## Лекция 6 (продолжение)

## Динамические одномерные массивы

## 6.3 Типы памяти

Как нам известно, для работы с массивами их необходимо, как и любой другой программный объект, объявить, выделить память и проинициализировать. Для рассмотренных примеров в предыдущих лекциях, память под любой массив выделялась на этапе компиляции, и ее размер определяется заранее заданной константой (НЕ ПЕРЕМЕННОЙ!). Выделить память под массив, так же, как и изменить размер массива во время работы было нельзя. Напомним фрагмент ошибочной программы:

|  |
| --- |
| **int N1=5;**  **double ar[N1];//ошибка!Размерность должна быть задана константой** |

Очень часто мы не можем заранее сказать, каков размер требуемых массивов – размер может вычисляться в процессе выполнения программы. В таком случае возможны два варианта действий:

1. Зарезервировать память с запасом на этапе компиляции, но при этом всегда существует вероятность того, что запаса может не хватить, или наоборот, неиспользуемый резерв окажется очень большим.
2. Применить механизм, позволяющий выделить память во время выполнения программы в необходимом объеме.

С++ существует три типа распределения: статическое, автоматическое и динамическое**.**

***Статическая*** *память*, куда компилятор помещает на время выполнения программы *глобальные данные и данные, объявленные с ключевым словом* **static**. Вкратце смысл переменной, объявленной с ключевым словом **static** таков. Если переменная внутри блока объявляется как **static** (например, **static int k=0;**),то обращаться к такой переменной также можно только внутри блока, но существовать она будет до завершения всей программы, т.е. значение переменной будет сохраняться до следующего выполнения этого блока. Таким образом, такая переменная инициализируется нулем только один раз, при первом вызове функции. Если при работе функции такая переменная получала другое значение, то при повторном вызове той же самой функции используется уже ее измененное значение. То есть между вызовами функции статическая переменная сохраняет то значение, которое она имела при последнем вызове.

***ВАЖНОЕ ЗАМЕЧАНИЕ. Хотя мы называли уже рассмотренные в предыдущих лекциях массивы статическими (имея в виду, что они не могут изменять свой размер), такое название, как вы теперь понимаете, не совсем верно, так как с точки зрения выделения памяти они не являются static. Правильнее распределение памяти для таких массивов является – автоматическая динамическая память.***

***Автоматическая*** *память –* стек (структура данных, которая работает по принципу LIFO «последним пришёл — первым ушёл» или, другими словами, добавлять и удалять значения в стеке можно только с одной и той же стороны), которая до выполнения программного кода автоматически выделяется *локальным данным* и *параметрам функций* и автоматически освобождается при выходе из блоков. Напомним, блок – это любой фрагмент кода, заключенный в фигурные скобки, в частности составной оператор, и любая переменная, объявленная внутри блока или в теле функции, является локальной в этом блоке, т.е. «живет» только в блоке, после выхода из блока ее значение теряется, и обратиться к такой переменной вне блока нельзя. Например, если в теле блока объявлялась локальная переменная как

|  |
| --- |
| **int k=0;** |

то, как только работа блока завершалась, то сама переменная становилась недоступной, память и выделенная для этой переменной, очищалась. Соответственно, эта память становилась доступной к перераспределению для других переменных. При каждом новом вызове функции под такую переменную заново выделялась память, и она снова инициализировалась, как в данном примере, нулем.

***Динамическая*** *память* – иногда ее называют ***куча***, в которой требуемый блок памяти выделяется только по явному запросу программиста, используя операции **new** и освобождается программистом тоже явно – оператором **delete**.

С помощью оператора **new** можно динамически выделить память для любого объекта. Пример выделения памяти в куче для одиночного значения целого типа **int**:

|  |
| --- |
| **Int \*p=new int;** |

Оператор **new**, исходя из указанного типа, выделяет необходимое число (**sizeof(int)**) байт памяти и адрес начала этой памяти присваивает указателю **p**.

**Замечание**. На самом деле выделяется не столько байтов, сколько запросил программист, а больше, т.к. для каждого запрошенного программой блока памяти формируется дополнительная служебная информация.

