};
class Stuff
{ public:
      static const char * doAction() { return "Stuffing"; }
};
```

Тепер гнучкий шаблон закладу харчування:

```
template <class Guest, class Action, class Traits = GuestTraits<Guest> >
class SnackBar
{
// різновид закладу харчування, що уточнює СПОСІБ обслуговування гостей
// вважається, що цей спосіб задано статичним методом класу Action
      typedef typename Traits::beverage_type beverage_type;
      typedef typename Traits::snack_type snack_type;
      Guest theGuest;
      beverage_type bev;
      snack_type snack;
public:
```

```
        SnackBar(const Guest& g): theGuest(g) {}
        void entertain()
        {     std::cout << "*SnackBar*   " << Action::doAction() << ' '
                    << theGuest << " serving " << bev << " and " << snack << '\n';
        }
};
```

Його використання і отримане виведення:

```
        SnackBar<Boy,Feed> R1(boy);
        R1.entertain();
```

```
*SnackBar*   Feeding Boy Mathew serving Milk and Cookies
```

Як підсумок. Пов'язування об'єктів з атрибутами чи функціональністю за допомогою характеристик і політик робить програму гнучкішою та полегшує її розширення. Списки параметрів шаблонів залишаються невеликими і читабельними. Якщо б з кожним «гостем» було асоційовано декілька десятків типів і методів, було б дуже складно вказувати їх усіх при кожному оголошенні *SnackBar*. Об'єднання пов'язаних оголошень у окремі класи суттєво спрощує ситуацію.

Проект "інтелектуального" шаблону

Напередодні цієї лекції слухачі курсу отримали таке ***завдання***:

Виникла потреба створити шаблон функції для виведення в потік довільного контейнера, що вміє надавати ітератори на свій вміст. Шаблон повинен бути досить інтелектуальним: виведення потрібно оздобити відповідно до контейнера.

*Наприклад, перед виведенням вектора (vector) в потоці має з'явитися гасло "*** ВЕКТОР ***", а кожен елемент вектора при виведенні треба обрамляти квадратними дужками: [10] [20] [30]. Якщо вектор порожній, то замість елементів друкують "[порожній]" (або "[empty]").*

*Виведення списку (list) мало б виглядати так: "*** Двозв'язний Список *** <10> <20> <30>"*

*Виведення множини (set) – "*** МНОЖИНА *** {10} {20} {30}"*

Словом, є атрибути – гасло і обрамлення, – які залежать від контейнера. Проблема: як їх повідомити шаблонові функції виведення? Перевірки типу в шаблоні – найгірший зі способів, адже неможливо передбачити наперед всі типи, з якими шаблон працюватиме. Не хотілося б робити у шаблоні більше, ніж два параметри. Було б добре, якби атрибути вибиралися автоматично відповідно до заданого контейнера.

Здається, ми вже знаємо, як виконати це завдання: потрібно оголосити і використати класи характеристик контейнерів. Символи обрамлення елементів контейнера – значення типу *char*. Це інтегральний тип, тому їх можна задати статичним членом класу характеристик. Гасло – рядок. Його може повертати статичний метод класу. Здається, все ясно, можна починати. Але є ще одна особливість, якої не було в попередньому проекті. Там усі класи (гостей, напоїв, закусок) були конкретними, а контейнери – шаблони. Ця обставина накладає певні обмеження, які доведеться враховувати. Ми зробимо це двома різними способами.

Характеристики класів контейнерів

Ось перша спроба оголосити шаблон функції виведення контейнера в потік:

```
// "ієрархія" класів характеристик
template <class Container> class ContainerTraits;
// конкретизацій для вектора - декілька: для кожного типу елементів
template <> class ContainerTraits<std::vector<int>>
{ public:
        static const char * GetTitle() { return " * Vector of Integers *"; }
```

```
};
template <> class ContainerTraits<std::vector<double>>
{ public:
      static const char * GetTitle() { return " * Vector of Reals *"; }
};
// шаблон функції працює з нешаблонним типом контейнера
template <class Container, class Traits = ContainerTraits<Container>>
void PrintContainer(const Container& S)
{
      typename Container::const_iterator it = S.cbegin();
      cout << Traits::GetTitle() << '\n';
      while (it != S.cend())
      {     std::cout.width(8); cout << *it++;     }
      cout << '\n';
}

