В.Л. Тарасов

Лекции по программированию на C++

Лекция 8

Программы из нескольких файлов

Как правило, исходные тексты программ на C++ размещают в нескольких файлах. Это удобно, так как небольшой код легче понять и отладить, а работу можно распределить между несколькими программистами.

Средством работы с программами, состоящими из нескольких файлов, являются *проекты*, объединяющие файлы, из которых состоит программа. Проекты могут организовываться по-разному в различных средах разработки, но в проекты всегда включаются `cpp`-файлы с исходным кодом. Заголовочные `h`-файлы в проект могут не входить, так как они вставляются в состав других файлов директивой препроцессора `#include`.

8.1. Области видимости имен

Областью видимости имени называется часть программы, в которой имя доступно. По умолчанию областью видимости имени является часть исходного файла от точки объявления имени до конца файла. Одно имя нельзя использовать для нескольких элементов программы, например, переменных, в одной области видимости. (Несколько функций можно назвать одинаково, но они должны отличаться количеством или типами аргументов, см. §5.6).

Единицей компиляции в языке C++ является файл, содержащий различные объявления и определения, в том числе объявления и определения переменных и функций. В одном файле можно помещать определения нескольких функций. Нельзя части одной функции помещать в различных файлах.

В C++ нельзя определять функции внутри других функций, поэтому имя функции всегда глобально и может быть доступно в любой другой функции при соответствующем объявлении.

Для автоматических переменных, описанных в начале функции, областью видимости является эта функция. То же относится к

параметрам функций, которые, фактически, являются локальными переменными.

Переменные, определенные вне любой функции, являются глобальными. Имена функций и глобальных переменных видны от точки определения до конца файла без дополнительного объявления.

Когда программа состоит из нескольких файлов, необходимо обеспечить, чтобы функция или переменная, определенная в одном файле, была видима в других файлах, если они там используется. Это достигается с помощью *объявлений* переменных или функций перед их использованием. Переменные, определенные во внешнем файле, становятся видимыми в другом файле после их объявления в этом файле с ключевым словом `extern`. Объявление функции состоит из ее заголовка и точки с запятой. При объявлении функций слово `extern` можно не использовать.

Программа 8.1. Программа из нескольких файлов

Рассмотрим простой пример программы из трех файлов.

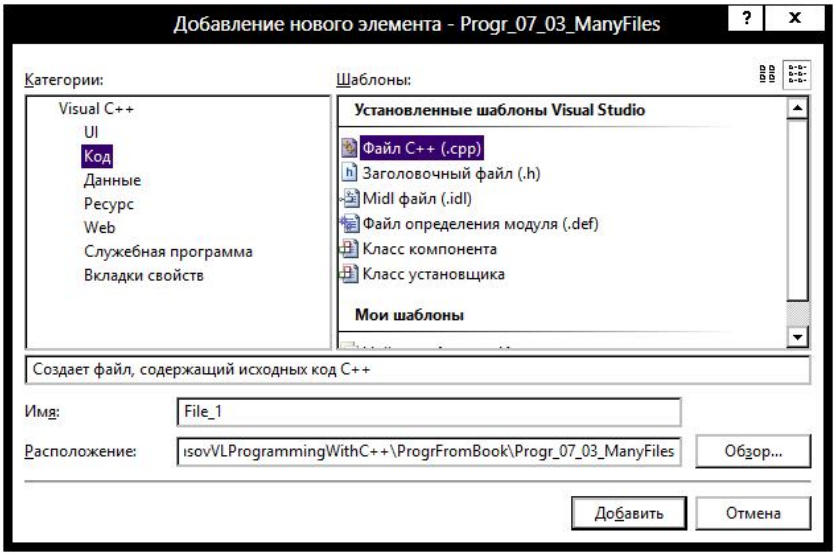

Рис. 8.1. Добавление файла в проект

В среде Visual Studio для добавления в проект нового файла с исходным кодом надо выполнить команду **`Проект, Добавить новый`**

элемент… . Появится диалоговое окно (рис.8.1), где надо выбрать категорию добавляемого элемента (**Файл C++(cpp)**), в поле **Имя** ввести имя, а в поле **Расположение** указать папку для добавляемого файла.

