Шаблоны в C++

С помощью шаблонов (templates) можно создавать родовые (обобщённые) функции (generic functions) и родовые (обобщённые) классы (generic classes). В родовой функции или классе тип(ы) данных, с которыми функция или класс работают, задается в качестве параметра.

Шаблон представляет собой функцию (template function) или класс (template class), реализованные для одного или нескольких типов данных, которые не известны в момент написания кода. При использовании шаблона, в качестве аргументов ему явно или неявно передаются конкретные типы данных. Поскольку шаблоны являются средствами языка, для них обеспечивается полная поддержка проверки типов и областей видимости.

Таким образом, шаблоны дают возможность создавать многократно используемые программы.

Поскольку шаблоны являются логическим развитием механизма макроподстановок, то данные о типах, например длина операндов, подставляется на этапе компиляции (точнее, даже еще раньше). На момент выполнения все длины операндов, размеры элементов массивов и прочие величины уже вычислены, так что процессор работает с хорошо оптимизированным прямолинейным кодом.

Шаблоны функций

Обобщенная функция (шаблон функции – template function) определяет универсальную совокупность операций, применимых к различным типам данных. Тип данных, с которыми работает функция, передается в качестве параметра.

Многие алгоритмы носят универсальный характер и не зависят от типа данных, которыми они оперируют. Например, для массивов целых и действительных чисел используется один алгоритм быстрой сортировки. С помощью обобщенной функции можно определить алгоритм независимо от типа данных. После этого компилятор автоматически генерирует правильный код, соответствующий конкретному типу. По существу, обобщенная функция автоматически перегружает саму себя.

Обобщенная функция объявляется с помощью ключевого слова template. Создается шаблон, который описывает действия функции и позволяет компилятору самому уточнять необходимые детали.

Определение шаблонной функции выглядит следующим образом:

```
template <class Ттип> <тип_возвращаемого_значения>
<имя_функции> (<список_параметров>)
{
// Тело функции
}
```

Здесь параметр Ттип – имя типа – задает тип данных, с которым работает функция. Этот параметр можно использовать и внутри функции, однако при создании конкретной версии обобщенной функции компилятор автоматически подставит вместо него фактический тип. Традиционно обобщенный тип задается с помощью ключевого слова class, хотя вместо него можно применять ключевое слово typename.

Шаблоны функций

Обобщенная функция, меняющая местами две переменные.

Поскольку процесс перестановки не зависит от типа переменных, его можно описать с помощью обобщенной функции.

```
// Пример шаблонной функции
#include <iostream>
using namespace std;
// Шаблон функции
template <class X>
        void swapargs(X &a, X &b)
{
 X temp;
 temp = a;   a = b;   b = temp;
}
```

см. продолжение

```
int main()
{ int i=10, j=20;
  double x=10.1, y=23.3;
  char  a='x'.  b='z';
cout << "Исходные значения i, j: " << i << ' '
                          << j << '\n';
cout << "Исходные значения x, y: " << x <<
                    ' ' << y << '\n';
cout << "Исходные значения a, b:   " << a <<
                        '  ' << b << '\n';
swapargs  (i,  j);  // Перестановка целых чисел
swapargs(x, y);   // Перестановка действительных
                  // чисел
swapargsfa, b);   // Перестановка символов.
cout << "Переставленные значения i,  j:   "
          << i <<'  ' << j <<'\n';
cout << "Переставленные значения x, y:  "
          << x << ' ' << y <<'\n';
cout << "Переставленные значения a, b:   "
<< a <<  '  ' << b <<  ' \n';
return 0;
}
```

продолжение

Строка

template <class X> void swapargs(X &a, X &b)

сообщает компилятору, что: во-первых, создается шаблон, во-вторых, начинается описание обобщенной функции.

Здесь параметр X задает обобщенный тип, который впоследствии будет заменен фактическим типом. После этой строки объявляется функция swapargs(), в которой переменные, подлежащие перестановке, имеют обобщенный тип X. В функции main() функция swapargs() вызывается для трех разных типов данных: int, double и char. Поскольку функция swapargs() является обобщенной, компилятор автоматически создает три ее версии: для перестановки целых и действительных чисел, а также символов.

Шаблоны функций

Уточним понятия связанные с шаблонами.

Обобщенная функция (т.е. функция, объявленная с помощью ключевого слова template) называется также **шаблонной функцией** (**template function**). Эти термины являются синонимами.

Конкретная версия обобщенной функции, создаваемая компилятором, называется **специализацией** (**specialization**) или **генерируемой функцией** (**generated function**). **Процесс генерации конкретной функции** называется **конкретизацией** (**instantiation**). Генерируемая функция является конкретным экземпляром обобщенной функции.