// приклад використання
      vector<int> A; vector<double> D; . . .
      PrintContainer(A);
      PrintContainer(D);
```

З точки зору шаблона *PrintContaiter* тип *Container* – якийсь конкретний тип, тому класи характеристик доводиться записувати *для кожної* конкретизації шаблона контейнера. У фрагменті вище це зроблено для *vector<int>* та *vector<double>*. Зрозуміло, що це не дуже красиве рішення і зовсім не універсальне. Його можна використовувати в простих випадках, коли наперед відомі типи всіх конкретизацій всіх контейнерів. Загальний розв'язок мав би враховувати, що тип елементів контейнера може бути довільним. Так ми приходимо до думки про *шаблони класів характеристик*.

Шаблони характеристик класів контейнерів

Якщо тип контейнера – шаблон, параметризований типом елементів, то й тип класу характеристик може бути шаблоном, що параметризований тим самим типом. Основою для всіх характеристик стане шаблон з двома параметрами: типом даних і типом контейнера. Таким чином, характеристика всіх векторів – часткова спеціалізація, у якій тип контейнера задано, а тип даних залишається довільним. За бажання для окремих типів даних можна визначати повні спеціалізації. Таким чином нове рішення не звужує можливостей попереднього.

```
// характеристики контейнерних КЛАСІВ, часткова спеціалізація для конкретних класів
template <class T, class Succession> class SuccessionTraits;
/* тут Succession - довільний клас: можливо, шаблонний, а може й ні, T - тип даних, які
зберігаються в контейнері, він нам буде потрібний у спеціалізаціях. Сам шаблон - порожній,
оскільки слугує основою для сім'ї шаблонів-спеціалізацій */

// клас характеристик вектора, часткова спеціалізація для Succession = std::vector<T>
// параметр T залишається вільним, оскільки спеціалізація стосується всіх векторів
template <class T> class SuccessionTraits<T, std::vector<T>>
{ public:
      static const char * GetTitle() { return " * Vector *"; }
};
// повна спеціалізація для вектора значень цілого типу
template <> class SuccessionTraits<int, std::vector<int>>
{ public:
      static const char * GetTitle() { return " * Vector of Integers *"; }
};

// клас характеристик списку, часткова спеціалізація для Succession = std::list<T>
template <class T> class SuccessionTraits<T, std::list<T>>
{ public:
      static const char * GetTitle() { return " * List *"; }
```

```
};
// клас характеристик дека, часткова спеціалізація для Succession = std::deque<T>
template <class T> class SuccessionTraits<T, std::deque<T>>
{ public:
      static const char * GetTitle() { return " * Deque *"; }
};

// клас характеристик множини, часткова спеціалізація для Succession = std::set<T>
template <class T> class SuccessionTraits<T, std::set<T>>
{ public:
      static const char * GetTitle() { return " -*- Set -*-"; }
};

template <class T, class Succession, class Traits = SuccessionTraits<T, Succession>>
void PrintSuccession(const Succession& S)
{
      typename Succession::const_iterator it = S.cbegin();
      cout << Traits::GetTitle() << '\n';
      while (it != S.cend())
      {     std::cout.width(8); cout << *it++;     }
      cout << '\n';
}
```

Нова функція виведення майже нічим не відрізняється від попередньої, але зміниться її використання. У її оголошенні єдиний параметр *S* не залежить від типу *T* елементів контейнера, тому його доведеться вказувати явно.

```
vector<int> A; vector<double> D; set<int> S; . . .
PrintSuccession<int>(A);
PrintSuccession<double>(D);
PrintSuccession<int>(S);
```

Давайте продовжимо наші вправи і спробуємо написати шаблон, який не висуває таких вимог.

Шаблони характеристик шаблонів контейнерів

Якщо ми хочемо пов'язати клас характеристик з шаблоном контейнера, доведеться явно вказати, що параметром характеристики є саме шаблон. При цьому важливо вказати точну структуру параметрів цього шаблона. Тому заглянемо в документацію і уточнимо, як саме оголошено послідовні контейнери бібліотеки STL. Тепер характеристики і шаблон функції виведення матимуть вигляд:

```
// характеристики ШАБЛОНІВ контейнерних класів
template <template <class T, class Allocator = std::allocator<T>> class Succession>
    class SuccTraits;
/* зручніший у використанні, бо тепер T стає частиною оголошення контейнера
   але підходить не для всіх контейнерів, а лише для тих, чиї шаблони мають
   відповідний набір параметрів
*/
// спеціалізації для шаблонів стандартних послідовних контейнерів
template <> class SuccTraits<std::vector>
{ public:
      static const char * GetTitle() { return " +++ VECTOR +++"; }
      static const char leftBound = '[';
      static const char rightBound = ']';
};
template <> class SuccTraits<std::list>
{ public:
      static const char * GetTitle() { return " +++ DOUBLE-LINKED LIST +++"; }
      static const char leftBound = '<';
      static const char rightBound = '>';
};
```

