VIII

Сборка программ

Первое, с чего начнём данную лекцию, это пристальный взгляд на комманду **#include**:

```
1 #include <cmath>
2 #include <cstdio>
3 #include <cstdlib>
```

Большинство использовало её в своих программах. Как правило, всё, что об этой конструкции говорилось, это «включение соответствующих файлов в конкретную программу». Сейчас разберёмся, в какой момент это включение происходит, к какой части процесса компилирования относится эта команда и какие ей подобные доступны в арсенале C++.

Сборка программ

Рассмотрим принципиальную схему работы компилятора по преобразованию некоторой программы на C++ в исполняемый файл. Пусть дана программа:

```
1 #include <cstdio>
2
3 int main()
4 {
5   int val = 15;
6   printf("val = %d\n", val);
7 }
```

Не особо много работы она делает, но для дальнейшего анализа подойдёт.

Прежде, чем идти дальше

Далее речь будет идти об «сборке» программ — процессе преобразования кода на C++ в исполняемый или библиотечный файл. С этим процессом связана следующая терминология:

- **Трансляция** — преобразование исходного кода на конкретном языке программирования в **другое представление**. Под **другим представлением** может пониматься как код на другом языке программирования, в частности на ассемблере, так и файл с машинными кодами.
- **Компилятор** — программа, осуществляющая трансляцию для конкретного языка программирования.
- **Компиляция** — запуск всей работы компилятора по трансляции исходного кода в исполняемый или библиотечный файл.

Сборка программ

Сборка программ: стадии компиляции I

Характеризуем основные этапы каждой стадии со схемы слайда 5.

1. **Лексический анализ**.
   Во время данной стадии компилятор проверяет соответствие всех символов в файле с исходным кодом ожидаемой кодировке, удаляет лишние пробелы и переносы строк, убирает комментарии и осуществляет разбор кода на **лексемы**. Под этим термином понимаются отдельные наборы символов, представляющие важность с точки зрения синтаксических правил языка программирования. Для наглядного примера можно представить, что программа со слайда 4 преобразуется в следующий массив лексем:

Сборка программ: стадии компиляции II

```
"#include", "<cstdio>", "int", "main",
"(", ")", "{", "int", "val", "=",
"15", ";", "printf", "(",
"\"val = %d\n\"", ",", "val", ")", ";", "}"
```

Пример приведён с точностью до переносов строк и пробельных символов. Все лексемы заключены в двойные кавычки. Как можно заметить, знаки скобок, запятая и точка с запятой также представляют собой отдельные элементы лексического анализа.
После данной стадии компилятор, как правило, продолжает работать с подобным набором лексем.

Сборка программ: стадии компиляции III

2. **Работа препроцессора**.
   Препроцессор — специальная программа в рамках компилятора, которая осуществляет *предварительную обработку* исходного текста программы после лексического анализа. Управление этой обработкой осуществляется с помощью специальных команд, которые получили название **директивы препроцессора**. В C++ директивы начинаются с символа решётки «#», как пример — используемая во всех программах директива **#include**. Основной работой препроцессора является замена одних текстовых фрагментов на другие, плюс его возможности позволяют огранизовывать условные проверки ещё до выполнения стадии трансляции и даже прерывать процесс компиляции по заданным условиям. Доступные директивы подробно рассмотрим по завершении обсуждения процесса компиляции.

Сборка программ: стадии компиляции IV

Директива **#include** примечательна тем, что при её применении, препроцессор запускает стадию лексического анализа для того кода, который добавляется в программу из указанного файла. И это происходит рекурсивно при появлении **#include** во включаемом файле. Для примера, набор лексем со слайда 6 расширяется всеми лексемами из файла <**cstdio**>. Прежде чем переходить к следующей стадии, стоит обратить внимание на ответвление «создание единицы трансляции» (слайда 5). Под данным термином понимают весь набор лексем, получающийся после работы препроцессора. Единицы трансляции будут иметь значение, когда происходит разбор темы по разделению кода C++ на отдельные файлы (например, для создания собственной полезной библиотеки). Не уверен, что в этом семестре доберёмся, но, забегая вперёд, можно выделить два факта:

- каждый файл с исходным кодом на C++ (файлы с расширением «**.cpp**») после работы препроцессора преобразуется в отдельную единицу трансляции;
- каждую единицу трансляции компилятор может преобразовывать в двоичный (машинный) код независимо от других.

