Лабораторная работа № 15: РЕШЕНИЕ ЗАДАЧ С ПРИМЕНЕНИЕМ ПЕРЕГРУЗОК И ШАБЛОНОВ ФУНКЦИ

(4 часа)

Цель лабораторной работы – рассмотреть понятия, объявление и использование в программах перегруженных и шаблонированных функций в C++, освоить механизмы выполнения перегрузки функций, разработки шаблонов, приобрести умения по повышению эффективности программ за счет перегрузки функций.

План лабораторной работы

1. Познакомиться с технологией перегрузок функций.
2. Изучить возможности и преимущества перегрузки и шаблонирования функций.
3. Изучить правила сопоставления параметров при программировании перегрузок и сопоставлении фактических параметров.
4. На представленных примерах изучить способы перевода программ на более высокий уровень технологи программирования.

1. Необходимые теоретические сведения к выполнению лабораторной работы

1.1. Понятие перегрузки функции

Функции (а также шаблоны функций) называются *перегруженными* (overloaded), если они объявлены в одной области видимости (scope) и имеют одно и то же имя. Другими словами, под *перегрузкой функции* понимается, определение нескольких функций (две или больше) с одинаковым именем, но различными параметрами.

Наборы параметров перегруженных функций *могут отличаться* порядком следования, количеством, типом. Перегруженные функции не могут иметь разные типы возвращаемого значения при одинаковых параметрах.

```
void F();
char F(); // ошибка
void F(int x);
```

При этом следует понимать, что, например, группа описателей функции

```
void F(int x[4]);
void F(int x[]);
void F(int *x);
```

не являются перегруженными функциями, это одно и то же.

Таким образом перегрузка функций нужна для того, чтобы избежать дублирования имён функций, выполняющих сходные действия, но с различной программной логикой.

Для транслятора в таких перегруженных функциях общее только одно - имя. Очевидно, по смыслу такие функции сходны, но язык не способствует и не препятствует выделению перегруженных функций. Таким образом, *определение перегруженных функций служит, прежде всего, для удобства записи*. Но для функций с такими традиционными именами, как `sqrt, print` или `open`, нельзя этим удобством пренебрегать. Если само имя играет важную семантическую роль, то такое удобство становится существенным фактором.

В приведённом ниже примере при вызове функции с именем `FF` транслятор должен разобраться, какую именно функцию следует вызывать. Для этого сравниваются типы фактических параметров, указанные в вызове, с типами формальных параметров всех описаний функций с именем. В результате *вызывается та функция, у которой формальные параметры наилучшим образом сопоставились с параметрами вызова*, или выдается ошибка если такой функции не нашлось.

```
void print(double);
void print(long);

 void FF()
 {
print(1L);     // print(long)
print(1.0);    // print(double)
print(1);      // ошибка, неоднозначность: что вызывать
 // print(long(1)) или print(double(1)) ?
 }
```

Правила сопоставления параметров применяются в следующем порядке по убыванию их приоритета:

1. Точное сопоставление: сопоставление произошло без всяких преобразований типа или только с неизбежными преобразованиями
2. Сопоставление с использованием стандартных целочисленных преобразований (т.е. `char` в `int, short` в `int` и их беззнаковых двойников в `int`), а также преобразований `float` в `double`.
3. Сопоставление с использованием стандартных преобразований, (например, `int` в `double, unsigned` в `int`).
4. Сопоставление с использованием пользовательских преобразований.

Если найдены два сопоставления по самому приоритетному правилу, то вызов считается неоднозначным, а значит ошибочным. Эти правила сопоставления параметров работают с учетом правил преобразований числовых типов для C и C++. Пусть имеются такие описания функции `print`:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);
```

тогда результаты следующих вызовов `print()` будут такими:

```
void h(char c, int i, short s, float f)
{
       print(c);    // точное сопоставление: вызывается print(char)
       print(i);    // точное сопоставление: вызывается print(int)
       print(s);    // стандартное целочисленное преобразование:
                    // вызывается print(int)
       print(f);    // стандартное преобразование:
                    // вызывается print(double)
       print('a');  // точное сопоставление: вызывается print(char)
       print(49);   // точное сопоставление: вызывается print(int)
       print(0);    // точное сопоставление: вызывается print(int)
       print("a");  // точное сопоставление:
                    // вызывается print(const char*)
}
```

Таким, образом, цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для обеспечения перегрузки функций необходимо для каждого имени функции определить сколько различных функций с ним связано.

1.2 Шаблоны функции вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов.

Объявление шаблона функции начинается с заголовка, состоящего из ключевого слова `template`, за которым следует список параметров шаблона.

```
// Описание шаблона функции
template <class T>
T max (T a, T b)
{    return a > b ? a : b;
}
```

Ключевое слово `class` в описании шаблона означает тип, идентификатор в списке параметров шаблона `T` означает имя любого типа. В описании заголовка функции этот же идентификатор означает тип возвращаемого функцией значения и типы параметров функции.