```
template <> class SuccTraits<std::deque>
{ public:
	static const char * GetTitle() { return " +++ DEQUE +++"; }
	static const char leftBound = '\\';
	static const char rightBound = '/';
};
template <> class SuccTraits<std::forward_list>
{ public:
	static const char * GetTitle() { return " +++ SINGLE-LINKED LIST +++"; }
	static const char leftBound = '|';
	static const char rightBound = '|';
};

template <class E, template <class T, class Allocator = std::allocator<T>> class Succession,
    class Traits = SuccTraits<Succession> >
void PrintSucc(const Succession<E>& S)
{
	typename Succession<E>::const_iterator it = S.cbegin();
	cout << Traits::GetTitle() << '\n';
	if (S.empty())
	{	cout << Traits::leftBound << " empty " << Traits::rightBound << '\n';
		return;
	}
	while (it != S.cend())
	{	cout << Traits::leftBound << *it++ << Traits::rightBound << ' ';	}
	cout << '\n';
}
```

Використання дуже просте:

```
vector<int> A; vector<double> D; . . .
PrintSucc(A); PrintSucc(D);
```

На жаль, оновлений шаблон не підходить для множин чи мультимножин, оскільки ті мають у оголошенні інший склад параметрів шаблона.

Наведемо ще декілька прикладів використання характеристик.

Рядок, нечутливий до регістра

За допомогою зміни класу характеристик символів можна легко змінити поведінку стандартного рядка бібліотеки STL. Справа в тому, що звичний нам тип *string* насправді є спеціалізацією загального шаблона *basic_string* для типу *char*. Загальний шаблон описує алгоритми опрацювання рядків, незалежні від множини символів. Логіку порівняння символів відокремлено від алгоритмів. Її описують класи характеристик *char_traits*. Така архітектура робить шаблон *basic_string* готовим до використання нових типів літер. Достатньо описати їхні характеристики, і ви отримаєте рядок хоч би й 8-байтових символів (таких поки що нема). Тим часом ми визначимо новий клас характеристик для стандартного типу *char* і отримаємо новий тип рядка, що не робить різниці між малими і великими латинськими літерами. Треба сказати, що клас характеристик символів досить об'ємний, тому ми не будемо писати його весь, а застосуємо наслідування і перевизначимо тільки окремі методи.

```
struct ichar_traits: char_traits<char>
{
	static bool eq(char a, char b)
	{	return toupper(a) == toupper(b);
	}
	static bool ne(char a, char b)
	{	return !eq(a,b);
	}
	static bool lt(char a, char b)
	{	return toupper(a) < toupper(b);
	}
```

```
        static int compare(const char * str1, const char * str2, size_t n)
        {       for (size_t i=0; i<n; ++i)
                {       if (str1 == 0) return -1;
                        if (str2 == 0) return 1;
                        if (tolower(*str1) < tolower(*str2)) return -1;
                        if (tolower(*str1) > tolower(*str2)) return 1;
                        assert(tolower(*str1) == tolower(*str2));
                        ++str1; ++str2;
                }
                return 0;
        }
        static const char * find(const char * str, size_t n, char c)
        {       while (n-->0)
                {       if (toupper(*str) == toupper(c)) return str;
                        ++str;
                }
                return 0;
        }
};
```

Тепер новий тип рядка можна оголосити як нову спеціалізацію загального шаблона. Створити його екземпляри можна так само як для стандартного типу.

```
typedef basic_string<char, ichar_traits> istring;