Надо понимать, что использование динамической памяти – весьма дорогое средство, поэтому прибегать к нему следует лишь тогда, когда оно действительно необходимо. Поскольку:

* для небольших объектов дополнительная служебная информация может более, чем вдвое увеличить количество памяти для каждого выделяемого блока;
* в процессе выполнения программы на динамическое выделение памяти тратится дополнительное время (на формирование служебных структур данных);
* неочевидным следствием использования динамического выделения памяти является фрагментация кучи. То есть при выполнении программы может возникнуть такая ситуация, когда суммарно запрашиваемый объем памяти имеется, но эта свободная память разрезана на кусочки, которые компилятор склеивать не умеет.

Выделенную оператором **new** динамическую память необходимо явно освободить после того, как она уже не нужна следующим образом:

|  |
| --- |
| **delete p;//для освобождения блока памяти,адрес которого**  **//содержится в указателе р, С++ использует**  **//информацию, сформированную в служебном блоке**  **p=0; //после освобождения памяти в переменной р еще**  **//сохраняется значение адреса уже**  **//недействительного блока,**  **//поэтому безопаснее такой указатель обнулить** |

Если выделенный динамически блок памяти программист забыл освободить, то эта память будет освобождена операционной системой только после завершения приложения. Поэтому, если выделять память интенсивно и большими блоками (особенно под графические объекты), то даже виртуальная память может закончиться. Особенно внимательно надо относиться к динамическим объектам, которые создаются внутри функций. Если функция использует локальную переменную как указатель на память, выделенную с использованием **new**, то указатель будет уничтожен по окончании работы функции, но память останется выделенной и при этом недоступной из программы. Поэтому освобождение памяти после того, как она была использована и больше не требуется, является хорошим тоном, а зачастую и просто необходимо.

В предыдущей лекции и сейчас мы рассматриваем традиционные указатели языка C/C++. В Visual C++ понятие указателя расширено, и используются указатели двух типов:

* регулируемые указатели;
* нерегулируемые указатели.

Понятие «регулируемый указатель» появляется при создании приложений в режиме CLR. CLR является добавкой-расширением С++, введенной фирмой Microsoft, начиная с версии VC++ 2005. Когда мы создаем программы и работаем с программными объектами без режима CLR, мы должны сами заботиться об их размещении в памяти. Память для приложения вне режима CLR выделяется в так называемой «неуправляемой куче», в ней мы сами размещаем свои объекты и сами должны освобождать память, когда перестаем работать с объектом, иначе куча может переполниться и процесс выполнения приложения прервется. Рассмотренные нами указатели, обозначаемые символом \*, как раз являются указателями на участки памяти в такой «неуправляемой куче». Это традиционные указатели языка C/C++ на объекты в нерегулируемом объеме памяти, выделяемой приложению.

Приложение в режиме CLR, отличается от обычного тем, что его заготовка обеспечивает подключение к приложению специального системного пространства System, содержащего объекты, размещение в памяти, которых надо регулировать. Режим CLR работает уже с управляемой кучей памяти, в которой размещение объектов и её освобождение от них происходит под управлением среды. Для этого введено понятие «регулируемый указатель» – это тип указателя, который ссылается на объекты, расположенные в общей регулируемой куче памяти, предоставленной приложению в момент исполнения. Для таких указателей введено специальное обозначение: вместо символа \* применяется символ ^.

Мы уже видели при работе с элементами управления на форме, что регулируемые указатели автоматически создаются в обработчиках событий компонентов. В среде VC++ существует специальная утилита *gcnew*, которая формирует экземпляр объекта, выделяя экземпляру некоторую память, и возвращает ссылку на этот экземпляр. С регулируемыми указателями мы будем работать позже.

## 6.4 Динамические одномерные массивы

Чтобы создать динамический одномерный массив, необходимо указать тип и размерность массива. Например,

|  |
| --- |
| **int n = 20;**  **float \*p = new float [n];** |

В этом фрагменте создается переменная-указатель **p** на **float**, в динамической памяти отводится непрерывная область для размещения 20 элементов вещественного типа, и адрес начала этой области записывается в указатель **p**.