```
...
// Использование шаблона функции
int m = min (1, 2);
...
//Экземпляр шаблона функции порождается (генерируется) компилятором
int min (int a, int b)
{  return a<b ? a : b;
}
```

В списке параметров шаблона слово `class` может также относится к обычному типу данных. Таким образом, список параметров шаблона просто означает, что `T` представляет собой тип, который будет задан позднее. Так как `T` является параметром, обозначающим тип, *шаблоны иногда называют параметризованными типами*.

Рассмотрим описание шаблона ещё одной функции.

```
template <class T>
T toPower (T base, int exponent)
{     T result = base;
      if (exponent==0) return (T)1;
      if (exponent<0) return (T)0;
      while (--exponent) result *= base;
      return result;
}
```

Переменная `result` имеет тип `T`, так что, когда передаваемое в программу значение есть 1 или 0, то оно сначала приводится к типу `T`, чтобы соответствовать объявлению шаблона функции. Типовой аргумент шаблона функции определяется согласно типам данных, используемых в вызове этой функции:

```
      int i = toPower (10, 3);
      long l = toPower (1000L, 4);
      double d = toPower (1e5, 5);
```

В первом примере `T` становится типом `int`, во втором - `long`. Наконец, в третьем примере `T` становится типом `double`.

Следующий пример приведет к ошибке компиляции, так как в нем принимающая переменная и возвращаемое значение имеют разные типы:

```
      int i = toPower (1000L, 4);
```

Требования к фактическим параметрам шаблона. Шаблон функции `toPower()` может быть использован почти для любого типа данных. Предостережение "почти" проистекает из характера операций, выполняемых над параметром `base` и переменной `result` в теле функции `toPower()`. Какой бы тип мы ни использовали в функции `toPower()`, эти операции для нее должны быть определены. В противном случае компилятор не будет знать, что ему делать. Вот список действий, выполняемых в функции `toPower()` с переменными `base` и `result`:

```
1. T result = base;
2. return (T)1;
3. return (T)0;
4. result *= base;
5. return result;
```

Все эти действия определены для встроенных типов. Однако если вы создадите функцию `toPower()` для какого-либо пользовательского типа или класса, то в этом случае такой тип *должен будет включать общедоступные принадлежащие функции,* которые обеспечивают следующие возможности:

- *действие 1* инициализирует объект типа `T` таким образом, что объект `T` должен содержать конструктор копирования,
- *действия 2* и *3* преобразуют значения типа `int` в объект типа `T`, поэтому класс `T` должен содержать конструктор с параметром типа `int`, поскольку именно таким способом в классах реализуется преобразование к классовым типам,
- *действие 4* использует операцию `*=` над типом `T`, поэтому класс должен содержать собственную *функцию-operator* `*=()`.
- *действие 5* предполагает, что в типе `T` предусмотрена возможность построения безопасной копии возвращаемого объекта (см. конструктор копирования).

Схема такого класса выглядит следующим образом:

```
class T
{     public:
      T (const T &base); // конструктор копирования
      T (int i); //приведение int к T
      operator *= (T base);
// ... прочие методы
}
```

Эти требования на фактические параметры и приведённый пример будут вами востребованы, когда придётся работать с объектными типами.

Отождествление типов аргументов. Так как компилятор генерирует экземпляры шаблонов функций согласно типам, заданным при их вызовах, то

критическим моментом является передача корректных типов, особенно если шаблон функции имеет два или более параметров. Хорошим примером является классическая функция max():

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

Функция max() будет работать правильно, если оба ее аргумента имеют один и тот же тип: int i = max (1, 2); double d = max (1.2, 3.4); Однако, если аргументы различных типов, то вызов max() приведет к ошибке, так как компилятор не сможет понять, что ему делать.

Один из возможных способов для разрешения неоднозначности состоит в *использовании приведения типов*, чтобы прояснить наши намерения: int i = max ((int)'a', 100);

Вторая возможность - это *явно объявить версию экземпляра* шаблона функции перед ее вызовом: int max (int, int); int j = max ('a', 100);

Третий способ решить проблему состоит в создании шаблона функций, который имеет параметры различных типов.

