ЛЕКЦИЯ 21

Структура программы на С++

- Программа на языке С++ состоит из директив препроцессора, глобальных описаний и одной или нескольких функций. Одна из функций должна иметь имя main.
- Программа может размещаться в одном или нескольких текстовых файлах, содержащих исходный код программы.
- Таким образом, программа, состоящая из нескольких функций, может иметь вид:

глобальные описания
определение функции1
определение функции2
…
определение функцииn
головная функция

Такая, структура программы неприменима в следующих случаях: когда вызовы функций могут быть циклическими или когда работает механизм раздельной компиляции (программа записана в отдельных файлах, компилируемых раздельно). Для того, чтобы обойти это препятствие, в языке С++ введен механизм прототипов (заголоков) функций.

Использование прототипов дает возможность организовать следующую, более удобную структуру программы:

глобальные описания
прототип функции_1
прототип функции_2
прототип функции_n
головная функция
определение функции_1
определение функции_2
…
определение функции_n

Типичная структура простой консольной программы , разработанной в соответствии с парадигмой процедурного программирования:

- *Директивы препроцессора (обычно директивы **#include** для включения заголовочных файлов);*
- *Определения глобальных переменных;*
- *Объявления (прототипы) используемых функций;*
- *Определение главной функции **main**;*
- *Определения используемых функций.*

Структура простой консольной программы, которая разработана в соответствии с парадигмой объектно-ориентированного программирования, и размещается в одном файле исходного кода, может выглядеть так:

- Директивы препроцессора (обычно директивы #include для включения заголовочных файлов);
- Определения глобальных переменных;

- Объявления классов;
- Объявления (прототипы) используемых функций;
- Определение главной функции main;
- Определения используемых функций (в том числе, методов классов).

Время существования переменной определяет, как долго сохраняется ее значение. Значения локальных переменных теряются при выходе из блока, в котором они описаны, если только эти переменные не были описаны с классом памяти static. Время жизни глобальных переменных и переменных с классом памяти static – постоянное.

Время существования функции определяет, как долго сохраняется ее код. Время существования функций в C++ - все время выполнения программы.

Спецификаторы классов памяти

Это ключевые слова, управляющие временем существования переменных и возможностями ссылаться на идентификаторы вне области их определения. Спецификаторы классов памяти не изменяют область видимости идентификатора.

В C++ имеются следующие спецификаторы классов памяти:

- **extern** - указывает компилятору, что переменная или функция уже определена или будет определена вне области видимости своего объявления. Если этот спецификатор указывается при объявлении локальной переменной, то в этом случае он указывает, что эта переменная ссылается на глобальную переменную с тем же идентификатором. Если спецификатор **extern** используется с глобальной переменной, то эта переменная ссылается на глобальную переменную с тем же именем, определенную в другом файле исходного кода программы. Использование класса памяти extern – это один из примеров объявления, но не определения переменной. Объявление функции имеет этот спецификатор по умолчанию.
- **static** - управляет временем существования локальных переменных и возможностями доступа к глобальным переменным и функциям. Если локальная переменная определена со спецификатором **static**, то она существует в течение всего времени исполнения программы. Если такая перменная не инициализируется при своем объявлении, то ее значение устанавливается компилятором в нулевое (соответствующее типу переменной). Для функции или глобальной переменной спецификатор static ограничивает доступ к ней только файлом исходного кода, в котором эта переменная или функция определена, то есть к ней нельзя обратиться из другого исходного файла.
- **register** - указывает компилятору, что значение переменной желательно хранить в регистре процессора. Данный спецификатор может применяться только к локальным переменным и параметрам функций. Компилятор не обязан исполнять требование этого спецификатора и может хранить значение соответствующей переменной в оперативной памяти.
- **auto** - управляет временем существования локальной переменной и указывает, что локальная переменная, определенная внутри блока инструкций (в том числе - внутри блока, образующего тело функции), существует только во время исполнения этого блока. Класс памяти auto означает, что память для переменной выделяется в стеке. Локальные перменные имеют этот спецификатор по умолчанию.

Головная функция

Одна из функций, входящих в состав проекта, должна иметь имя **main**. Возможны следующие форматы ее заголовка:

```
void main(void);
void main();
int main(void);
int main();
void main(int argc, char *argv[]);
int main(int argc, char *argv[]);
```

- Аргумент **argc** определяет, сколько параметров, включая имя программы, записано в командной строке.
- Параметр **argv** – указатель на массив указателей нуль-терминированных строк, которые передаются программе при ее вызове: (**argv[0]** – имя exe-файла с полным путем к нему, **argv[1]** – первый параметр , **argv[2]** – второй параметр и т.д.).