Поскольку язык C++ не считает конец строки символом конца оператора, раздел template в определении обобщенной функции не обязан находиться в одной строке с именем функции.

Такую особенность иллюстрирует следующий пример:

```
template <class X>
    void swapargs(X &a,  X &b)
      {
       X temp;
       temp = a; a = b; b = temp;
      }
```

НО, в этом случае следует помнить, что между оператором template и началом определения обобщенной функции не должно быть других операторов. Например, следующий фрагмент ошибочен.

```
// Этот фрагмент содержит ошибку:
template <class X>
   int i;  // Ошибка
   void swapargs(X &a, X &b)
{
X temp;
temp = a; a = b; b = temp;
}
```

Т.е., спецификация template должна непосредственно предшествовать определению функции.

Шаблоны функций

Функция с двумя обобщенными типами

Используя список, элементы которого разделены запятыми, можно определить несколько обобщенных типов данных в операторе template.

Например, в следующей программе создается шаблонная функция, имеющая два обобщенных типа.

```
#include <iostream>
using namespace std;

template <class typel,  class type2>
 void myfunc(typel  x,  type2  y)
{
cout  <<  x <<'   ' <<  y  <<  '\n';
}

int main()
 {
   myfunc(10, "Я люблю C++");
   myfunc(98.6,   19L);
  return 0;
}
```

Шаблоны функций

Перегрузка шаблонной функции

Шаблонная функция может быть перегружена другим шаблоном или нешаблонной функцией с таким же именем, но с другим набором параметров.

```
#include <iostream>
using namespace std;
// шаблон для сравнения на равенство двух значений
template   <class T>
  static  bool   eq (const T &x, const T &y)
{ return x == y ; }
// шаблон для сравнения на равенство двух массивов
template  <class T>  static  bool  eq
   (const T *px, const T *py, const int size)
{
for ( int i = 0; i < size; i++ )
  if (! eq (px[i], py[i]))
    return false;
  return true ;
}
// не шаблонная функция для сравнения строк на равенство
static bool eq (const char *sl, const char *s2)
   {  return ! strcmp (si, s2);  }
```

// см. продолжение

// продолжение

```
int main ( )
{
  bool  b1 [ ] = { true, false, false, true };
  bool b2 [ ] = { true, false, false, true };
char  s [ ] = "C++\x0"  ;
char  c1 ('M'), c2  ('m') ;
 cout <<eq (c1, c2)<<endl;    //выводится 0
cout <<eq (b1,b2,4)<<endl;  //выводится 1
cout<<eq ("C++",s)<<endl; //выводится 1
 return 0 ;
}
```

Шаблоны функций

Перегрузка шаблонной функции

Шаблон функции eq() перегружен двумя способами: другим шаблоном и явной функцией.

Первая шаблонная функция возвращает true, если два значения одного и того же типа, заданного параметром типа T и формальными параметрами ссылок на тип T, равны. Иначе результат отношения x = = y будет ложным, и функция вернет false.

Второй шаблон проверяет все элементы двух массивов на равенство и возвращает true только при совпадении значений всех элементов. Первая пара элементов, где разные значения, приводит к выходу из функции со значением false. В теле шаблона для сравнения элементов вызывается первая шаблонная функция. Формальные параметры представлены указателями на параметр типа T, которые при вызове должны содержать адреса сравниваемых массивов, и переменной целого типа, которая задает их размер.

Для сравнения строк существует стандартная функция strcmp(), поэтому шаблон перегружен явной функцией, которая будет вызываться при обращении с типом параметров char*. Библиотечная функция strcmp() возвращает 0 при равенстве строк, поэтому в операторе return использовано логическое отрицание результата библиотечной функции.

В главной функции перегруженные функции тестируются на примере двух символьных переменных c1 и c2, массивов логического типа b1 и b2, а также двух строк, одна из которых задана литералом "C++", а другая инициализирована через указатель s.