```
template <class T1, class T2>
T1 max (T1 a, T2 b)
{    return a > (T1)b ? a : (T1)b;
}
```

Использование этой новой версии max() не приведет к неоднозначности в случае использования двух различных типов. Например, если написать max ('a', 100); то компилятор будет использовать два заданных (посредством аргументов типа) и построит версию функции max() с заголовком char max (char, int); Далее компилятор перед выполнением сравнения приведет тип второго аргумента к типу первого аргумента. Такой способ допустим, однако использование двух типовых параметров в шаблоне функции, которая должна была бы работать только с одним типом, часто лишь затрудняет жизнь. Довольно тяжело помнить, что max ('a', 100) дает значение типа char, в то время как max (100, 'a') передает в вызывающую программу int.

1.2 Пример программ с применением перегрузки функций.

Составить программу, которая бы позволяла находить векторное произведение трёхмерных векторов, заданных в вещественном и комплексном пространствах. Это значит, что есть два типа векторов: у первого типа – компонентами являются действительные числа, у второго – комплексные.

Примечание. Конечно, можно было бы рассматривать пространство вещественных векторов как подмножество комплексного пространства. При таком подходе программирование задачи существенно бы упростилось, достаточно было бы ввести функцию трансформации вещественного числа в комплексное и все операции программировать бы для комплексного пространства.

Однако в задаче сказано, и в реальности такие ситуации имеют место быть, чтобы оба множества имели право на существование, как не зависимые, потому что каждое из них имеет свои методы, свойства, хотя у них имеется область пересечения. Поэтому мы будем разрабатывать программу для двух типов данных.

Решение. Разработаем контрольный пример.

Сначала запишем необходимые выражения в алгебраическом формате.

Есть вектор $\vec{x}=x_1.\overrightarrow{e_1}+x_2\,\overrightarrow{e_2}+x_3.\overrightarrow{e_3}$. Орты $(\overrightarrow{e_1},\overrightarrow{e_2},\overrightarrow{e_3})$ – единичные векторы пространства.

Тогда векторное произведение $\vec{z}$ двух векторов $\vec{x}$ и $\vec{y}$ будет получено как коэффициенты определителя:

$$\vec{z}=\vec{x}\times\vec{y}=\begin{pmatrix}\overrightarrow{e_1} & \overrightarrow{e_2} & \overrightarrow{e_3}\\ x_1 & x_2 & x_3\\ y_1 & y_2 & y_3\end{pmatrix}= \qquad (*)$$

$$=(x_2\cdot y_3-y_2\cdot x_3)\overrightarrow{e_1}+(x_3\cdot y_1-x_1\cdot y_3)\overrightarrow{e_2}+(x_1\cdot y_2-x_2\cdot y_1)\overrightarrow{e_2}$$

Составим числовые контрольные примеры. Допустим заданы действительные векторы

$\overrightarrow{w_1}=1.\overrightarrow{e_1}+1.\overrightarrow{e_2}+1.\overrightarrow{e_3}$; и $\overrightarrow{w_2}=1.\overrightarrow{e_1}+2.\overrightarrow{e_2}-1.\overrightarrow{e_3}$,

комплексные векторы

$\overrightarrow{v_1}=(1.+\mathrm{i})\overrightarrow{e_1}+(2.-i)\,\overrightarrow{e_2}+(0.+i)\overrightarrow{e_3}$; и $\overrightarrow{v_2}=(2.+0.\,i)\overrightarrow{e_1}+(1.-i)\overrightarrow{e_2}+(1.+i)\overrightarrow{e_3}$,

Следуя уравнению (*) получим вектор – результат векторного произведения для всех возможных сочетаний векторов:

а) $\overrightarrow{v_1}\times\overrightarrow{v_2}=(2.+0.i)\overrightarrow{e_1}+(0.+0.i)\,\overrightarrow{e_2}+(-2.+2.i)\overrightarrow{e_3}$; - комплексный вектор;

б) $\overrightarrow{w_1}\times\overrightarrow{w_2}=(-3.)\overrightarrow{e_1}+(2.)\,\overrightarrow{e_2}+(1.)\overrightarrow{e_3}$; -вещественный вектор;

в) $\overrightarrow{v_1}\times\overrightarrow{w_1}=(2.-2.i)\overrightarrow{e_1}+(-1.+0.i)\,\overrightarrow{e_2}+(-1.+2.i)\overrightarrow{e_3}$; - комплексный вектор;

г) $\overrightarrow{w_1}\times\vec{v}=(-2.+2.i)\overrightarrow{e_1}+(1.+0.i)\,\overrightarrow{e_2}+(1.-2.i)\overrightarrow{e_3}$; - комплексный вектор;

Представленный набор числовых примеров исчерпывает все возможные варианты сочетаний типов векторов, т.е. контрольный пример *обеспечивает требование проверки полноты программы*. А это означает, что при совпаде-

нии результатов программы с разработанным контрольным примером, программный комплекс может быть применён для массовых расчётов – любых возможных значений векторов из вещественного и комплексного пространств.

Разработка словесного алгоритма

1. Разработаем структуру – комплексное число – complex:
```
struct complex { double x, y;/* real and imaginary parts */ };
```
2. Разработаем ещё несколько структур для описания векторов. Фактически, каждый вектор – массив координат, и поскольку нам однозначно задано 3-х мерное пространство, мы можем не включать в структуру размер массива – он для всех векторов будет равен 3:
```
struct C_vector { complex x[3]; };//complex vector
struct R_vector { double x[3]; };// real vector
```
3. Чтобы программа, вычисляющая векторное произведение по формуле (*), была алгоритмически одинакова для различных комбинаций типов сомножителей – вещественных и комплексных векторов, разработаем несколько перегруженных функций сложения add и для умножения product вещественных и комплексных чисел:
```
complex add(complex, complex,double c=1.);//complex number
                                          // addition/subtruction
complex add(complex, double, double c=1.);
complex add( double, complex, double c=1.);
double add(double, double, double c=1.);
```
Третий параметр со значением по умолчанию введен для того, чтобы одной функций выполнять и сложение (c=1.), и вычитание (c=-1.).
```
complex product(complex, complex);//complex number product
complex product(complex, double);
complex product(double, complex);
double product(double, double);
```
Разработав операции по действию с комплексными и вещественными числами, мы можем разработать функции для выполнения векторного умножения по формуле (*) для всевозможных сочетаний сомножителей:
```
C_vector product(C_vector, C_vector);//complex vector product
C_vector product(C_vector, R_vector);
C_vector product(R_vector, C_vector);
R_vector product(R_vector, R_vector);
```
Для инициализации векторов и для вывода результата также необходимо разработать соответствующие функции:
```
void inputv(C_vector &);
void inputv(R_vector &);
void print(std::string, complex);//complex number output
```