Классический пример программы, которая выводит значения своих параметров вызова:

```
int main (int argc, char* argv[])
{
for (int i=1; i<argc; i++)
     cout << argv[i] << endl;
}
```

Функция **main** может иметь и третий параметр, который принято называть **argp**, и который служит для передачи в функцию **main** параметров операционной системы (среды) в которой выполняется С-программа. Т. е. заголовок функции **main** может иметь вид:

```
int main (int argc, char *argv[], char *argp[])
```

Если, например, командная строка С-программы имеет вид:

```
A:\>cprog working 'C program' 1
```

то аргументы **argc, argv, argp**
представляются в памяти примерно так, как показано в схеме на рис.1.
Это представление зависит от ОС.

```
#include <stdio.h>
#include <iostream>
int main ( int argc, char *argv[], char *argp[])
{ setlocale(LC_ALL,".1251");
int i=0;
     printf ("\n Имя программы %s", argv[0]);
     for (i=1; i<argc; i++)
 printf ("\n аргумент %d равен %s",i,argv[i]);
 printf("\n Параметры операционной системы:");
 while (*argp)
 {  printf ("\n %s",*argp);
    argp++;
 }
 return (0);
}
```

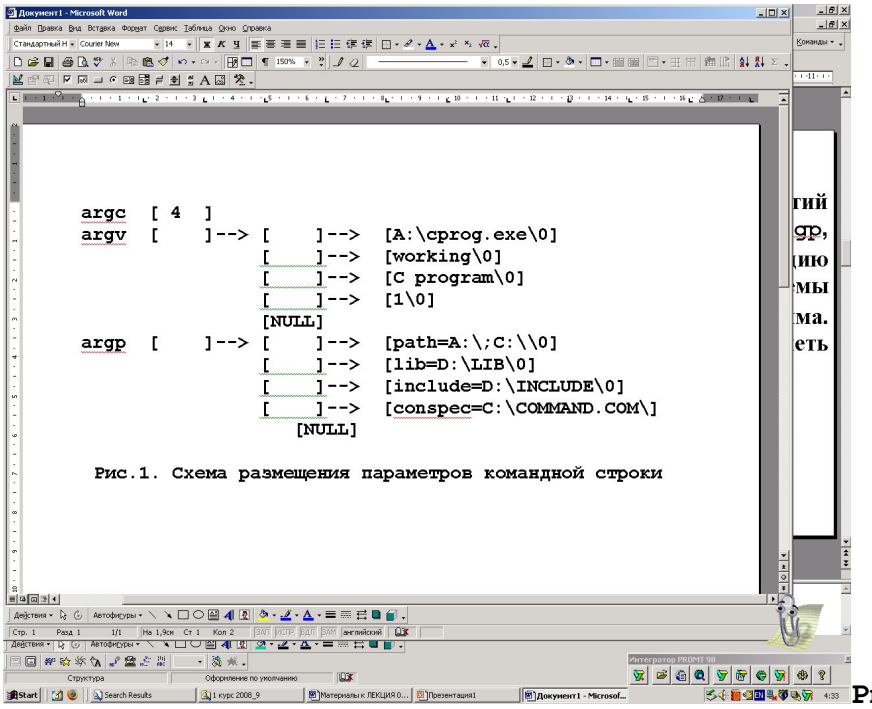

Рис.1. Схема размещения параметров командной строки

Рис.1.

```
Имя программы e:\UP_2007\parametry_main\Debug\parametry_main.exe
  Параметры операционной системы:
ALLUSERSPROFILE=H:\Documents and Settings\All Users
APPDATA=H:\Documents and Settings\Valuha\Application Data
CommonProgramFiles=H:\Program Files\Common Files
COMPUTERNAME=VALENTINA
ComSpec=H:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=H:
HOMEPATH=\Documents and Settings\Valuha
LOGONSERVER=\\VALENTINA
NUMBER_OF_PROCESSORS=2
```

Пример передачи параметров в функцию main

Задача: в программу в качестве параметра передаётся имя файла для обработки. Если параметр не передан, программа должна запросить им я файла в диалоге.

```
int main(int argc, char *argv[]) {
  char FileName[40];
  setlocale(LC_ALL, ".1251");
  if (argc>2) {
    cout << "Слишком много параметров\n";
    return 1;
  }
  if (argc==1) {
    cout << "Введите имя обрабатываемого файла: \n";
    cin.getline(FileName, 40);
  }
  else
    strcpy(FileName, argv[1]);
...
```

Код возврата

Целочисленное значение, возвращаемое функцией **main**, интерпретируется как код завершения программы, который может быть обработан в операционной системе (например, с помощью конструкции **errorlevel** из пакетных файлов).