3. **Трансляция**.
   В данной стадии объеденены следующие действия компилятора:
   - трансляция кода на C++ в представлениие на языке ассемблера (которое зависит от архитетуры ЭВМ);
   - компилирование ассемблерного представления в машинный (двоичный, бинарный) код;
   - создание **объектного файла**.

Сборка программ: стадии компиляции VI

Интерес представляет **объектный файл**. В него компилятор помещает названия глобальных объектов из текущего файла с исходным кодом (идентификаторы функций и глобальных переменных), реализацию функций (уже в машинных кодах) и названия функций, которые используются из внешних библиотек (например, стандартная библиотека C++). Терминологически, каждое название именуют *символом*. Каждый символ имеет область видимости: либо его может использовать только код в данном объектном файле, либо он (а, следовательно, и связанная с ним функция или глобальная переменная) может быть использован из других объектных файлах. Для кода на слайде 4 объектный файл может определять символы следующим образом (получено с использованием компилятора **gcc**):

Сборка программ: стадии компиляции VII

```
0000000000000000 T main
                 U printf
```

В примере с кодом определена только функция **main** и используется функция **printf** тз стандартной библиотеки. Это отражает полученный объектный файл: первая строка говорит о том, что **main** определена в данном файле с исходном коде; буква «**T**» означает, что код этой функции находится в этом же объектном файле («T» — «Text»); «**U**» у функции **printf** — код функции находится вне данного объектного файла («U» — «Undefined»). Много нулей перед символом **main** это сопоставленное ему целое значение в шестнадцатиричном представлении. В ОС Windows компилятор от Microsoft обычно присваивает объектным файлам расширение «**.obj**», в других ОС (Mac OS, дистрибутивы Linux, BSD-системы) основным является — «**.o**».

Сборка программ: стадии компиляции VIII

Раз уж коснулись неглубокого погружения в объектные файлы, продемонстрируем одну особенность компиляторов C++. Добавим в пример собственную функцию:

```
1  #include <cstdio>
2
3  void incr_val(int& value, int by = 2);
4
5  int main()
6  {
7    int val = 15;
8    incr_val(val);
9    printf("val = %d\n", val);
10 }
11
12 void incr_val(int& value, int by)
13 {
14   value += 2;
```

```
15 }
```

Тогда символы в объектном файле могут выглядить следующим образом:

```
0000000000000000 T main
                 U printf
0000000000000072 T _Z8incr_valRii
```

И вместо символа с названием **incr_val** появился некий **_Z8incr_valRii**. Такое поведение компиляторов получило название *искажение имён* идентификаторов (англ. mangling names). Основная причина — существование перегрузки функций в C++. На данный момент правила по которым преобразуются имена функций не являются стандартизированными, поэтому каждый компилятор может придерживаться собственной стратегии переименования. Отдельно отметим, что искажение имён

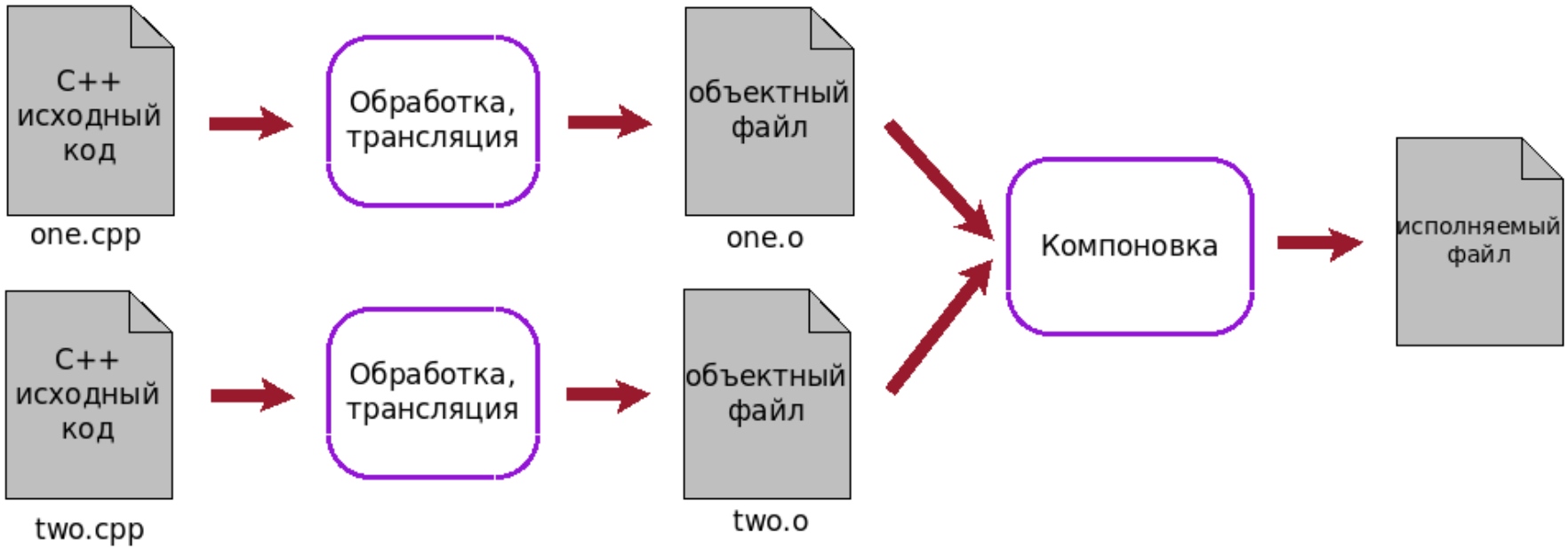
не распостраняется на функцию **main** и на функции стандартной библиотеки языка C (такие как **printf**, **scanf**, **sin**, **pow** и прочие).