```
  void print(std::string, C_vector);//complex vector output
  void print(std::string, R_vector);
```

Теперь на основе построенных прототипов можно записать главную программу – собрать её как конструктор из готовых модулей:

```
  int main()
  {
     C_vector v1, v2, v3;
     R_vector w1, w2, w3;
     complex a, b, c;
          a.x = 1; a.y = 2; b.x = -1; b.y = 3;
          cout << "\n Checking of simple operations\n";
          c = add(a, b); print("add: ",c);
          c = product(a, b); print("Prod: ", c);
          cout << "Initialisation of a complex vectors\n";
     inputv(v1); print("\n v1 = ",v1);
     inputv(v2); print("\n v2 = ",v2);
     cout << "\nResults of the comlex vector product\n";
     v3 = product(v1, v2); print("\n v3 = ", v3);
     cout << "Initialisation of a real vectors\n";
     inputv(w1); print("\n w1 = ", w1);
     inputv(w2); print("\n w2 = ", w2);
     cout << "\nResults of the vector product of complex and real
  vectors\n";
     v3 = product(v1, w1); print("\n v3 = ", v3);
     cout << "\nResults of the real vector product \n";
     w3 = product(w1, w2); print("\n w3 = ", w3);
     cout << "\nResults of the vector product of real and complex
  vectors\n";
     v3 = product(w1, v1); print("\n v3 = ", v3);
     system("pause");
     return 0;
  }
```

Как видим, применение структур и перегруженных функций *повысило качество и пользовательского интерфейса, и читаемость программы*.

Коды функций, прототипы которых мы уже обсудили, показаны ниже:

```
complex add(complex x, complex y, double c=1.0)
{
     complex z; z.x = x.x + c*y.x; z.y = x.y +c*y.y; return z;
}
complex add(complex x, double y, double c=1.0)
{
     complex z; z.x = x.x + c*y; z.y = x.y; return z;
}
double add(double x, double y, double c=1.0)
{
     return x + c*y;
}
```

```
complex add(double x, complex y, double c=1.0)
{
      complex z; z.x = x+c*y.x; z.y =y.y; return z;
}
complex product(complex x, complex y)
{
      complex z; z.x = x.x * y.x-x.y*y.y;
      z.y = x.y*y.x + x.x*y.y; return z; }
complex product(complex x, double y)
{
      complex z; z.x = x.x * y ;
      z.y = x.y*y; return z;
}
complex product(double x, complex y)
{
      complex z; z.x = x * y.x;
      z.y =  x*y.y; return z;
}
double product(double x, double y)
{ return x * y; }

void inputv(C_vector & v)
{
      cout << "Input complex vector:\n";
            for (int i = 0; i < 3; i++)
            {
                  cout << "x[" << i + 1 << "]: Re = "; cin >> v.x[i].x;
                  cout << "x[" << i + 1 << "]: Im = "; cin >> v.x[i].y;
            }
            return;
}
void inputv(R_vector & v)
{
      cout << "Input real vector:\n";
      for (int i = 0; i < 3; i++)
      {
            cout <<  "x[" << i + 1 << "]:  = "; cin >> v.x[i];

      }
      return;
}
void print(std::string capt, C_vector v)
{
      std::cout << capt;
      for (int i = 0; i < 3; i++)
            printf(" + (%3.1f%+4.1f*i)e%1d", v.x[i].x, v.x[i].y, i + 1);
      cout << endl << endl;
      return;
}
```

Обратите внимание на шаблон `(%3.1f%+4.1f*i)`: здесь между реальной и мнимой компонентами комплексного числа не поставлен знак +/-. Дело в том, что, если в шаблоне `f` перед количество выделяемых позиций поставить + (`1f%+4.1`), то не только отрицательные, но и положительные числа будут печататься со знаком плюс. Берите это на заметку!