В C++ на функцию **main** накладываются следующие ограничения:

- Нельзя получить адрес функции **main**;
- Нельзя объявить функцию **main** как статическую;
- Нельзя объявить функцию **main** как встроенную;
- Нельзя перегрузить функцию **main**;
- Нельзя вызывать функцию **main** внутри программы.

Рассмотрим подробнее последнее ограничение. При попытке вызова при компиляции будет предупреждение:

warning C4717: 'main' : recursive on all control paths, function will cause runtime stack overflow

Стоит обратить на него внимание!!!

Пространства имен C++

Пространства имен (называемые также поименованными областями) служат для логического группирования объявлений и разграничения доступа к ним. Чем больше программа, тем более актуально использование пространств имен. Простейшим примером является случай, когда несколько человек работают над одним и тем же проектом, и необходимо совместить код, написанный одним человеком, с кодом, написанным другим. При использовании единственной глобальной области видимости сделать это сложно из-за возможного совпадения и конфликта имен. Использование пространств имен является одним из возможных решений этой проблемы.

Объявление пространства имен

Пространство имен объявляется с помощью оператора

namespace *[имя_пространства]*{ *объявления* };

В операторе **namespace** могут присутствовать не только объявления, но и определения программных объектов (тела функций, инициализаторы переменных и т.д.).

Поименованная область может объявляться неоднократно, причем последующие объявления рассматриваются как дополнения к предыдущим. Более того, пространство имен является понятием, уникальным для всего проекта, а не одного программного файла, так что дополнение пространства имен может выполняться и за рамками одного файла.

```
namespace NS1 {
 int i=1;
 int k=0;
 void f1(int);
 void f2(double);
}
…
namespace NS2 {
 int i, j, k;
}
```

```
…
namespace NS1 {
 int i=2;           // Ошибка – повторное определение
 int k;             // Ошибка – повторное объявление
 void f1(void);     // Перегрузка
 void f2(double); //А такое повторное объявление допустимо!
}
```

Анонимные пространства имен

Если имя пространства имен не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, отдельного для каждого программного модуля. Такое пространство будем считать безымянным. В безымянное пространство имен входят также все глобальные объявления.

Стандартное пространство имен

Объекты стандартной библиотеки классов (потоки ввода-вывода также входят в состав этой библиотеки) могут быть расположены в пространстве имен std. При этом программист в отдельных случаях может выбрать режим описания: в заголовочном файле <iostream.h> потоки ввода вывода описаны как объекты из безымянного пространства, а в заголовочном файле <iostream> они же включены в пространство std. С другой стороны, класс string описан только в заголовочном файле <string> и находится в пространстве имен std.

К настоящему времени стандартным стал способ записи заголовочных файлов без ”.h” (и, соответственно, указание пространства имен std либо явно, либо с помощью оператора using). Более того, заголовочные файлы стандартной библиотеки с суффиксом ”.h”считаются устаревшими и существуют только для обратной совместимости.

Разрешение области действия идентификатора

Все программные объекты, описанные внутри некоторого пространства имен, становятся видимыми вне оператора **namespace** с помощью оператора :: , трактуемого как оператор разрешения области действия. При этом можно использовать одноименные переменные из различных пространств и собственные переменные:

```
void main(void)
{
  int i = 2 + NS1::i;
}
```

Объявление и директива using

Если имя часто используется вне своего пространства, его можно объявить доступным с помощью оператора

using *имя_пространства* :: *имя_в_пространстве*;

после чего такое имя можно использовать без явного указания имени пространства, например

```
void main(void)
{
  using NS2::k;
  int i = 2 + k;
}
```

Наконец, можно сделать доступными все имена из какого-либо пространства, записав оператор

using namespace *имя_пространства*;

Имена, объявленные где-нибудь явно или с помощью оператора **using**, имеют приоритет перед именами, доступными по оператору **using namespace**.