В принципе, создавать динамически массивы имеет смысл, когда:

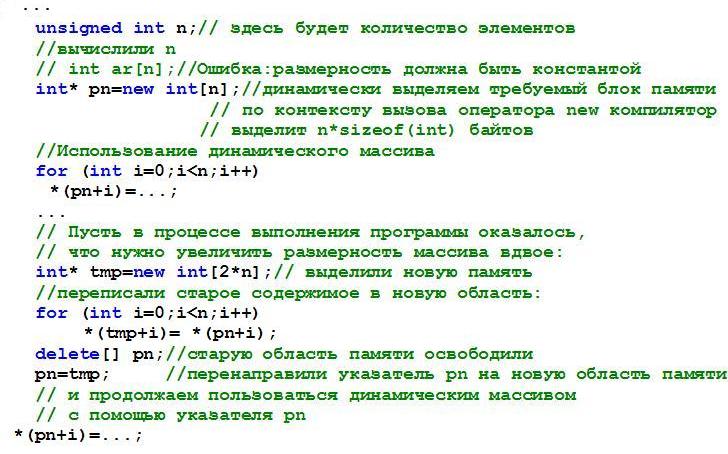
* размер массива вычисляется во время выполнения программы;
* размер массива может изменяться во время работы программы.

Для освобождения блока памяти, который выделялся под массив оператором **new**, используется также оператор **delete**, но с квадратными скобками, которые означают, что освобождается массив:

|  |
| --- |
| **delete[] p;** |

Как уже было показано ранее, для освобождения памяти, занятой скалярными объектами, квадратные скобки не нужны, но при освобождении памяти, занимаемой массивом, квадратные скобки обязательны. Их использование подразумевает, что мы освобождаем память, используемую для всех элементов массива, одновременно.

**Пример 6.8.** Пример создания и освобождения динамического одномерного массива (рисунок 10).

****

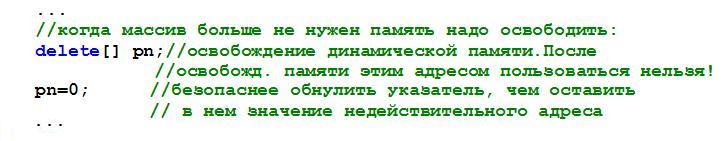
****

Рисунок 10 – Фрагмент программного кода с использованием одномерного динамического массива.

В соответствии с принципами структурного программирования, мы будем разрабатывать отдельные функции для ввода, вывода и формирования динамических массивов так же, как и раньше, для массивов, распределенных в стеке. Программный код функции вывода динамического массива абсолютно ничем не отличается от уже рассмотренной ранее функции для массивов, распределенных в стеке, так как в ней используется (выводится) уже созданный массив. Однако, для функций, в теле которых будет создаваться динамический массив, программный код будет другим. Для создания динамических массивов в теле функций требуется выделение памяти под них, поэтому в вызывающую функцию необходимо вернуть указатель на начало созданного массива, и программный код таких функций, естественно, отличается. Рассмотрим подробнее эти отличия.

**Пример 6.9.** Написать функцию формирования вещественного динамического массива из **n** элементов случайными числами из диапазона **[a;b]**.

Ранее, в лекции 4, мы рассматривали выполнение такого задания для массива, распределенного в стеке. (пример 4.1, рисунок 6). На рис. 11 представлен программный код функции ввода для динамического массива. Функция имеет три входных параметра по значению – число элементов массива **n**, и границы диапазона случайных чисел **a, b**. В теле функции прежде всего создается объект **mas** – указатель на **float**, оператором **new** выделяется динамическая память под **n** элементов типа **float**, и адрес начала этой памяти присваивается созданному указателю. Далее в регулярном цикле с помощью объекта класса **Random** и метода **NextDouble()** этого же класса генерируются случайные числа из отрезка [0,1], которые приводятся к нужному диапазону, и присваиваются элементам массива. По окончании цикла оператором **return** происходит возврат созданного указателя на массив в вызывающую функцию. Обратите внимание на способ доступа к элементам массива – запись **\*(mas+i)**, напомним, эквивалентна записи **mas[i]**, но в динамических массивах более распространена.