```
void print(string capt, R_vector v)//вывод на консоль вещественного вектора
{
	std::cout << capt;
	for (int i = 0; i < 3; i++)
	printf(" (%3.1f)e%1d", v.x[i], i + 1);
	cout << endl << endl;
	return;
}
void print(string capt, complex v)//вывод на консоль комплексного вектора
{
	std::cout << capt;
		printf(" + (%3.1f%+4.1f*i)\n", v.x, v.y);
	return;
}

C_vector product(C_vector x, C_vector y)
{
	C_vector z;
	z.x[0] = add(product(x.x[1],y.x[2]), product(x.x[2], y.x[1]), -1.);
	z.x[1] = add(product(x.x[2],y.x[0]), product(x.x[0], y.x[2]), -1.);
	z.x[2] = add(product(x.x[0],y.x[1]), product(x.x[1], y.x[0]), -1.);

	return z;
}
C_vector product(C_vector x, R_vector y)
{
	C_vector z;
	z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
	z.x[1] = add(product(x.x[2], y.x[0]), product(x.x[0], y.x[2]), -1.);
	z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);
	return z;
}
C_vector product(R_vector x, C_vector y)
{
	C_vector z;
	z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
	z.x[1] = add(product(x.x[2], y.x[0]), product(x.x[0], y.x[2]), -1.);
	z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);
	return z;
}
R_vector product(R_vector x, R_vector y)
{
	R_vector z;
	z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[1], y.x[0]), -1.);
	z.x[1] = add(product(x.x[1], y.x[0]), product(x.x[1], y.x[0]), -1.);
	z.x[2] = add(product(x.x[0], y.x[1]), product(y.x[0], x.x[1]), -1.);
	return z;
}
```

Как видим, благодаря тому что нами разработаны методы `add` и `product` как для вещественных, так и для комплексных чисел, функция для вычисления векторного произведения имеет унифицированный. Функция умножения перегружена для 8 случаев сочетания сомножителей: оба сомножителя ком-

плексные числа, оба сомножителя вещественные числа, два варианта для смешанных произведений чисел – 4 функции, и таких же 4 функции для различных сочетаний типов векторов.

Перегрузка функции `add` сделана только в 4-х вариантах – для чисел, т.к. в задачу проекта не ставилось получить сложение векторов.

Результат работы программы с целью проверки с решением контрольного примера представлен по частям в соответствии с этапами решения задачи.

Блок проверки простейших операций

```
 Checking of simple operations
a:  + (1.0+2.0*i)
b:  + (-1.0+3.0*i)
add:  + (0.0+5.0*i)
Prod:  + (-7.0+1.0*i)
```

Блок ввода и вывода комплексных векторов:

```
Initialisation of complex vectors
Input complex vector:
x[1]: Re = 1
x[1]: Im = 1
x[2]: Re = 2
x[2]: Im = -1
x[3]: Re = 0
x[3]: Im = 1

 v1 =  + (1.0+1.0*i)e1 + (2.0-1.0*i)e2 + (0.0+1.0*i)e3

Input complex vector:
x[1]: Re = 2
x[1]: Im = 0
x[2]: Re = 1
x[2]: Im = -1
x[3]: Re = 1
x[3]: Im = 1

 v2 =  + (2.0+0.0*i)e1 + (1.0-1.0*i)e2 + (1.0+1.0*i)e3
```

Блок вычисления векторного произведения комплексных векторов (вариант а контрольного примера)

```
Results of the comlex vector product

 v3 = v1xv2 = + (2.0+0.0*i)e1 + (0.0+0.0*i)e2 + (-2.0+2.0*i)e3
```

Блок ввода и вывода вещественных векторов

```
Initialisation of real vectors
Input real vector:
x[1]:  = 1
x[2]:  = 1
x[3]:  = 1

 w1 =  + (1.0)e1 + (1.0)e2 + (1.0)e3

Input real vector:
x[1]:  = 1
x[2]:  = 2
x[3]:  = -1

 w2 =  + (1.0)e1 + (2.0)e2 + (-1.0)e3
```

Блок вычисления Векторных произведений со смешанным типом векторов (варианты б-г контрольного примера).

```
Results of the real vector product

 w3 = w1xw2 = + (-3.0)e1 + (2.0)e2 + (1.0)e3


Results of the vector product of complex and  real vector

 v3 = v1xw1 =  + (2.0-2.0*i)e1 + (-1.0+0.0*i)e2 + (-1.0+2.0*i)e3


Results of the vector product of real and complex vector

 v3 = w1xv1 =  + (-2.0+2.0*i)e1 + (1.0+0.0*i)e2 + (1.0-2.0*i)e3
```

Все представленные результаты полностью соответствуют контрольным примера. Контрольные примеры составлены с учётом полноты программы, поэтому мы можем применять наш комплекс программ для массовых расчётов.

1.3 Пример программ с применением шаблонов функций.

Обратим внимание и на то, что *отличия в кодах* перегруженных функций для вычисления произведения векторов *относятся лишь к типу используемых переменных*. И это даёт нам основание уменьшить количество перегруженных функций за счёт введения шаблонов. Хотя перегруженные функции произведения для чисел, комплексных и вещественных, отличаются по коду. Если мы составим шаблон для векторных произведений, а для числовых произведений оставим перегруженные функции, разберётся ли транслятор, когда что применять?

Оценим коды векторных произведений:

```
A)  C_vector product(C_vector x, C_vector y)
{
      C_vector z;
      z.x[0] = add(product(x.x[1],y.x[2]), product(x.x[2], y.x[1]), -1.);
      z.x[1] = add(product(x.x[2],y.x[0]), product(x.x[0], y.x[2]), -1.);
      z.x[2] = add(product(x.x[0],y.x[1]), product(x.x[1], y.x[0]), -1.);
      return z;
```

```
}
B)  R_vector product(R_vector x, R_vector y)
{
      R_vector z;
      z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
      z.x[1] = add(product(x.x[2], y.x[0]), product(x.x[0], y.x[2]), -1.);
      z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);
return z;
}

C)   C_vector product(C_vector x, R_vector y)
{
      C_vector z;
      z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
      z.x[1] = add(product(x.x[2], y.x[0]), product(x.x[0], y.x[2]), -1.);
      z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);
      return z;
}

D)   C_vector product(R_vector x, C_vector y)
{
      C_vector z;
      z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
      z.x[1] = add(product(x.x[2], y.x[0]), product(x.x[0], y.x[2]), -1.);
      z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);
      return z;
}
```

Видим, что, когда оба вектора имеют одинаковый тип (случай A-B), то этот же тип возвращают и функции. Следовательно, для этих двух функций можно использовать один шаблон:

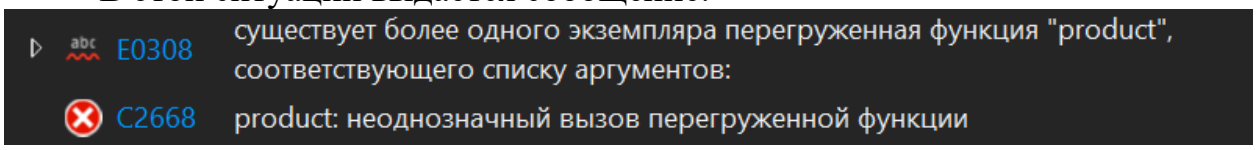
```
template <class T> T product(T x, T y)
{
      T z;
      z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
      z.x[1] = add(product(y.x[2], x.x[0]), product(x.x[0], y.x[2]), -1.);
      z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);

      return z;
}
```

Для случаев (C-D), казалось бы, можно сделать два шаблона:

```
template <class T1, class T2> T1 product(T1 x, T2 y);
template <class T1, class T2> T1 product(T2 x, T1 y);
```

Однако, в таком «обезличенном» типе формальных параметров транслятор не в состоянии различить, какой из шаблонов выбрать, например при вызове оператора `v3 = product(v1, w1);`

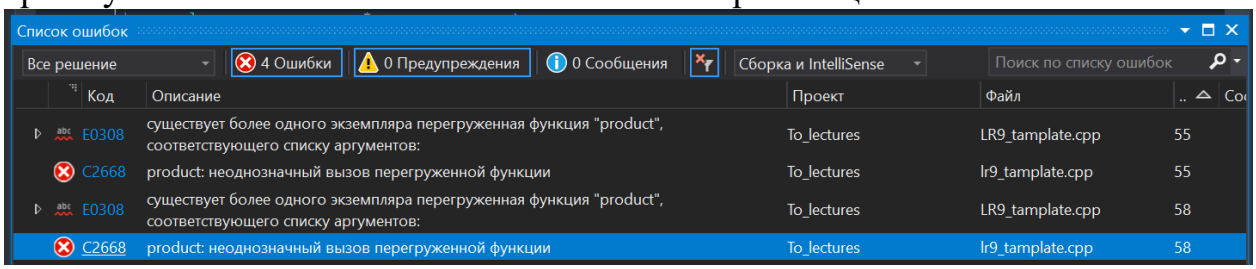
В этой ситуации выдаётся сообщение:

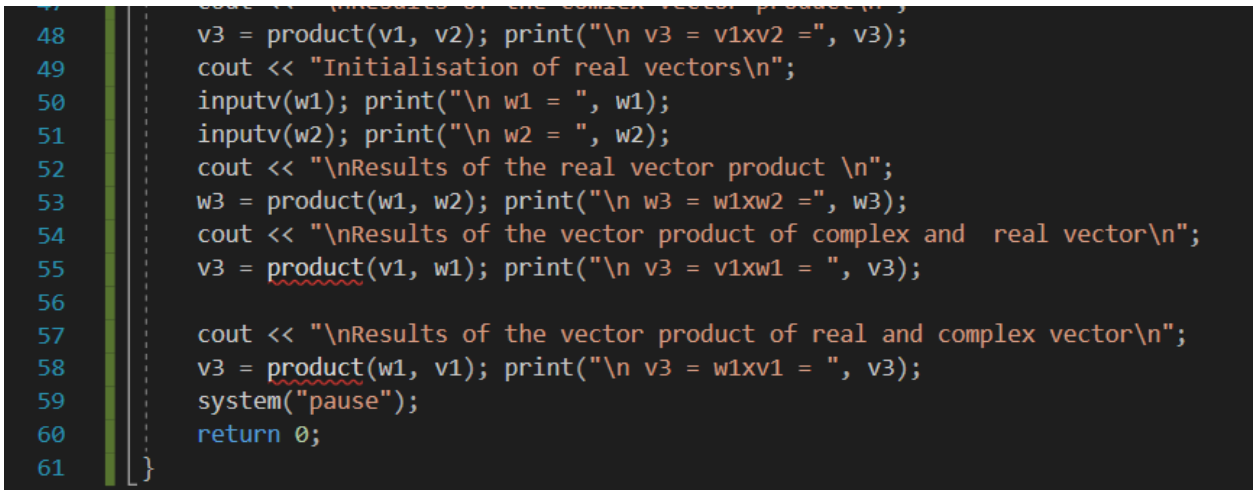


Тогда оставляем только первый вариант шаблона и убираем из проекта соответствующие функции:

```
template <class T1, class T2> T1 product(T1 x, T2 y)
{
	T1 z;
	z.x[0] = add(product(x.x[1], y.x[2]), product(x.x[2], y.x[1]), -1.);
	z.x[1] = add(product(y.x[2], x.x[0]), product(x.x[0], y.x[2]), -1.);
	z.x[2] = add(product(x.x[0], y.x[1]), product(x.x[1], y.x[0]), -1.);

	return z;
	}
```

Код главной программы `main` при этом мы не меняем и запускаем программу на выполнение. Увы! Опять ошибка трансляции:





Как видим, транслятор по-прежнему не может понять, каким шаблоном пользоваться при вычислении произведения смешанных типов векторов. При этом нет сообщений об ошибках перегруженных функций для произведения чисел. Это позволяет сделать вывод: транслятор *сначала просматривает перегруженные функции*, если среди них находит подходящую, то работает с ней, если не находит, обращается к шаблонам. В нашем случае шаблон не однозначен. Если мы для одного из вариантов векторного произведения добавим перегруженную функцию с явным описанием типов используемых параметров, есть надежда, что это поможет разрешить все конфликты с транслятором. На основании выполненного анализа сообщений транслятора добавляем в проект перегруженную функцию для произведения вещественного вектора с комплексным.

Теперь список всех прототипов функций имеет следующий вид:

```
struct complex { double x, y;/* real and imaginary parts */ };
struct C_vector { complex x[3]; };
struct R_vector { double x[3]; };
complex add(complex, complex, double c = 1.);//complex number
addition/subtruction
complex add(complex, double, double c = 1.);
complex add(double, complex, double c = 1.);
double add(double, double, double);
complex product(complex, complex);//complex number product
complex product(complex, double);
complex product(double, complex);
double product(double, double);

void inputv(C_vector &);
void inputv(R_vector &);
void print(std::string, C_vector);//complex vector output
void print(std::string, R_vector);
void print(std::string, complex);//complex number output