Псевдонимы пространства имен

Можно задать псевдоним (синоним) пространства имен с помощью оператора

using *имя_синонима* = *имя_пространства*;

Препроцессор С++

компьютерная программа, принимающая данные на входе, обрабатывающая их, и в результате выдающая данные, предназначенные для входа другой программы. Наиболее частый случай использования препроцессора — обработка исходного кода программы перед передачей его на следующий шаг компиляции. Результат и вид обработки зависят от вида препроцессора.

Препроцессор С++

- Лексические препроцессоры (C, C++) анализируют исходный код программы, выполняя замену макросов, текстовые вставки из других файлов, а также условную компиляцию или подключение файлов в соответствии с директивами пользователя.
- Синтаксические препроцессоры (Лисп) обычно используются для уточнения синтаксиса языка, расширения языка путем добавления новых примитивов, или встраиванием предметно-ориентированного языка программирования в основной язык.

Препроцессор С++

Препроцессор в языке C++ унаследован от C, некоторые возможности препроцессора C++ являются избыточными и реализованы средствами языка программирования. Фактически одной из целей развития языка C++ является исключение препроцессора.

Директива препроцессора

строка в исходном коде программы, задающая для препроцессора правило обработки исходного кода (фрагмента исходного кода).

Обработка исходных текстов программ компилятором в C++, как и в Ассемблере, выполняется в два этапа: препроцессорная обработка и собственно компиляция, т.е. построение объектного кода.

На первом этапе происходит преобразование исходного текста, а на втором компилируется уже преобразованный текст.

Препроцессор обрабатывает собственные директивы. Эти директивы задаются, как правило, в отдельной строке и начинаются с символа ‘#’

Директивы препроцессора

- директивы компилятора #pragma, указывающие компилятору, как именно необходимо строить объектный код;

- директива включения файла #include, с помощью которой можно включить в текст программы текст из другого файла;

- директивы условной компиляции #if, #else, #elif, #endif, #ifdef, #ifndef, defined;

- директива определения лексем #define;
- директива отмены определения #undef
и некоторые другие.

1. Директива **#include** включает в текст программы содержимое указанного файла. Она может быть записана в одном из двух форматов:

#include <*имя_файла*>
#include "*имя_файла*"

Кавычки или угловые скобки указывают компилятору, где искать вставляемый файл с указанным именем.

Если имя заключено в угловые скобки, поиск осуществляется в каталоге, указанном компилятору как стандартный для хранения включаемых файлов (обычно - каталог INCLUDE).

Если имя заключено в кавычки, то вначале поиск осуществляется в рабочем каталоге. Если файл не найден в рабочем каталоге, его поиск будет продолжен так, будто имя файла было заключено в угловые скобки. Второй формат дает возможность записать произвольное имя файла в терминах операционной системы.

Программы на языке C++ используют директиву #include не только для включения файлов, но и заголовков (headers).

В частности, в C++ имеется набор стандартных заголовков, необходимых для работы библиотечных функций. Эти заголовочные файлы несут, в частности, информацию об именах функций, количестве и типах принимаемых аргументов и типе возвращаемого значения.

2. Директива **#define** позволяет определить новые лексемы. Ее формат:

#define *имя_лексемы [(параметры)] текст_лексемы*

Приведем пример такого длинного описания, взятый из стандартного файла stdarg.h:

```
#define va_arg(list, mode) \
  ( *(((list).offset += ((int)sizeof(mode) + 7) & -8) , \
        (mode *)((list).a0 + (list).offset - \
((__builtin_isfloat(mode) &&(list).offset <= (6 * 8)) ? \
           (6 * 8) + 8 : ((int)sizeof(mode) + 7) & -8) \
            ) \
        ) \
    )
```

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

Эти директивы изменят в тексте программы каждое слово WIDTH на число 80, а каждое слово LENGTH на выражение (80+10) вместе с окружающими его скобками.