Шаблоны функций

Использование стандартных параметров шаблонных функций

При определении шаблонной функции можно смешивать стандартные и обобщенные параметры. В этом случае стандартные параметры ничем не отличаются от параметров любых других функций. Рассмотрим пример.

```
// Применение стандартных параметров в
//                                  шаблонной функции
#include <iostream>
using namespace std;
const int TABWIDTH = 8;
// Выводит на экран данные в позиции   tab
template  <class X>  void tabOut (X data,  int tab)
 {
   for(;   tab;   tab--)
     for(int i=0;   i<TABWIDTH;   i++)
       cout << '  ';
    cout << data << "\n";
}

int main()
  {
    tabOut("Проверка",  0);
    tabOut(100, 1);
    tabOut('X',  2);
    tabOut(10/3,3);
   return 0;
  }
```

Программа выводит на экран следующие сообщения:

```
Проверка
        100
          X
3
```

В этой программе функция tabOut() выводит на экран свой первый аргумент, позиция которого определяется вторым параметром tab. Поскольку первый аргумент имеет обобщенный тип, функция tabOut() может выводить на экран данные любого типа. Параметр tab является стандартным и передается по значению. Смешение обобщенных и стандартных параметров не вызывает никаких проблем и может оказаться полезным.

Шаблоны функций

Ограничения на обобщенные функции

Обобщенные функции напоминают перегруженные, но на них налагаются еще более жесткие ограничения. При перегрузке внутри тела каждой функции можно выполнять разные операции. В то же время обобщенная функция должна выполнять одну и ту же универсальную операцию для всех версий, различаться могут лишь типы данных.

РЕЗЮМЕ

Шаблонная функция (template function) – это функция, полностью контролирующая соответствие типов данных, которые задаются ей как параметры.

Объявление шаблона функции повторяет определение обычной функции, но первой строкой в объявлении обязательно должна быть строка следующего вида:

template <class имя_типа1, …, class имя_типаN >

В угловых скобках < > после ключевого слова template указывается список формальных параметров шаблона. Имя_типа называют параметром типа, который представляет собой идентификатор типа и используется для определения типа параметра в заголовке шаблонной функции, типа возвращаемого функцией значения и типа переменных, объявляемых в теле функции. Каждый параметр типа должен иметь уникальный идентификатор, который может быть использован в разных шаблонах. Каждому параметру типа должно предшествовать ключевое слово class.

За строкой template следует строка со стандартным объявлением функции, где в списке параметров используются как параметры типа, так и любые другие допустимые базовые или производные типы данных.

Шаблонная функция может быть перегружена другим шаблоном или не шаблонной функцией с таким же именем, но с другим набором параметров.

Компилятор выбирает вариант функции, соответствующий вызову. Сначала подбирается функция, которая полностью соответствует по имени и типам параметров вызываемой функции. Если попытка заканчивается неудачей, подбирается шаблон, с помощью которого можно сгенерировать функцию с точным соответствием типов всех параметров и имени. Если же и эта попытка неудачна, выполняется подбор перегруженной функции.

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

Сортировка представляет собой типичный пример универсального алгоритма. Как правило, алгоритм сортировки совершенно не зависит от типа сортируемых данных.

Функция bubble (), предназначенная для сортировки данных методом пузырька, упорядочивает массив любого типа. Ей передается указатель на первый элемент массива и количество элементов в массиве. Несмотря на то что этот алгоритм сортировки является одним из самых медленных, он очень прост и нагляден.

```
#include <iostream>
using namespace std;
  template <class X>
                    void bubble (
                            X *items,  // Указатель на упорядочиваемый массив
                            int count)  // Количество элементов массива
   {
    register  int a, b;
    X t;
     for(a=1; a<count; a++)
       for(b=count-1; b>=a; b--)
          if(items[b-1]  > items[b])
             // Перестановка элементов
             {  t = items[b-1];  items[b-1] = items[b];  items[b] = t; }
   }
```

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

```
int main()
 {
   int iarray[7]  = {7,  5,  4,  3,  9,  8,  6} ;
   double darray[5]  = {4.3,  2.5,  -0.9,  100.2, 3.0};
   int i;
  cout << "Неупорядоченный массив целых чисел: ";
    for(i=0;  i<7; i++)
      cout << iarray[i] <<  '  ';
    cout << endl;
  cout << "Неупорядоченный массив действительных чисел: ";
    for(i=0;  i<5; i++)
      cout << darray[i] <<   ' ';
    cout << endl;
  bubble(iarray, 7);    bubble(darray,  5);
  cout « "Упорядоченный массив целых чисел: ";
    for(i=0;   i<7; i++)
      cout << iarray[i] << '   ';
    cout << endl;
   cout << "Упорядоченный массив действительных чисел: ";
     for(i=0;  i<5; i++)
       cout << darray[i]  <<  '  ';
     cout << endl;
 return 0;
 }
```

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

Программа выводит на экран следующие результаты:

- **Неупорядоченный массив целых чисел: 7 5 4 3 9 8 6**
- **Неупорядоченный массив действительных чисел: 4.3 2.5 -0.9 100.2 3**
- **Упорядоченный массив целых чисел: 3 4 5 6 7 8 9**
- **Упорядоченный массив действительных чисел: -0.9 2.5 3 4.3 100.2**

Как видим, в программе создаются два массива, имеющие соответственно типы int и double.