4. **Компоновка** (linking).
На этой стадии компилятор берёт полученный объектный файл, осуществляет поиск тех символов, которые используются, но не определены в нём и создаёт итоговый *исполняемый файл*. Что неоднократно все проделывали на практиках с помощью пункта меню «Построить решение» в Visual Studio IDE. Поиск недостающих символов осуществляется по некоторым стандартным путям ОС, например в месте, где установлен сам компилятор (для примера, на ОС Windows путь вида «C:\Windows\Program Files\MSVC\lib») Формат исполняемого файла зависит от ОС, обычно в него входят команды для загрузки библиотечных файлов, которые

содержат используемые внешние функции, вызов функции **main** и некоторые другие несущественные на данном этапе рассмотрения действия.
Такая же схема применяется и при создании собственного *библиотечного файла*, только в этом случае компоновщик просто преобразует код функций из исходного кода в файл специального формата, из которого другие программы смогут делать соответствующие вызовы.

На основе всего перечисленного, если исходная программа разбивается на несколько файлов с исходным кодом, то общая схема работы компилятора может быть показана как:

Сборка программ: стадии компиляции XII



Кому скучно на дистационном обучении:

- не боитесь английского и терминологии: актуальный черновик стандарта C++ **eel.is/c++draft/lex** рассказывает об 9 фазах трансляции исходного кода в исполняемый или библиотечный файл;
- привыкли к Visual Studio IDE и интересно просмотреть объектные файлы: утилита **DUMPBIN** к вашим услугам **docs.microsoft.com/en-us/cpp/build/reference/dumpbin-reference**. В частности, обратить внимание на опцию **/SYMBOLS**;
- работаете с ОС отличной от Windows: утилиты **nm** и **objdump** делают аналогичную работу, что и программа в предыдущем пункте.

Директивы препроцессора

Далее переходим к тому, ради чего разбирали процесс работы компилятора: **директивы препроцессора**. Можно выделить следующие основные группы:

1. Директивы для включения других файлов в исходный код программы.
2. Директивы для определения текстовых замен.
3. Директивы для условной компиляции некоторой части исходного кода.
4. Единственная директива для информирования об ошибке и остановки работы компилятора.

Директивы препроцессора: включение файлов

Две формы:

```
(1) #include <file_name>
(2) #include "file_name"
```

Обе формы позволяют «включить» некоторый файл в исходный код программы. При включении становятся доступны все функции/типы/переменные, определённые во включаемом файле. Обе формы осуществляют поиск файлов, но стратегия поиска определяется конкретным компилятором. Как правило, форма **(1)** ищет указанный файл только в *стандартных путях поиска*. Форма **(2)** может сначало осуществлять поиск в той же директории, что и файл с исходным кодом. Если не найден — делается попытка поиска в стандартных путях.

Стандартные пути поиска библиотек зависят от способа, как компилятор языка был установлен в ОС. Также, через опции компилятора, могут быть добавлены дополнительные пути для поиска.

Директивы препроцессора: включение файлов

Начиная со стандарта C++17 доступна ещё директива для проверки, видит ли компилятор те файлы, которые хотим включить себе в программу

```
(1) __has_include( <file_name> )
(2) __has_include( "file_name" )
```

Обе формы возвращают целое значение «**1**», если файл найден. Иначе — «**0**». Поиск файлов происходит аналогично директиве **#include**.
Вопрос, реализован ли функционал этой директивы в Visual Studio IDE, требует проверки. Возможно, в версии **Visual Studio 2019** будут работать.