Сначала создадим файл File_1.cpp, в котором определим одну переменную и одну функцию:

```
// Файл File_1.cpp

int N;                  // Определение внешней (глобальной) переменной

void f1()               // Определение функции f1()
{ N = 1; }              // Использование внешней переменной
```

Затем добавим в проект файл File_2.cpp, в котором используются переменная N и функция f1() из файла File_1.cpp. Для этого N и f1() *объявляются* в начале файла File_2.cpp:

```
// Файл File_2.cpp

void f1();              // Объявление (прототип) функции f1()

extern int N;           // Объявление внешней переменной

int f2()                // Определение функции f2()
{
   f1();                // Обращение к функции f1()
   int k = N;           // Использование внешней переменной, k = 1
   return k;
}
```

В третьем файле File_Main.cpp объявляется функция f2() и вызывается из main():

```
//Файл File_Main.cpp

#include <iostream>
using namespace std;

int f2();               // Объявление функции f2()

// int N;               // Определение еще одной внешней переменной N

void main()
{
    cout << "f2() = " << f2() << endl;    // Использование f2()
    cin.get();                            // Ждем нажатия Enter
}
```

Программа выводит:

```
f2() = 1
```

Недостаток данной программы состоит в том, что не просто понять, почему функция f2() возвращает 1, так как в ее коде используется переменная N из другого файла.

При попытке определить еще одну внешнюю переменную с именем N в файле File_2.cpp (соответствующая строка закомментирована) возникает ошибка на этапе компоновки:

```
error LNK2005: "int N" (?N@@3HA) уже определен в File_1.obj
```

8.2. Статические имена

Зону действия имен можно ограничить одним файлом, объявив имена *статическими* с помощью ключевого слова static.

Внесем изменения в программу 8.1. Сделаем переменную N и функцию f1() доступными только в своем файле, объявив их статическими:

```
// Файл File_1.cpp

static int N;                 // Определение внешней переменной

static void f1()              // Определение функции f1()
{ N = 1; }                    // Использование внешней переменной
```

Компиляция всех файлов программы пройдет успешно, так как в них есть все необходимые объявления, но на этапе компоновки возникнет ошибка:

```
1>File_2.obj : error LNK2019: ссылка на неразрешенный внешний символ "void
__cdecl f1(void)" (?f1@@YAXXZ) в функции "int __cdecl f2(void)" (?f2@@YAHXZ)
```

Компоновщик не находит функцию f1() и переменную N, из-за ограниченности их зоны действия файлом File_1.cpp, так как f1() и N объявлены статическими.

8.3. Заголовочные файлы

При написании программы 8.1 нужно было заботиться об объявлениях внешних функций и переменных в файлах программы. Для облегчения этого, принято размещать необходимые объявления в отдельных файлах с расширением .h, которые называют *заголовочными* (от header – заголовок), так как их основное содержание составляют объявления или *заголовки* функций.

Страж включения

Заголовочные файлы вставляются в другие файлы директивой препроцессора #include, причем это надо делать однократно, так как при повторном включении возможны ошибки из-за дублирования определения какого-либо объекта. Чтобы не было повторного

включения заголовочных файлов, используется прием, называемый *стражем включения*. Страж включения реализуется директивами препроцессора в следующем виде:

```
// Файл header.h

#ifndef HEADERH                 // Страж
#define HEADERH                 // включения

// Содержимое header.h
// …

#endif
```

Условная директива препроцессора `#ifndef` проверяет, определен или нет макрос `HEADERH`. Если этот макрос *не определен*, то в программу включаются все строки, вплоть до строки `#endif`, при этом определяется макрос `HEADERH`. Если же директива `#ifndef` обнаружит, что макрос `HEADERH` *определен*, то препроцессор *не включает* все строки, расположенные между директивами `#ifndef HEADERH` и `#endif`, благодаря чему не будет повторного включения содержимого файла `header.h`.

Имя макроса, который фигурирует в страже включения (`HEADERH`), произвольно, но принято выбирать его похожим на имя заголовочного файла, чтобы данный макрос был уникальным в рамках проекта, так как все файлы проекта должны иметь разные имена.

Таким образом, если в некотором файле будут две директивы:

```
// Файл File1.cpp

#include "header.h"
#include "header.h"
```

то файл `header.h` будет вставлен только первой директивой, а вторая будет, фактически, проигнорирована благодаря стражу включения.

8.4. Понятие стека

Стеком называется набор элементов одного типа, организованный по принципу: *"последний пришел – первый вышел"* или LIFO (Last In - First Out). Английское слово stack переводится как копна сена, стопка бумаги. Для стопки бумаги действительно легче всего взять верхний лист, который попадает в стопку последним. Место, куда помещаются и из которого извлекаются элементы стека, называется *вершиной*.

Для работы со стеком определяют два основных действия или операции:

`push` – поместить новый элемент в стек и
`pop` – извлечь последний элемент, помещенный в стек.

Стек можно представлять себе как вертикальную стопку книг или тарелок, положенных друг на друга. Очередная тарелка кладется на верх стопки, извлекается же из стопки проще всего самая верхняя тарелка. Таким образом, LIFO – это естественная дисциплина работы со стопкой тарелок. В программировании стеки используются при решении задач, где данные обрабатываются в порядке, обратном порядку их появления.

Реализовать стек можно различными способами. В приводимой далее программе элементы стека размещаются в векторе.