Поскольку функция bubble() является шаблонной, она автоматически перегружается для каждого из этих типов.

Эту функцию можно применить для сортировки данных другого типа, даже объектов какого-нибудь класса. Компилятор автоматически создаст соответствующую версию функции для нового типа.

Шаблоны функций

Применение обобщенных функций. Уплотнение массива

Функция compact()

Эта функция уплотняет элементы массива.

Довольно часто приходится удалять несколько элементов из середины массива и перемещать оставшиеся элементы влево, заполняя образовавшуюся пустоту.

Эта операция носит универсальный характер и не зависит от типа массива.

Обобщенная функция compact () получает указатель на первый элемент массива, количество его элементов, а также индексы первого и последнего элементов удаленного отрезка массива.

Затем функция удаляет указанные элементы и уплотняет массив. Неиспользуемые элементы, образующиеся при уплотнении массива, обнуляются.

Шаблоны функций

Применение обобщенных функций. Уплотнение массива

Функция compact()

```
#include <iostream>
using namespace std;
template <class X>
  void compact (
   X *items, //Указатель на уплотняемый
                    // массив
   int count, // Количество элементов
                    // массива
   int start,  // Индекс первого удаленного
                    // элемента
    int end)   // Индекс последнего удален-
                    // ного элемента
{
   register  int  i;
  for (i=end+1; i<count; i++, start++)
      items[start]  = items[i];
/*  Для иллюстрации оставшаяся часть
массива заполняется нулями. */
   for(; start<count; start++)
      items[start]  =  (X) 0;
}
```

см. продолжение

продолжение

```
int main()
{
  int nums[7] = {0, 1, 2, 3, 4, 5, 6);
  char str [18]="Обобщенные функции";
  int i;
cout << "Неуплотненная часть целочислен-
                  // ного массива: ";
 for(i=0;   i<7; i++)
    cout <<  nums[i]  <<   b' ';
 cout << endl;
cout << "Неуплотненная  строка:  ";
 for(i=0;   i<18; i++)
   cout << str[i] << '  ';
 cout << endl;
compact (nums,  7,  2,  4);
compact (str,  18,  6, 10);
  cout << "Уплотненный целочисленный
                  // массив: ";
  for(i=0;   i<7; i++)
     cout << nums[i] <<   '  ';
   cout << endl;
cout << "Уплотненная  строка: ";
  for(i=0;   i<18; i++)
     cout << str[i] << '  ';
   cout << endl;
return 0;
}
```

Шаблоны функций

Применение обобщенных функций. Уплотнение массива

Функция compact()

Эта программа уплотняет массивы двух типов: целочисленный и строку.
Однако функция compact () может работать с массивом любого типа.

Результаты работы программы:

- Неуплотненный целочисленный массив: 0 1 2 3 4 5 6
- Неуплотненный целочисленный массив:
- Обобщенные функции
- Уплотненный целочисленный массив: 0 1 5 6 0 0 0
- Уплотненная строка: Обобщефункции

Во многих случаях шаблоны являются вполне естественными. Если логика функции не зависит от типа данных, ее можно преобразовать в шаблонную.

Шаблоны классов

В ООП можно определить обобщенные классы (шаблон класса – template class) – это класс, в котором определены все алгоритмы и данные, однако фактический тип данных задается в качестве параметра при создании объекта.

Обобщенные классы полезны, если логика класса не зависит от типа данных. Например, к очередям, состоящим из целых чисел или символов, можно применять один и тот же алгоритм, а для поддержки связанных списков адресов рассылки и мест для парковки автомобилей использовать один и тот же механизм.

Объявление обобщенного класса имеет следующий вид:

```
template <class Ттип> class <имя_класса>
{
  ……….
}
```

Здесь параметр Ттип задает тип данных, который уточняется при создании экземпляра класса. При необходимости можно определить несколько обобщенных типов, используя список имен, разделенных запятой.

Конкретный экземпляр обобщенного класса создается с помощью следующей синтаксической конструкции:

```
<имя_класса> <тип> <имя_объекта>;
```

Здесь параметр тип задает тип данных, которыми оперирует класс. Функции-члены обобщенного класса автоматически становятся обобщенными. Для их объявления не обязательно использовать ключевое слово template.

Шаблоны классов

Шаблоны дают возможность многократно использовать один и тот же код, который позволяет компилятору автоматизировать процесс реализации типа.