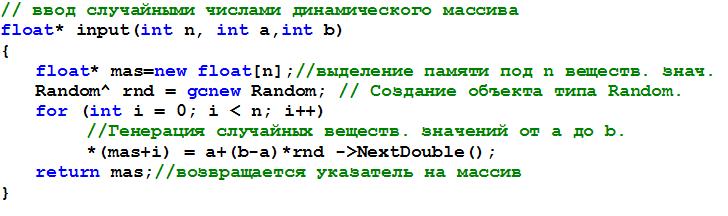


Рисунок 11 – Программный код функции создания вещественного динамического массива случайными числами.

Соответственно, вызов такой функции в событийной процедуре для создания динамического массива с именем **mas1** осуществляется после ввода необходимых значений **n1, a1** и **b1**, например, так:

|  |
| --- |
| **float \*mas1=input(n1,a1,b1);** |

И, конечно, в конце программы, когда все действия с массивом уже выполнены, и он больше не нужен, необходимо освободить занятую им память оператором **delete[] mas1;**

**Пример 6.10.** Разработать процедуру формирования динамического вещественного массива **y** из положительных элементов исходного вещественного массива **x**.

Эта задача аналогична задаче примера 4.5, рассмотренной в лекции 4. Однако, так как будет создаваться динамический массив, то рекомендуется вначале посчитать количество его элементов с помощью того же самого счетчика – переменной **n**, который использовался и в примере 4.5, и выделять динамическую память под уже посчитанное необходимое число элементов. Если же в исходном массиве нет положительных элементов и число элементов нового массива **n=0**, то новый массив пуст, и память под него выделять не надо. При создании динамического массива вычисленное в результате работы процедуры значение **n** (число элементов нового массива) возвращается через параметр по ссылке. Через возвращаемое значение оператором **return** процедура вернет указатель на созданный массив, или ноль, если массив не создавался. Как и в примере 4.5, входными параметрами процедуры являются: исходный массив **x** (в процедуру передается указатель на него) и число элементов исходного массива **m**. Программный код процедуры представлен на рисунке 12.

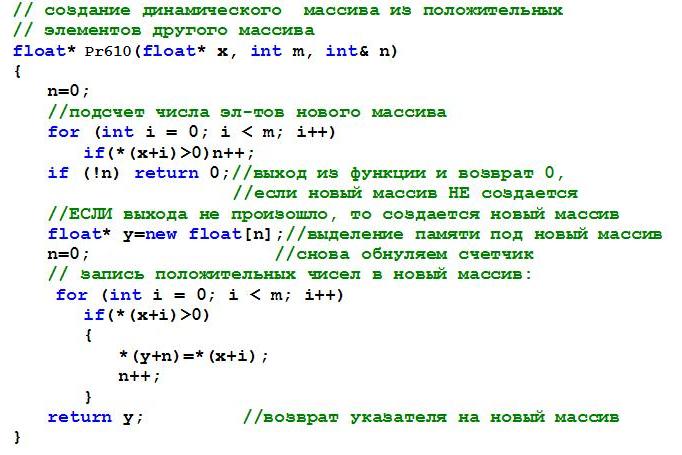


Рисунок 12 – Программный код функции создания нового динамического одномерного массива   
из положительных элементов другого массива

Следует заметить, что если предварительное вычисление числа элементов нового массива требует значительного дополнительного программного кода, то можно выделить память под новый массив по числу элементов исходного массива, т.е. для данного примера:

|  |
| --- |
| **float\* y=new float[m];** |

Однако, все-таки предварительный анализ того, стоит ли в принципе выделять память под новый массив (т.е. для данной задачи, есть ли в исходном массиве хотя бы один положительный элемент) всегда рекомендуется выполнить.