Программа 8.2. Реализация стека в виде вектора

Создадим стек, элементами которого будут символы. Программа состоит из трех файлов: `Stack.h`, `Stack.cpp` и `TestStack.cpp`. Как показано на рис.8.1, файл `Stack.h` включен в раздел **`Заголовочные файлы`** проекта, а файлы `Stack.cpp` и `TestStack.cpp` – в раздел проекта **`Файлы исходного кода`**.

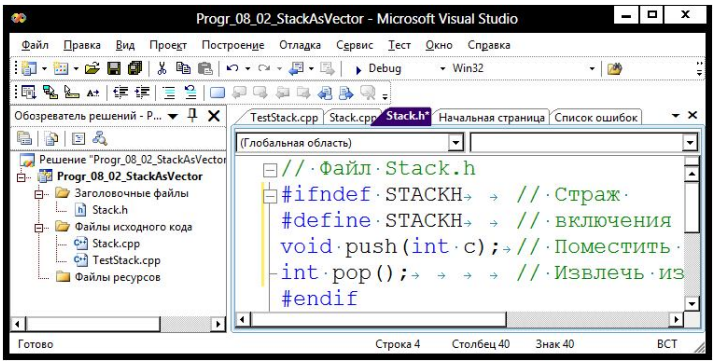

Рис. 8.1. Файлы программы

Файл `Stack.h` содержит объявления функций для работы со стеком.

```
// Файл Stack.h

#ifndef STACKH                  // Страж
#define STACKH                  // включения

void push(int c);               // Поместить с  в стек
int pop();                      // Извлечь элемент из вершины стека

#endif
```

Файл `Stack.cpp` содержит реализацию функций для работы со стеком.

```
// Файл Stack.cpp
```

```
#include <iostream>
#include <vector>

using namespace std;

#include "Stack.h"

static const int SZ = 100;      // Максимальный размер стека
static vector<char> s(SZ);      // Вектор для элементов стека
static int sp = 0;              // Позиция свободного элемента вектора

// push: поместить с в стек
void push(int c)
{
    if(sp < SZ)                 // Если в векторе есть место,
        s[sp++] = c;            // помещаем с в стек
    else
        cerr << "\nStack is full\n";
}

// pop: извлечь элемент из вершины стека
int pop()
{
    if(sp > 0)                  // Если стек не пуст,
        return s[--sp];         // берем значение с вершины стека
    else{
        cerr << "\nStack is empty\n";
        return EOF;
    }
}
```

В файле TestStack.cpp находится функция main(), где в стек сначала записываются 10 чисел от 0 до 9, затем из стека извлекаются 11 чисел.

```
// Файл TestStack.cpp

#include <iostream>
using namespace std;

#include "stack.h"

const int N = 10;

int main()
{
    cout << "Filling the stack:\n";
    for(int i = 0; i < N; ++i ){        // Заполнение стека

        cout << i << " ";               // Помещаем в стек N чисел
        push(i);
    }
    cout << "\nThe numbers from the stack:\n"; // Числа из  стека
    for(int i = 0; i <= N; ++i )        // Извлекаем из стека
        cout << pop() << " ";           // N + 1 число
    cin.get();                          // Ждем Enter
```

```
    return 0;
}
```

Результаты работы программы:

```
Filling the stack:
0 1 2 3 4 5 6 7 8 9
The numbers from the stack:
9 8 7 6 5 4 3 2 1 0
Stack is empty
-1
```

Видно, что элементы извлекаются из стека в порядке, обратном порядку их помещения в стек. При извлечении 11-го числа стек уже пуст, при этом функция pop() выводит соответствующее сообщение и возвращает EOF, числовое значение которого равно -1.

8.5. Пространства имен

В C++ имеется специальный инструмент управления именами – *пространство имен*. Пространства имен создаются с использованием ключевого слова namespace. Назначение пространств имен состоит в том, чтобы не допускать конфликтов имен при разработке больших программ.

Изменим программу 8.2, создав для стека пространство имен Stack.

Программа 8.3. Пространство имен для стека

Функции для работы со стеком объявим в файле Stack_ns.h, поместив их в пространство имен Stack:

```
// Файл Stack_ns.h

#ifndef STACKH                  // Страж
#define STACKH                  // включения

namespace Stack{                // Создание пространства имен Stack

   void push(int c);            // Поместить с  в стек
   int  pop();                  // Извлечь элемент из вершины стека
}                               // Конец объявления пространства имен

#endif
```

Теперь имена функций push() и pop() принадлежат пространству имен Stack, и к ним надо обращаться в виде: Stack::push() и Stack::pop(). Обратите внимание, что после фигурной скобки, закрывающей пространство имен, не ставится точка с запятой.