istring first = "Ukraine";
istring second= "UKRAINE";
// first == second
```

Обмеження конкретизацій шаблону за допомогою характеристик

У бібліотеці STL є тип для відображення комплексних чисел – це шаблон *complex<T>*, де *T* – дійсний тип. На практиці використовують *complex<float>*, або *complex<double>*, залежно від вимог точності. Проте мова не накладає обмежень на способи конкретизації цього шаблону. Ви можете оголосити *complex<int>* і навіть, *complex<string>*. Комплексні «числа» з рядків можна буде навіть вводити з клавіатури, виводити в потік і «додавати» (в сенсі конкатенації), але будь-яка інша дія спричинить помилку компіляції. Це зрозуміло, бо для рядків не визначено, наприклад, оператор множення чи віднімання. А хотілося б мати якийсь засіб, який би застеріг програміста від таких необдуманих кроків. Спробуємо ввести обмеження на можливі конкретизації шаблона раціональних чисел за допомогою класів характеристик.

Множину раціональних чисел визначено так: $\mathbb{Q} = \left\{ \frac{p}{q} \middle| p \in \mathbb{Z}, q \in \mathbb{N} \right\}$, тобто раціональне число можна зобразити парою цілих чисел. Залежно від потрібної кількості десяткових цифр у чисельнику і знаменнику такого числа, можна використовувати різні цілі типи: *short int*, *int*, *long long int*, класи «довгих цілих». Використання дійсних чи нечислових типів не має сенсу. Вберегти шаблон від неправильної конкретизації можна за допомогою характеристик. Наприклад, конструктор раціонального числа міг би використовувати стандартні значення, які визначено в характеристиках. Для всіх допустимих класів такі характеристики треба визначити, а всі інші спричинятимуть помилку під час компіляції.

```
// Раціональне число реалізовано шаблоном класу Frac<TNum>, де
// TNum - тип цілого числа. Для того, щоб запобігти конкретизаціям
// невідповідним типом, використано класи характеристик.
// Цілий тип характеризують такі атрибути: NumTag - розмір у байтах,
// numerator, denominator - значення за замовчуванням чисельника і знаменника

template<typename TNum> class NumTraits;
```

```
template<> class NumTraits<short>
{ public:
	const int NumTag = 2;
	static const short numerator = 0;
	static const short denominator = 1;
};
template<> class NumTraits<int>
{ public:
	const int NumTag = 4;
	static const int numerator = 0;
	static const int denominator = 1;
};
template<> class NumTraits<long long>
{ public:
	const int NumTag = 8;
	static const long long numerator = 0LL;
	static const long long denominator = 1LL;
};
// обчислення найбільшого спільного дільника за
// алгоритмом Евкліда
template<typename TNum>
TNum GCD(TNum a, TNum b)
{	a = a < 0 ? -a : a;
	while (a != b)
		if (a > b) a -= b;
		else b -= a;
	return b;
}
// раціональне число описує шаблон структури
template<typename TNum, typename Traits = NumTraits<TNum>>
struct Frac
{	TNum num; TNum den;
	Frac() : num(Traits::numerator), den(Traits::denominator) {}
	Frac(TNum n, TNum d)
	{	d = (d == Traits::numerator) ? Traits::denominator : d;
		if (n == Traits::numerator)
		{	num = n; den = Traits::denominator;	}
		else
		{	TNum gcd = GCD(n, d);
			num = n / gcd; den = d / gcd;
		}
	}
	static int FracSize() { return NumTraits<TNum>::NumTag * 2; }
};
// усі оператори задано зовнішніми функціями
template<typename TNum> Frac<TNum> operator+(Frac<TNum> a, Frac<TNum> b)
{	return Frac<TNum>(a.num*b.den + b.num*a.den, a.den*b.den);
}
template<typename TNum> Frac<TNum> operator*(Frac<TNum> a, Frac<TNum> b)
{	return Frac<TNum>(a.num*b.num, a.den*b.den);
}
template<typename TNum> std::ostream& operator<<(std::ostream& os, const Frac<TNum>& f)
{	os << f.num << '/' << f.den; return os;
}
template<typename TNum> std::istream& operator>>(std::istream& is, Frac<TNum>& f)
{	is >> f.num >> f.den; return is;
}
```

Звичайно, в оголошенні типу *Frac* можна було б обійтися і без характеристик, але саме таким способом – через звертання до них у кожному конструкторі – ми й можемо обмежити конкретизації шаблону лише тими класами, для яких такі характеристики визначено.

```
int main()
{
	Frac<short> fd; cout << fd << '\n';
```

```
	Frac<short> fs(1, 2), rs(-1, 3);
	cout << fs << " + " << rs << " = " << fs + rs << '\n';
	Frac<long long> fll(123456, 339966);
	cout << fll << '\n';
	//Frac<char> fc;   // raise error of compilation
	return 0;
}
```

Інші корисні приклади використання характеристик можна знайти у [Еккель, Еллісон]: *numeric_limits* ст. 226-227, *iterator_traits* ст. 284-285, використання характеристик в шаблонах потоків ст. 136.

Програмування на етапі компіляції

Шаблони задумували як удосконалення макровизначень, які забезпечують швидкий і безпечний спосіб заміни загального типу конкретним. Досвід використання шаблонів підказав, що крім цього корисними будуть параметри шаблонів цілого типу та явні спеціалізації. І от, як це часто буває, розроблені засоби отримали ширші можливості, ніж було задумано. Проаналізуйте, як компілятор збудує конкретизацію *Factorial<5>* оголошеного нижче шаблона.