Приведем фрагмент кода событийной процедуры (рис. 13), в которой осуществляется вызов рассмотренных в примерах 6.9 и 6.10 процедур создания динамических одномерных массивов, а также вызывается функция вывода одномерного массива в список на форме, которая была уже рассмотрена (пример 4.4). Здесь вначале создается массив **mas1** из 10 случайных вещественных чисел отрезка [-5, 5] и выводится в **ListBox**, затем с помощью функции **Pr610** (рис. 12) создается новый динамический массив **mas2** из положительных элементов исходного массива. Далее проверяется значение логического выражения в скобках оператора условного перехода: **if (mas2).** Если это значение не равно нулю (т.е. истинно), что означает, что новый массив создан, то вызывается функция вывода нового массива и затем, в предположении, что этот массив больше не нужен, его память очищается. В противном случае (**mas2=0**, следовательно, новый массив не создавался) в окно **MessageBox** выводится соответствующее сообщение. В конце событийной процедуры в любом случае очищается память, занимаемая исходным массивом **mas1**.

|  |
| --- |
| **мас дин при6_10соб.jpg** |

Рисунок 13 – Фрагмент программного кода событийной процедуры для решения задачи примера 6.10

**6.5 Пример выполнения лабораторной работы 6**

1. **Задание**. Создать динамический одномерный массив **mas1** из чисел, записанных в многострочном текстовом поле **textbox** формы. Если в этом массиве есть хотя бы одно простое число, то сформировать новый массив **mas2** из тех элементов массива **mas1,** которыеявляются простыми числами.

2. **Формализация задачи**. Подобная задача была решена в предыдущей лабораторной работе №5, но теперь требуется формировать динамические массивы. Из условия задания следует, что его выполнение распадается на следующие этапы:

1. формирование массива **mas1** чтением чисел из текстового поля на форме;
2. анализ (проверка) есть ли в исходном массиве **mas1** хотя бы одно простое число, чтобы узнать, надо ли формировать новый массив;
3. формирование массива **mas2**;
4. вывод массива **mas2**.

Для программной реализации задания создадим следующие процедуры:

1. функцию **input()** формирования массива чтением чисел из многострочного текстового поля на форме;
2. функцию **output()** для вывода одномерного массива целых чисел в заданный список (элемент управления ***ListBox***) формы;
3. функцию **simple(),** которая проверяет, является ли ее формальный параметр простым числом;
4. функцию **analys()**, которая анализирует, есть ли в массиве хотя бы одно простое число;
5. функцию **task()** формирования нового массива из простых чисел исходного массива.

Вызов процедур формирования динамических массивов и вывода нового массива будем производить в событийной процедуре по нажатию кнопки **Решить задачу**.

Возможный вид формы проекта для организации интерфейса пользователя приведен на рисунке 14.

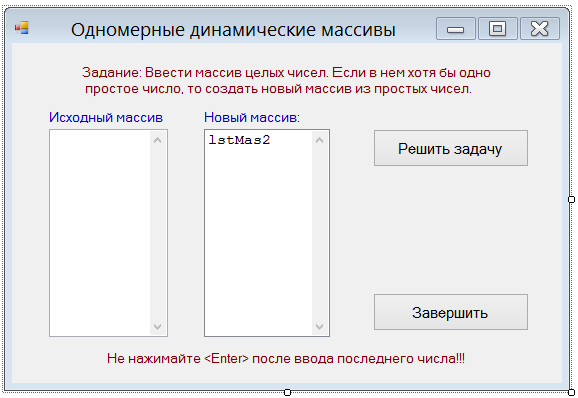


Рисунок 14 – Графический интерфейс проекта

На форме слева имеется одно многострочное текстовое поле **txtMas1** для ввода значений элементов исходного массива **mas1** и список **ListBox** с именем **lstMas2** – для вывода сформированного массива **mas2.** Числа для исходного массива будут во время работы приложения записываться в многострочное текстовое поле (элемент управления **textbox**). Чтобы получить многострочное текстовое поле, необходимо изменить значение его свойства **Multiline**, заданное по умолчанию, с **False** на **True**, и затем растянуть его по высоте. Также на форме имеются кнопка **Решить задачу (cmdTask)** для запуска программы на выполнение и кнопка **Завершить (cmdExit)** для останова ее выполнения.