Для шаблонных классов характерными являются следующие свойства:

- **Шаблон позволяет передать в класс один или несколько типов в виде параметров. Передаваемым типом может быть любой базовый или производный тип, включая тип class.**
- **Параметрами шаблона могут быть не только типы, но и константные выражения.**
- **Объявление шаблона должно быть только глобальным.**
- **Каждый реализованный шаблон (созданный объект шаблона классов) требует собственного откомпилированного объектного кода. Чтобы сократить размер объектного модуля программы, рекомендуется наследовать наиболее общую часть шаблона от базового класса. Для базового и производного классов в этом случае необходим один и тот же реализованный тип.**
- **Базовый класс для шаблона может быть как шаблонным, так и обычным классом. Обычный класс может быть порожден от реализованного шаблона классов.**
- **Нельзя использовать указатель на базовый шаблонный класс для получения доступа к методам производных классов; т.к. типы, полученные даже из одного и того же шаблона, всегда являются разными.**
- **В описание шаблона классов можно включать дружественные функции. Если функция-друг не использует спецификатор шаблона, то она считается универсальной для всех экземпляров шаблона. Если же в прототипе функции-друга содержится шаблон параметров, то эта функция будет дружественной только для того класса, экземпляр которого создается.**
- **Статические члены-данные специфичны для каждого реализованного шаблона.**

Чаще всего шаблоны классов используются для программирования контейнерных классов (контейнеров), которые предназначены для хранения объектов других классов.

Классы

Создадим обобщенный класс stack

Но прежде вспомним программы для работы со стеком:

Пример_1: простая работа с классом stack в ООП

```
#include <iostream>
#define SIZE 100
using namespace std;
class stack
{
  int stck[SIZE];
  int tos;
 public:
   void init ();
   void push (int i);
   int pop ();
};
//-----------------------------------------------
void stack :: init ()
{  tos = 0;  }
//-----------------------------------------------
void stack :: push (int i)
{ if (tos == SIZE)
    { cout << "Стек полон.\n"; return;  }
   stck[tos] = i;
   tos++;
}                    // см. продолжение
```

```
// продолжение
int stack :: pop ()
{ if (tos == 0)
   { cout << "Стек пуст.\n"; return 0;  }
  tos--;
  return stck[tos];  }
//-----------------------------------------------
int main ()
{   stack    stack1, stack2; /* создаем 2 объекта
класса stack */
    stack1.init ();   // Вызов init для объекта stack1
    stack2.init ();   // Вызов init для объекта stack2
  stack1.push (1);
  stack2.push (2);
    stack1.push (3);
    stack2.push (4);

 cout << stack1.pop() <<  "  ";
 cout << stack1.pop() <<  "  ";
 cout << stack2.pop() <<  "  ";
 cout << stack2.pop() <<  " \n ";
return 0;
}
```

Вывод на экран: 3 1 4 2

Классы

Пример_2: класс stack с конструктором и деструктором

```
#include <iostream>
using namespace std;
#define SIZE 100
class stack
 {
   int stck[SIZE];
   int tos; public:
    stack();       // Конструктор
    ~stack();      // Деструктор
       void push (int i);
       int pop ();
 };
stack::stack()   // Конструктор класса stack
 {
   tos = 0;
   cout << "Стек инициализирован\n";
 }
stack::~stack()    // Деструктор класса stack
 {   cout << "Стек уничтожен\n";   }
void stack: :push(int i)
  {
     if(tos==SIZE)
       {    cout << "Стек полон.\n";   return;  }
     stck[tos] = i; tos++;
  }
int stack::pop ()
 {
   if(tos==0) {cout << "Стек пуст.\n"; return 0; }
   tos--;    return stck[tos];
}                          // см. продолжение
```

```
// продолжение
int main()
 {
  setlocale(LC_CTYPE, "Russian");
  stack a, b; // Создаем два объекта класса stack
     a. push(1);
     b. push(2);
     a. push(3);
     b. push(4);
  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";
    return 0;
}
```

Результат работы программы :

Стек инициализирован
Стек инициализирован
3 1 4 2
Стек уничтожен
Стек уничтожен

(На самом деле класс stack не нуждается в деструкторе, здесь он приведен в качестве иллюстрации.)