template <class T1, class T2> T1 product(T1 x, T2 y);
template <class T> T product(T x, T y);
C_vector product(R_vector x, C_vector y);
```

Вместе с двумя шаблонами в проект добавлена перегруженная функция для вычисления векторного произведения вещественного и комплексного векторов. Код главной программы не меняем. Запуск на выполнение такого варианта программного проекта оказывается успешным, и мы получаем те же результаты, что были получены при решении примера в пункте 1.2.

Таким образом, подведём итоги проделанных этапов усовершенствования проекта.

1. Разумное выделение обычных арифметических операций сложения и умножения в отдельные перегруженные функции позволило нам унифицировать более сложные перегруженные программы для вычисления произведения векторов.
2. Анализ кодов перегруженных функций для произведения векторов позволил нам ввести шаблоны для некоторых из них.
3. В результате введения шаблонов вместо четырёх перегруженных функций мы имеем два шаблона и одну перегруженную функцию.
4. В целом все манипуляции с программами касались только отдельных функций. Код главной программы оставался неизменным, это означает, что пользователю нашего проекта будет несложно работать с его программами, так как на уровне пользовательского интерфейса, разница в кодах и способах обращения к ним скрыта – инкапсулирована.

5. Проведённый анализ ошибок, выдаваемых транслятором, может служит вдумчивому студенту образцом самостоятельного разбора подобных ситуаций.

2. Методика выполнения самостоятельной работы

1. ***Внимательно изучить примеры***, приведенные в методической части лабораторной работы.
2. Изучить, как правильно стоить контрольные числовые примеры для рассмотренного типа задач.
3. Внимательно изучить, как в примерах 1.2 и 1.3 производится анализ кодов программ по мере усложнения технологии программы за счёт введения перегрузок и шаблонов.
4. Ознакомиться с условием задачи и примерами решений аналогичных задач из методической части лабораторной работы.
5. Составить контрольный пример.
6. Проверить полноту задачи: рассмотреть все возможные исходы решения в зависимости от исходных данных, предусмотреть случаи возможного зависания, зацикливания программы и запрограммировать корректную реакцию программы на эти ситуации.
7. Записать словесный алгоритм или составить блок-схему алгоритма.
8. Записать код программы на C^{++}.
9. Запустить программу, провести синтаксическую отладку.
10. Проверить работоспособность программы путём сравнения результатов с контрольным примером на все возможные случаи исходных данных.
11. Завершить работу составлением Отчёта, где будут описаны все этапы выполнения самостоятельного задания и приведены распечатки консольного вывода. Образец отчета приведен в Приложении.

3. Задания для самостоятельной работы

Номер варианта выбирается по формуле N%8+1, где N - порядковый номер студента в списке группы.

Задание 1. Соответственно варианту, разработать контрольный пример, построить необходимые перегруженные функции, проверить их работоспособность и полноту на основе контрольного примера. При необходимости разрешается использовать структурные переменные.

Вариант 1. Составьте программу для решения задачи. Определите значение: $x = \max\left(a, \max\left(\frac{a}{2}, cosb\right)\right) \cdot \max\left(2a - b, b\right)$. где max($u$, v) есть максимальное из чисел u, v. Разработайте перегруженные функции нахождения максимального из двух целых и вещественных чисел.

Вариант 2. Составьте программу для решения задачи. Найдите периметр треугольника, заданного координатами своих вершин, проверить, существует ли треугольник. Разработайте перегруженные функции нахождения расстояния между двумя точками, заданными своими координатами. Предусмотрите только случаи двумерного и трехмерного пространств.

Вариант 3. Составьте программу для решения задачи. Выясните, что больше: среднее арифметическое или среднее геометрическое трех положительных чисел. Разработайте перегруженные функции нахождения среднего арифметического и среднего геометрического трех целых и вещественных чисел.

Вариант 4. Составьте программу, которая в зависимости от входных данных переводит часы и минуты в минуты или минуты – в часы и минуты. Используйте перегруженные функции. Например, при вводе 134 мин будет выдано значение 2 час 14 мин, а при вводе 2 час 14 мин – значение 134 мин.

Вариант 5. Даны вещественные числа a и b, комплексное число c. Определить значение выражения: $\frac{a}{c}+\frac{a+c}{b}+\frac{1+a}{b+a}$, создав перегруженные функции для всех используемых операций.

Вариант 6. Разработать перегруженные функции для сложения десятичных и обыкновенных дробей.

Вариант 7. Разработать перегруженные функции для умножения десятичных и обыкновенных дробей.

Вариант 8. Разработать перегруженные функции для вычисления количества лет прошедших между двумя датами, заданных, либо как год, записанный десятичными цифрами, либо год, записанный римскими цифрами. Например: 2000 – 1812 или MM-MDCCCXII.

Задание 2. Написать программу языке C++ для обработки одномерных массивов, оформив каждый пункт задания в виде шаблона функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных не допускается. Привести примеры программ, использующих эти шаблоны для типов int, float, double.

Вариант 1

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

1) сумму отрицательных элементов массива;
2) произведение элементов массива, расположенных между максимальным и минимальным элементами.

Упорядочить элементы массива по возрастанию.

Вариант 2

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

1) сумму положительных элементов массива;
2) произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Вариант 3

В одномерном массиве, состоящем из n целых элементов, вычислить:

1) произведение элементов массива с четными номерами;
2) сумму элементов массива, расположенных между первым и последним нулевыми элементами.

Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные 0, считать положительными).

Вариант 4

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

1) сумму элементов массива с нечетными номерами;
2) сумму элементов массива, расположенных между первым и последним отрицательными элементами.

Сжать массив, удалив из него все элементы, модуль которых не превышает 1. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 5

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

1) максимальный элемент массива;

2) сумму элементов массива, расположенных до последнего положительного элемента.

Сжать массив, удалив из него все элементы, модуль которых находится в интервале [a,b]. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 6

В одномерном массиве, состоящем из *n* вещественных элементов, вычислить:

1) минимальный элемент массива;
2) сумму элементов массива, расположенных между первым и последним положительными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 7

В одномерном массиве, состоящем из *n* целых элементов, вычислить:

1) номер максимального элемента массива;
2) произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

Вариант 8

В одномерном массиве, состоящем из *n* вещественных элементов, вычислить:

1) номер минимального элемента массива;
2) сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом — все остальные.

* * *

4. Контрольные вопросы

1. С какой целью в программировании реализован полиморфизм функций?
2. Каким образом компилятор определяет, какую из перегруженных функций необходимо вызвать в программном коде?
3. Могут ли перегруженные функции возвращать результат одного типа? Ответ обоснуйте.
4. Для какого типа перегруженных функций допустимо применение шаблона?
5. Как работает транслятор, если у него есть и перегруженная и шаблонированная функция с одним именем.

5. Домашнее задание

Составить программу, которая бы позволяла находить сумму и разность 3-х мерных матриц, заданных в вещественном и комплексном пространствах. При разработке проекта использовать перегрузку и шаблоны функций.

https://ravesli.com/urok-173-shablony-funktsij/#toc-3