В проект, помимо автоматически сгенерированных системой файлов, надо добавить 2 файла исходного кода: файл ***GetPut\_din.cpp*** с функциями для ввода и вывода (т.е. с функцией **input** и функцией **output**)и файл ***task\_din.cpp***  с функциями **simple, analys** и **task().**

Программный код файла ***GetPut\_din.cpp***. приведен на рисунке 15.

|  |
| --- |
|  |

Рисунок 15 –Программный код файла ***GetPut\_din.cpp***

Функция **input()** предназначена для ввода исходного массива из многострочного текстового поля. Она имеет 2 формальных параметра: указатель **TextBox^ TB** на текстовое поле, в которое записываются элементы массива, и **n –** параметр по ссылке, в который будет записываться количество элементов в массиве; и возвращаемое значение типа указатель на **int** – адрес первого элемента массива. Значение параметра **n** в теле функции определяется как число строк в многострочном текстбоксе с помощью свойства Length. С помощью этого же свойства осуществляется контроль непустого текстового поля. Если число строк в многострочном текстовом поле равно нулю (**n=TB->Lines->Length==0**), то возникает окно сообщений **MessageBox**, в которое выводится подсказка о необходимости заполнения текстового поля, и в вызывающую функцию (событийную процедуру) оператором **return** возвращается ноль, как признак того, что массив не создан. Если же число строк **n** в текстовом поле не равно нулю, т.е. в него записаны какие-то числа, то далее, в теле функции, происходит выделение памяти под **n** элементов целого типа **int**, и адрес начала этой памяти присваивается созданному указателю **mas**. Затем в цикле каждая строка из текстового поля конвертируется в целое число, которое и присваивается очередному элементу массива **mas**. По окончании цикла функция возвращает оператором **return** указатель на созданный массив, как возвращаемое значение в событийную процедуру.

Функция **output()** выводит массив в список на форме. Она имеет 3 формальных параметра: указатель на массив, число выводимых элементов и указатель на объект типа **ListBox**, в который требуется произвести вывод. Функция отличается от рассмотренной ранее в лабораторной работе 5 тем, что в случае пустого массива (если число его элементов равно нулю) осуществляет вывод в **ListBox** соответствующей строки.

На рисунке 16 приведен программный код файла ***task\_din.cpp***, содержащий необходимые функции для решения задачи. В файле записаны три функции.

|  |
| --- |
|  |

Рисунок 16 - Программный код файла ***task\_din.cpp***

Первая функция **simple()** проверяет, является ли натуральное число простым числом, и абсолютно аналогична уже рассмотренной в лабораторной работе 5.

Следующая функция **analys()** определяет, имеется ли в массиве хотя бы одно простое число. Формальными параметрами этой функции являются: указатель на массив и число элементов массива. В теле функции в регулярном цикле для каждого элемента массива вызывается функция **simple**, проверяющая, является ли этот элемент простым числом. Если функция **simple** вернула **true**, т.е. в массиве найдено простое число, то и сама функция возвращает **true** оператором **return**, соответственно, цикл прерывается. Если же ни одного простого числа не найдено, то цикл выполнится до конца и функция вернет **false**.

Последняя функция в файле ***task\_din.cpp*** - функция **task()** создает новый динамический массив из простых чисел исходного массива. Входными параметрами функции являются указатель на исходный массив и число элементов в массиве, выходным – счетчик чисел нового массива (как параметр по ссылке), возвращаемым значением – указатель на новый массив. В теле функции, после обнуления счетчика элементов нового массива, прежде всего вызывается функция **analys**, которая определяет, есть ли в исходном массиве хотя бы одно простое число. Если функция **analys** вернула **false**, то новый массив не создается, а функция **task** оператором **return** возвращает ноль и прекращает свою работу. Если же функция **analys** вернула **true**, то для нового массива выделяется динамическая память, адрес которой присваивается указателю на новый массив **newmas**. Заметим, что здесь мы не стали заранее считать количество простых чисел (как количество положительных чисел в примере 6.10 на рис. 12), а выделили память по числу элементов исходного массива, что тоже допускается, однако предварительно убедились в том, что простые числа имеются, и новый массив надо создавать. Далее в функции **task** реализуется типовой алгоритм создания нового массива из элементов другого массива точно так же, как и в лабораторной работе №5. В конце работыфункция **task** возвращает оператором **return** указатель **newmas** на созданный новый массив, а через параметр по ссылке – число элементов нового массива **k**.