Шаблоны классов

Пример: Шаблон класса stack

```
#include <iostream>
using namespace std;
const int SIZE = 10;
// Создаем обобщенный класс stack
template  <class StackType>   class stack
{  StackType stck[SIZE]; // Элементы стека
int tos; // Индекс вершины стека
  public:
stack()   { tos =0;  }  // Инициализирует стек
void  push(StackType  ob);  //Заталкивает объект в
                            // стек.
StackType pop();  // Выталкивает объект из стека
};
    // Заталкиваем объект в стек.
template <class StackType>
  void stack<StackType>::push(StackType ob)
     {  if(tos == SIZE)
        { cout << "Стек полон.\n";  return;  }
        stck[tos] = ob;    tos++;
     }
    // Выталкиваем объект из стека
template <class StackType> StackType
       stack<StackType>::pop()
  {
   if(tos==0)
   // Если стек пуст, возвращается  константа null
      { cout << "Стек пуст.\n";  return 0; }
   tos--;
   return stck[tos];
   }                          // см. продолжение
```

```
                                  // продолжение
int main()
{
// Демонстрация стека символов
stack<char> s1,  s2;  // Создаем два стека символов
int i ;
s1.push('a');      s2.push('x');
s1.push('b');      s2.push('y');
s1.push('c');      s2.push('z');
for(i=0; i<3; i++)
cout <<"Выталкиваем s1: "<< s1.pop()<< "\n";
for(i=0;  i<3;  i++)
cout <<"Выталкиваем s2: "<<s2.pop() << "\n";
// Демонстрация стека действительных чисел
 stack<double> dsl, ds2;  // Создаем два стека
                  // действительных чисел
ds1.push(1.1);   ds2.push(2.2);
ds1.push(3.3);   ds2.push(4.4);
ds1.push(5.5);   ds2.push(6.6);
for(i=0;  i<3;  i++)
cout<<"Выталкиваем ds1:"<<ds1.pop()<<"\n";
for(i=0;  i<3;  i++)
cout<<"Выталкиваем ds2:"<<ds2.pop()<<"\n";
return 0;
}
```

Шаблоны классов

Пример: Шаблон класса stack

Как видно, объявление обобщенного класса мало отличается от объявления обобщенной функции.

Фактический тип данных, размещаемый в стеке, в объявлении класса заменяется обобщенным параметром и уточняется лишь при создании конкретного объекта. При объявлении конкретного объекта класса stack компилятор автоматически генерирует все функции и переменные, необходимые для обработки фактических данных.

В примере на предыдущем слайде объявляются по два стека разных типов – целых чисел и действительных чисел.

Обратите особое внимание на следующие объявления:

```
stack <char>   s1, s2;      // Создаем два стека символов
stack <double> ds1, ds2;    // Создаем два стека действительных чисел
```

Требуемый тип данных задается в угловых скобках. Изменяя этот тип при создании объекта класса stack, можно изменять тип данных, хранящихся в стеке.

Например, можно создать стек для хранения указателей на символы:

```
stack <char *> chrptr();
```

Можно создавать стеки, хранящие объекты, тип которых определен пользователем.

Шаблоны классов

Пример: Использование двух обобщенных типов данных

Шаблонный класс может иметь несколько обобщенных типов. Их следует перечислить в списке шаблонных параметров в объявлении template.

```
/*Пример класса, использующего два обобщенных типа */
#include <iostream>
using namespace std;
  template <class Type1, class Type2>
  class myclass
   {  Type1 i;   Type2 j;
    public:
      myclass(Type1  a, Type2  b)
        { i = a;  j = b; }
      void show()
       { cout<<i<<' '<< j <<  '\n'; )    };
int main() {
  myclass  <int,  double> obi(10, 0.23);
   myclass <char,  char *> ob2('X',
"Шаблоны – мощный механизм.");
ob1.show();     // Выводим целое и действительное число
ob2.show();     // Выводим символ и указатель на символ
return 0;    }
```

Эта программа выводит следующие результаты:

```
10 0.23
X Шаблоны – мощный механизм.
```

В программе объявляются объекты двух типов:

- Объект ob1 использует целые и действительные числа,
- Объект ob2 использует символ и указатель на символ.

В обоих случаях при создании объектов компилятор автоматически генерирует соответствующие данные и функции.

Шаблоны классов

Применение шаблонных классов. Обобщенный массив

Рассмотрим способ, который довольно часто применяется:

Оператор "[]" можно перегрузить. Это позволяет создавать собственные реализации массива, в том числе "безопасные" массивы, предусматривающие проверку диапазона индексов в ходе выполнения программы.

В языке C++ нет встроенной проверки диапазона индексов, поэтому в ходе выполнения программы индекс может выйти за допустимые пределы, не генерируя сообщения об ошибке.

Однако, если создать класс, содержащий массив, и перегрузить оператор "[]", выход индекса за допустимые пределы можно предотвратить.