```
template<long long n>
struct Factorial
{
	enum { val = Factorial<n - 1>::val * n };
};
template<> struct Factorial<1> { enum { val = 1 }; };
```

Неймовірно, але інструкція *cout << Factorial<5>::val;* виведе на друк 120 – правильне значення 5!. Ще неймовірніше те, що обчислення відбудуться *на етапі компіляції*. Як таке можливо? Щоб збудувати *Factorial<5>*, компілятор потребує *Factorial<4>::val*, а для цього він почне побудову *Factorial<4>*, для якого йому потрібно *Factorial<3>*, *Factorial<2>*, для якого він отримає *Factorial<1>::val = 1* зі спеціалізації шаблона. Тоді компілятор легко визначить *Factorial<2>::val = 2*, *Factorial<3>::val = 6*, *Factorial<4>::val = 24*, *Factorial<5>::val = 120*, після чого компіляція завершиться. Послідовність факторіалів від 1! до 5! обчислено на етапі компіляції. Щоб виконати інструкцію виведення достатньо взяти готове число. Розмір виконуваного файла, звісно, збільшився, але для обчислень на етапі виконання потрібно 0 секунд.

З'ясувалося, що за допомогою шаблонів можна виконувати повторювані обчислення (за допомогою рекурсії, як у прикладі вище) та галуження (за допомогою спеціалізації шаблонів для значень *true* і *false*). Таким чином шаблони мови C++ володіють алгоритмічною повнотою за Тьюрінгом. Теоретично, з їхньою допомогою можна виконувати обчислення довільної складності. З примітивного засобу підставляння імен *template* перетворилися на інструмент *шаблонного метапрограмування*. Придивіться до прикладів метапрограм, зокрема, як реалізовано умовне виконання (галуження).

```
// Рекурсія за допомогою константи всередині структури
// (накопичення шкідливих викликів не відбувається
template<int n> struct Fib {
	enum { val = Fib<n-1>::val + Fib<n-2>::val };
};
template<> struct Fib<1> { enum { val = 1 }; };
template<> struct Fib<0> { enum { val = 1 }; };

// Арифметичний цикл через рекурсію
template<int N, int P> struct Power {
	enum { val = N * Power<N, P-1>::val };
};
template<int N> struct Power<N,0> {
	enum { val = 1 }; };
```

```
// Просте галуження
template<int a, int b> struct Max {
	enum { val = a>b ? a : b }; };

// Вибір способу виконання запрограмувати складніше
template<bool cond> struct Select {};

template<> struct Select<false> {
	static inline void f() {
		cout << "False statement executing\n";
	}
};
template<> struct Select<true> {
	static inline void f() {
		cout << "True statement executing\n";
	}
};
template<bool cond> void execute() {
	Select<cond>::f();
}

int main() {
// Числа Фібоначчі
	cout << "Fibonacci numbers\n\n";
	cout << "\tF( 5) = " << Fib<5>::val << '\n';
	cout << "\tF(35) = " << Fib<35>::val << "\n\n";
// Піднесення цілого до цілої степені
	cout << "Integer power\n\n";
	cout << "\t 2^5 = " << Power<2, 5>::val << '\n';
	cout << "\t2^10 = " << Power<2, 10>::val << "\n\n";
// Більше з двох цілих
	cout << "Max value\n\n";
	cout << "\tmax(10,20) = " << Max<10, 20>::val << '\n';
	cout << "\tmax(F(10),2^10) = " << Max<Fib<10>::val, Power<2, 10>::val>::val << "\n\n";
// Умовне виконання
	cout << "A condition dependent executing\n\n";
	cout << "\tif 2>5 then - "; Select<(2>5)>::f(); cout << '\n';
	cout << "\tif sizeof(int)=4 then - "; execute<sizeof(int) == 4>(); cout << "\n\n";
	cin.get();
	return 0;
}
```

Отримані результати:

```
Factorials
	 0! = 1
	12! = 479001600

Fibonacci numbers
	F( 5) = 8
	F(35) = 14930352

Integer power
	 2^5 = 32
	2^10 = 1024

Max value
	max(10,20) = 20
	max(F(10),2^10) = 1024

A condition dependent executing

	if 2>5 then - False statement executing

	if sizeof(int)=4 then - True statement executing
```