После того, как эти файлы исходного кода созданы и успешно откомпилированы, создается заголовочный файл ***task\_din.h*** , в который из файлов исходного кода переносятся директивы **using** и записываются прототипы необходимых функций. Затем редактируется системный заголовочный файл ***stdafx.h***, добавлением в него директивы компилятора для подключения созданного заголовочного файла ***task\_din.h.*** На рисунке 16 приведен программный код созданного заголовочного файла ***task\_din.h*** и системного заголовочного файла ***stdafx.h***

|  |
| --- |
|  |

Рисунок 16 – Программные коды заголовочных файлов

И, наконец, создается программный код событийных процедур. На рисунке 17 приведен код этих процедур, вставляемых в файл ***Form1.h*** после директивы ***#pragma endregion***.

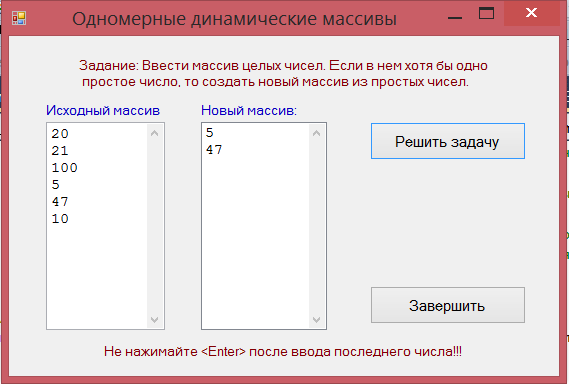
|  |
| --- |
|  |

Рисунок 17 – Программный код событийных процедур в файле ***Form1.h***

В событийной процедуре нажатия на кнопку «Решить задачу» (**cmdTask\_Click**) объявляются и инициализируются нулем две переменные целого типа, в которых будут храниться размерности массивов. Затем вызывается функция **input** создания динамического массива и заполнения его числами из многострочного текстового поля. Если текстовое поле **txtMas1** пусто, то функция **input** вернет ноль, т.е. исходный динамический массив не создан, и происходит выход из событийной процедуры. В противном случае вызывается функция **task** создания нового массива из простых чисел исходного массива, а затем функция **output** вывода полученного массива в предназначенный список на форме. Далее очищается память, занятая массивами, но только после проверки, существуют ли эти массивы (т.е. выделялась ли под них динамическая память).

**ВАЖНОЕ ЗАМЕЧАНИЕ. После ввода последнего числа для исходного массива в текстовое поле, надо оставить курсор на последней строке текстового поля и клавишу <Enter> НЕ нажимать, иначе пустая строка тоже будет учтена в свойстве Length, а числа-то нет, и возникнет ошибка!**

На рисунке 18 приведены результаты выполнения программы при различных исходных данных. Они подтверждают правильную работу используемых алгоритмов и соответствующих программ.



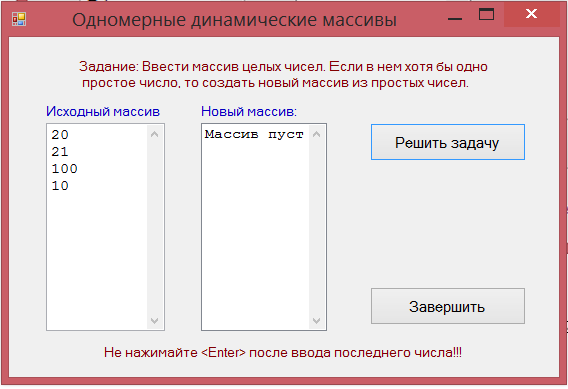


Рисунок 18 – Результаты выполнения проекта