Комбинируя перегруженный оператор с шаблонным классом, можно создать обобщенный безопасный массив произвольного типа.

Шаблоны классов

Применение шаблонных классов. Обобщенный массив

```
// Пример обобщенного безопасного массива
#include <iostream>
#include <cstdlib>
using namespace std;
const int SIZE = 10;
template <class AType>
  class atype
   {  AType a[SIZE];    public: atype()
   {   register int i;
     for(i=0;   i<SIZE;   i++)  a[i] = i; }
    AType &operator[](int i);     };
// Проверка диапазона для объекта atype
template <class AType>
Atype &atype<AType>::operator[](int i)
 {  if(i<0  ||  i> SIZE-1)
   { cout<<"\nЗначение индекса    ";
     cout<<i<<"  выходит за пределы
           допустимого диапазона\n";
exit(1);
   }
return a[i] ;
 }
```

// см. продолжение

// продолжение

```
int main()
 {
 atype <int> intob; //Целочисленный
                          массив
 Atype <double> doubleob;  // Массив
                  действительных чисел
  int i ;
 cout< "Целочисленный массив:   ";
for(i=0;  i<SIZE;  i++)   intob[i]  = i;
for(i=0;  i<SIZE;  i++)
   cout << intob[i]  << " ";
   cout <<   '\n';
cout<<"массив действительных
                    чисел: ";
for(i=0; i<SIZE; i++)
doubleob[i] = (double) i/3;
for(i=0;  i<SIZE;  i++)
cout<<doubleob[i]<<" ";  cout<< '\n';
intob[12]     =   100;     //   Генерирует
сообщение об ошибке
return 0;
 }
```

Шаблоны классов

Применение шаблонных классов. Стандартные типы

В спецификации шаблона обобщенного класса можно использовать стандартные типы. Т.е., в качестве шаблонных параметров можно применять стандартные аргументы, например, целочисленные значения или указатели. Синтаксис такого объявления не отличается от объявления обычных параметров функций: необходимо лишь указать тип и имя аргумента.

Рассмотрим один из наиболее удачных способов реализации безопасных обобщенных массивов, позволяющий задавать размер массива:

```
// Демонстрация стандартных шаблонных параметров
#include <iostream>
#include <cstdlik>
using namespace std;
// Здесь целочисленный аргумент size является стандартным
    template <class AType,  int size> class atype {
  AType a[size];  // Длина массива передается через параметр size
    public:
      atype()    {  register int i;    for(i=0;  i<size;  i++)  a[i]  = i;  }
     AType &operator[](int i);
};
// Проверка диапазона для объекта atype
   template <class AType,  int size>
            AType &atype<AType, size>::operator![](int i)
{
if(i<0  ||  i> size-1)  {
cout « "\nЗначение индекса  ";
cout « i <<" выходит за пределы допустимого диапазона.\n";
exit(1);
}
return a[i];
}
```

// см. продолжение

Шаблоны классов

Применение шаблонных классов. Стандартные типы

```
int main()                                          // продолжение
{   atype <int, 10> intob; // Целочисленный массив из 10 элементов
    atype <double,15> doubleob; //Массив действительных чисел, состоящий из 16-и элементов
 int i;    cout << "Целочисленный массив:  ";
   for(i=0;  i<10;  i++)  intob[i]  = i;
   for(i=0;  i<10;  i++)  cout << intob[i] << " ";      cout <<'\n';
cout << " массив действительных чисел: ";
 for(i=0;  i<15;  i++)    doubleob[i] = (double) i/3;
 for(i=0; i<15; i++) cout << doubleob[i] <<" ";     cout << '\n";
intob[12] = 100;  /* Генерирует сообщение об ошибке */   return 0;     }
```

Рассмотрим спецификацию шаблона atype:

Параметр size объявлен как целочисленный. Затем этот параметр используется в классе atype для объявления массива a. Хотя параметр size применяется в исходном коде как обычная переменная, его значение известно уже на этапе компиляции. Это позволяет задавать размер массива. Кроме того, параметр size используется при проверке диапазона индекса в операторной функции operator [] (). Обратите внимание на способ, которым создаются массивы целых и действительных чисел. Второй параметр задает размер каждого массива.

В качестве стандартных параметров можно применять лишь целые числа, указатели и ссылки. Другие типы, например float, не допускаются. Аргументы, которые передаются стандартным параметрам, должны содержать либо целочисленную константу, либо указатель, либо ссылку на глобальную функцию или объект. Таким образом, стандартные параметры можно рассматривать как константы, поскольку их значения нельзя изменять. Например, внутри функции operator [ ] () следующий оператор не допускается: size = 10; // Ошибка

Поскольку стандартные параметры считаются константами, с их помощью можно задавать размер массива, что довольно важно в практических приложениях.