Директивы препроцессора: текстовые замены

Директивы, использующиеся для замены одного текста другим (определение **макросов**):

```
(1) #define <идентификатор>
(2) #define <идентификатор> <текст_для_замены>
(3) #define <идентификатор>(<параметры>) <текст>
(4) #undef  <идентификатор>
```

1. Определяет **идентификатор** для пустого макроса.
2. Определяет макрос замены **идентификатора** на **текст_для_замены**.
3. **Идентификатор** может получать параметры и использовать в подстовляемом тексте. Синтаксис параметров аналогичен функциям, за исключением отсутствия каких-либо упоминаний об типах.
4. Отменяет любой ранее опредлённый **идентификатор**.

Директивы препроцессора: текстовые замены

Примеры формы **(2)** создания макросов.

```
1  #define ROWS 10
2  #define COLS 15
3
4  #define AUTHOR "This is me\n"
5
6  double matrix[ROWS][COLS];
7  /* После работы препроцессора, выражение
8     будет преобразовано в следующее:
9       double matrix[10][15];
10 */
11
12 printf(AUTHOR);
13 // на консоли появится: "This is me";
```

Здесь каждый макрос при работе препроцессора просто заменяется тем набором символов, которые указаны в его определении.

Директивы препроцессора: текстовые замены

Используя форму **(3)**, можно создавать макросы в стиле функций: они принимают параметры и могут что-нибудь с ними делать. Все параметры в макросы передаются в виде текста (групп символов).

```
1 #define MAX(x, y) (x > y) ? x : y
2
3 int fifteen_or_five = MAX(15, 5);
4 printf("Answer is %d\n", fifteen_or_five);
```

Здесь макрос **MAX** при работе препроцессора подставляет переданные ему символы («15» и «5») в текстовую замену. И в третьей строке вместо макроса появляется выражение «$(15 > 5)\ ?\ 15 : 5$».

Директивы препроцессора: текстовые замены

Препроцессор в C++ содержит два оператора: «#» и «##». Первый оборачивает переданные ему операнд в *строковый литерал*, действие второго продемонстрируем на примере:

```
1  #define MAKE_FUNC(name, a) int fn_##name() { return a;}
2
3  MAKE_FUNC(first, 12)
4  MAKE_FUNC(second, 2)
5  MAKE_FUNC(third, 23)
6
7  #undef MAKE_FUNC
8  #define MAKE_FUNC 34
9
10 printf("first: %d\n", fn_first());
11 printf("second: %d\n", fn_second());
12 printf("third: %d\n", fn_third());
13 printf("MAKE_FUNC macro value: %d\n", MAKE_FUNC);
```

Директивы препроцессора: текстовые замены

Строки **(3 - 5)** примера с предыдущего слайда создают три функции с помощью объявленного макроса **MAKE_FUNC**.
Строки **(7 - 8)** демонстрируют *переопределение макроса*: отмену предыдущей замены и подстановку новой.
Вообще говоря, в макросах замену нельзя перенести на новую строку. Но это можно обойти с помощью использования *обратного слеша*:

```
1 #define MAKE_FUNC(name, a) int fn_##name() \
2                            {                \
3                              return a;      \
4                            }
```

Дело в том, что такие строки объединяются компилятором C++ в одну ещё на стадии **лексического анализа** (если после слеша сразу идёт символ переноса строки). Поэтому определение макроса оказывается корректным в момент начала работы препроцессора.

Директивы препроцессора: текстовые замены

Использование оператора «#»:

```
1 #define OUTPUT(text) printf(#text "\n");
2
3 OUTPUT(Text without quotes!);
4 // преобразуется в:
5 // printf("Text without quotes!" "\n");
```

Оператор оборачивает переданный аргумент в двойные кавычки.