Определение функций для работы со стеком сделаем в файле Stack_ns.cpp:

```
// Файл Stack_ns.cpp

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

#include "Stack_ns.h"

static const int SZ = 100;        // Максимальный размер стека
static vector<char> s(SZ);        // Вектор для элементов стека
static int sp = 0;                // Позиция свободного элемента вектора

// push: поместить c  в стек

void Stack::push(int c)           // Используем полное имя Stack::push
{
    if(sp < SZ)                   // Если в массиве есть место,
        s[sp++] = c;              // помещаем c в стек
    else
        cerr << "\nStack is full\n";
}

// pop: извлечь элемент из вершины стека

int Stack::pop()                  // Используем полное имя Stack::pop
{
    if(sp > 0)                    // Если стек не пуст,
        return s[--sp];           // берем значение с вершины стека
    else{
        cerr << "\nStack is empty\n";
        return EOF;
    }
}
```

Используем стек для проверки правильности расстановки скобок во вводимом тексте. В функции `main()` посимвольно читается текст, все открывающие скобки, встреченные в тексте, помещаются в стек. Когда встречается закрывающая скобка, из вершины стека извлекается последняя открывающая скобка и сравнивается с прочитанной закрывающей. Если скобки соответствуют друг другу, продолжается анализ, если нет, то это ошибка. Ошибкой являются также случаи, когда закрывающая скобка идет раньше открывающей и когда открывающих скобок больше чем закрывающих.

Функцию `main()` поместим в файл `Brackets.cpp`:

```
// Файл Brackets.cpp

#include <iostream>
#include <cstdlib>
#include <locale>
using namespace std;

#include "stack_ns.h"
```

```
using namespace Stack;   // Делаем доступными имена
                         // из пространства имен Stack
int main()
{
    setlocale(LC_ALL, "Russian");
    int c, bro;
    while((c = cin.get()) != EOF)                  // Чтение символов
        if('(' == c || '[' == c || '{' == c)       // Открывающие скобки
            push(c);                               // помещаем в стек
        else if(')' == c || ']' == c || '}' == c){    // Встретилась
                                                   // закрывающая скобка
            bro = pop();              // Берем из стека открывающую скобку
            if(EOF == bro ||          // Если открывающей скобки нет,
                ('(' == bro && c != ')') ||        // или скобки не
                ('[' == bro && c != ']') ||        // соответствуют
                ('{' == bro && c != '}')){         // друг другу,
                    cerr << "Для " << char(c) << " нет открывающей\n";
                    exit(1);                       // завершение программы
            }   // if
        }       // else
        // Чтение текста закончено
        if(pop() != EOF)                           // Если стек не пуст
            cerr << "Открывающих больше чем закрывающих \n";
        else
            cout << "Скобки расставлены верно \n";
        return 0;
}
```

Директива `using`:

```
using namespace Stack;
```

делает доступными все имена из пространства имен `Stack`.

Можно делать доступными только отдельные имена из пространства имен с помощью *объявлений* `using`:

```
using Stack::push;   // Открываем имя push
using Stack::pop;    // Открываем имя pop
```

поместив их вместо директивы `using`.

Можно, не используя директиву `using` и объявления `using`, везде писать полные имена с квалификатором: `Stack::push()` и `Stack::pop()`.

Далее приведены результаты двух запусков программы.

Первый запуск:

```
)(^Z
Стек пуст
Для ) нет открывающей
```

Второй запуск:

```
{[()]}
^Z
```

```
Стек пуст
Скобки расставлены верно
```

Стандартное пространство имен

В C++ можно использовать библиотеки языка Си и соответствующие заголовочные файлы, например, `math.h`, `stdlib.h`, `stdio.h` (заголовочный файл библиотеки ввода/вывода языка Си) и другие. У собственных библиотек языка C++ заголовочные файлы не имеют расширения `.h`. Все имена из стандартной библиотеки C++ входят в пространство имен `std`. При использовании файла `iostream` это надо учитывать, например,

```
#include <iostream>
void main()
{
   std::cout << "Hello, ";
   std::cout << "World!" << std::endl;
   std::cin.get();
}
```

Чтобы не писать перед именами квалификатор `std`, можно использовать директиву `using`:

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

void main()
{
   cout << "Hello, World!" << endl;
   cin.get();
}
```

Есть реализации языка C++ поддерживающие старые варианты заголовочных файлов, в которых не использовалось пространство имен `std`. В этих средах наряду, например, с `iostream`, можно использовать старую версию `iostream.h`.

Некоторые заголовочные файлы языка Си были преобразованы в файлы C++, например, `math.h` превратился в `cmath`. Здесь префикс `c` указывает на происхождение данных файлов из языка Си. Такие файлы могут уже включать элементы, не относящиеся к Си, например, все имена могут входить в пространство имен `std`.