Шаблоны классов

Применение аргументов по умолчанию в шаблонных классах

Шаблонный класс может иметь аргумент обобщенного типа, значение которого задано по умолчанию.

Например, такой:

```
template <class X=int> class myclass { //...
```

Если при конкретизации объекта типа myclass не будет указан ни один тип, используется тип int.

Стандартные аргументы также могут иметь значения по умолчанию. Они используются при конкретизации объекта, если не заданы явные значения аргументов.

Синтаксическая конструкция, применяемая для этих параметров, не отличается от объявления функций, имеющих аргументы по умолчанию.

Шаблоны классов

Применение аргументов по умолчанию в шаблонных классах

Рассмотрим еще один вариант безопасного массива, предусматривающий аргументы по умолчанию как для типа данных, так и для размера массива.

```
// Демонстрация шаблонных аргументов по
                умолчанию,
#include <iostream>
#include <cstdlib>
using namespace std;
// Параметр типа AType по умолчанию равен int, а
переменная size по умолчанию равна 10.
template <class AType=int, int size=10>
      class atype
 { AType a[size]; // Размер массива передается
            аргументом size
public:  atype() {  register int i;
for(i=0;  i<size;  i++)  a[i]  = i;  }
AType &operator[](int i);
};
// Проверка диапазона для объекта atype.
template <class AType, int size>
       Atype &atype<AType,
            size>::operator[](int i)
{  if(i<0   || i> size-1)
    { cout << "\nЗначение индекса ";
      cout << i << " выходит за пределы
допустимого диапазона.\n";
     exit(1);
    }
return a[i];
}                    // см. продолжение
```

```
// продолжение
int main()
 {// Целочисленный массив из 100 элементов
  atype<int,  100> intarray;
    atype <double> doublearray;    // Массив
действительных чисел, размер задан по умолчанию
 atype< > defarray;  // По умолчанию объявляется
целочисленный массив,  состоящий из 10 элементов
  int i;
  cout << "Целочисленный массив: ";
for(i=0;  i<100;  i++)  intarray[i]  = i;
for(i=0;  i<100;  i++)
   cout << intarray[i]  <<" ";    cout <<  '\n';
cout  <<"Массив действительных чисел:  ";
for(i=0;  i<10;  i++)
  doublearray[i]  =  (double) i/3;
for(i=0;  i<10;  i++)
   cout << doublearray[i]  <<" ";
 cout << '\n';
 cout <<"Массив по умолчанию:   ";
for(i=0;  i<10;  i++)
  defarray[i]  = i;
for(i=0;  i<10;  i++)
 cout << defarray[i]  << " ";   cout << '\n';
return 0;
}
```

Шаблоны классов

Применение аргументов по умолчанию в шаблонных классах

Исследуем строку:

```
template <class AType=int, int size=10> class atype {
```

Здесь тип AType по умолчанию является типом int, а переменная size равна 10.

Как демонстрирует на предыдущем слайде программа, объекты класса atype можно создать тремя способами:

- **Явно задавая тип и размер массива.**
- **Явно задавая тип массива, используя размер по умолчанию.**
- **Используя тип и размер массива, установленные по умолчанию.**

Применение аргументов по умолчанию (особенно типов) повышает универсальность шаблонных классов.

Если некий тип используется чаще других, его можно задать по умолчанию, предоставив пользователю самому конкретизировать другие типы.

Шаблоны классов

Ключевые слова typename и export

Ключевое слово typename используется в двух ситуациях. Во-первых, оно может заменять ключевое слово class в объявлении шаблона.

Например, шаблонную функцию swapargs () можно определить так:

```
template <typename X> void swapargs(X &a,  X &b)
{ X temp;     temp = a;    a = b;    b = temp;     }
```

Здесь ключевое слово typename задает обобщенный тип X. В этом контексте ключевые слова class и typename не различаются.

Во-вторых, ключевое слово typename информирует компилятор о том, что некое имя используется в объявлении шаблонного класса в качестве имени типа, а не объекта.

Рассмотрим пример:

```
typename X::Name  someObject;
```

Здесь имя X::Name используется в качестве имени типа.

Ключевое слово export может предшествовать объявлению template. Оно позволяет использовать шаблон из другого файла, повторяя лишь его объявление, а не все определение.

Шаблоны функций и классов

Домашние задания

1. **Индивидуальные задания на шаблоны функций или классов каждому студенту.**