Описание

```
#define MAX_LENGTH 50
```

может быть использовано вместо

```
const int MAX_LENGTH = 50;
```

Особо следует сказать об определениях лексем с параметрами (макросах). Такие

конструкции позволяют выполнить замещение лексем по-разному, в зависимости от фактических параметров. Иногда использование макросов оказывается более полезным, чем задание множества перегруженных функций или шаблона функции. Так, функцию нахождения максимального из двух чисел удобно описать макросом:

```
#define max(x,y)((x)>(y)?(x):(y))
```

Скобки, содержащиеся в макроопределении, позволяют избежать недоразумений, связанных с порядком вычисления операций, если фактические аргументы являются выражениями. Например, при отсутствии скобок в примере

```
#define sqr(x) x*x
```

приводит к тому, что при вызове макроса

```
sqr(a+2)
```

получается неверное (хотя и синтаксически правильное) выражение

```
a+2*a+2
```

При наличии скобок фрагмент

```
t=MAX(i&j,s[i]||j);
```

будет заменен на фрагмент `t=((i&j)>(s[i]||j)?(i&j):(s[i]||j);`
а при отсутствии скобок - на фрагмент `t=(i&j>s[i]||j)?i&j:s[i]||j;`
в котором условное выражение вычисляется в совершенно другом порядке.

Директива

#**undef** *имя_лексемы*

отменяет определение лексемы, заданное директивой #define.

Не является ошибкой использование директивы #undef для идентификатора, который не был определен директивой #define.

```
#undef WIDTH
#undef MAX
```

Наконец, директива

defined(*имя_лексемы*)

дает возможность выяснить, определена ли указанная лексема.

3. Директивы условной компиляции дают возможность включить в исходный текст те или иные строки, в зависимости от значения выражения (которое должно быть выражением времени компиляции), например:

```
#define DEBUG_MODE 1
//эта строка будет исключена по
//завершении отладки
…
#if defined(DEBUD_MODE) && DEBUD_MODE==1
//вывод отладочной информации
#endif
```

4. Директива #define часто используется при организации т.н. стража включения, препятствующего повторному объявлению и определению идентификаторов при подключении нескольких заголовочных файлов. Поскольку язык C/C++ допускает вложенное использование директивы #include, может возникнуть ситуация, когда программист, сам того не желая, включит в свои исходные тексты одно и то же описание несколько раз. Рассмотрим соответствующий пример. Пусть у нас есть три заголовочных файла:

a.h	b.h	c.h

`void my_funcA()`
`{`
`…`
`}`
	`#include`
	`"a.h"`
	`…`
		`#include`
		`"a.h"`
		`…`

Теперь, записав в начале нашей программы директивы

```
#include "b.h"
#include "c.h"
```

получим неожиданное сообщение компилятора о повторном определении функции **my_funcA().** Чтобы избежать этого, можно организовать проверку повторного определения (страж включения) в файле **a.h.**

Для этого достаточно записать в этом файле конструкцию

```
#ifndef _my_funcA_defined_
#define _my_funcA_defined_
void my_funcA();
#endif /* _my_funcA_defined_ */
```

Имя идентификатора, используемого в страже включения, должно быть подобрано так, чтобы оно случайно не совпало ни с одним из других идентификаторов программы!

В последних вариантах защиты от повторного включения рекомендуют использовать следующий вариант:

```
#if !defined( _my_funcA_defined_)
#define _my_funcA_defined_
void my_funcA();
#endif /* _my_funcA_defined_ */
```

Директива #pragma

Директива #pragma используется для выдачи дополнительных указаний компилятору. Например, не выдавать предупреждений при компиляции, или вставить дополнительную информацию для отладчика. Конкретные возможности директивы #pragma у разных компиляторов различные.

При разработке программ с использованием MS Visual studio часто используется директива **#pragma once**, размещаемая в начале заголовочного файла. Это обеспечивает однократное включение заголовочного файла в случае его включения в программу несколько раз с помощью директивы **#include**. (Альтернативный способ решения этой же задачи -

использование директив условной компиляции).

Callback-функции

- Указатели на функции могут быть переданы в качестве параметров в другие функции, что позволяет выполнять разные вызовы в зависимости от условий выполнения.
- Функции, указатели на которые передаются в другие функции, и никогда не вызываемые напрямую, называются ***callback-функциями***.

Пример использования callback-функций

```
int max(int a, int b) {
  return (a>b ? a : b);
}
int min(int a, int b) {
  return (a<b ? a : b);
}
void PrintRezt (int f(int, int), int a, int b) {
  cout << ”Result is: ” << f(a, b) << endl;
}
…
int (*maxmin) (int, int);
…
if (…) //надо выводить минимум
 maxmin = min;
else
 maxmin = max;
…
PrintRezt(maxmin, 20, 40);
```