Кроме того, поддерживается специальный синтаксис для произвольного количества аргументов:

```
1 #define ALL_TO_CONSOLE(...) puts(#__VA_ARGS__);
2
3 ALL_TO_CONSOLE(3.5, "text", char);
4 // преобразуется в:
5 // pus("3.5, \"text\", char");
```

Директивы препроцессора: текстовые замены

В C++ определён некоторый набор стандартных макросов. Некоторые из них:

Макрос	**Во что разворачивается**
__cplusplus	Версия стандарта C++, некоторое целое число. До стандарта C++11 разворачивается в **199711L**, при использовании стандарта C++11 — **201103L**, C++14 — **201402L**, C++17 — **201703L**. Буква «**L**» — число типа **long**
__FILE__	имя текущего файла, строка
__LINE__	номер текущей строки в файле, целое число
__DATE__	дата, когда происходит компиляция, строка
__TIME__	время, когда происходит компиляция, строка

Попробывать использовать их в коде и посмотреть на совершаемые подстановки предлагается самостоятельно.

Директивы препроцессора: условная компиляция

Условные директивы, использующиемся для задание логики при работе препроцессора:

```
(1) #if <выражение>
(2) #ifdef <название макроса>
(3) #ifndef <название макроса >
(4) #elif <выражение>
(5) #else
(6) #endif
```

В выражения могут входить: вычисления, состоящие только из констант; условные операторы, в том числе их комбинация; и опрератор препроцессора **defined**. Последний проверяет, определён ли переданный ему идентификатор как макрос. Если да, оператор возвращает значение «**1**», иначе — «**0**». В выражениях любое ненулевое значение трактуется как истинное, нулевое — как ложное. В принципе, похоже на конструкцию **if** из самого языка программирования.

Директивы препроцессора: условная компиляция

Пример возможностей условных директив

```
1  #define MACROS1 2
2
3  #ifdef MACROS1
4    printf("(1): defined\n");
5  #else
6    printf("(1): not defined\n");
7  #endif
8  #ifndef MACROS1
9    printf("(2): not defined\n");
10 #elif MACROS1 == 2
11   printf("(2): has value 2\n");
12 #else
13   printf("(2): not defined\n");
14 #endif
15 #if !defined(DCBA) && (MACROS1 < 5*2 - 3)
16   printf("(3): expression is true\n");
17 #endif
```

Рекомендуется самостоятельно в примере попробывать поменять значение/название макроса.

Директивы препроцессора: условная компиляция

Более реальный пример: отчистка экрана текстовой консоли при работе программы.
Иногда в текстовых программах появляется необходимость убрать с экрана всё, что видит пользователь и начать снова выводить текст. Для этого в текстовом режиме в ОС Windows есть команда **cls**, а в других ОС — **clear**. Обе команды могут быть вызваны через функцию **system** из файла <**cstdlib**>.
Напишем универсальную функцию, которая будет компилироваться в любой ОС.

Директивы препроцессора: условная компиляция

Получится что-то вида:

```
1 void clear_screen()
2 {
3 #ifdef WINDOWS_H
4   system("cls");
5 #else
6   system("clear");
7 #endif
8 }
```

Компиляторы для ОС Windows определяют макрос **WINDOWS_H**, который служит индикатором, что компилирование исходного хода происходит на соотвествующей ОС. В этом случае, в функции выше препроцессор уберёт строчку 6 и после его работы функция условно «примет» следующий вид:

```
1 void clear_screen() { system("cls"); }
```

Противоположную ветку препроцессор выберет на других ОС.

Директивы препроцессора: условная компиляция

Условная компиляция может быть полезна в процессе написания и отладки программы, когда нужно временно убрать часть кода. Общий шаблон следующий:

```
1  void some_important_func()
2  {
3  // code here...
4
5  /* group of statements 1 */
6  compute1();
7  check1();
8  /* group of statements 2 */
9  compute2();
10 check2();
11
12 // code here...
13 }
```

С помощью многострочного комментария невозможно сразу «выключить» строки **(6, 7, 9, 10)**, поскольку вложенные многострочные комментарии не допустимы.

Директивы препроцессора: условная компиляция

Требуемого результата можно достичь с помощью директивы **#if**

```
1  void some_important_func()
2  {
3  // code here...
4  #if 0
5  /* group of statements 1 */
6  compute1();
7  check1();
8  /* group of statements 2 */
9  compute2();
10 check2();
11 #endif
12 // code here...
13 }
```

В этом случае, все нужные строки будут убраны из программы препросцессором. Для обратного включения достаточно заменить нуль на единицу в пятой строке.

Директива препроцессора: прервать компиляцию

Директива для сообщения об ошибке имеет следующий вид

`#error <сообщение_об_ошибке>`

Сообщение может быть набором слов, необязательно заключённых в двойные кавычки.

Для примера,

```
1 #if defined(WINDOWS_H)
2   #error Not compile in Windows OS
3 #endif
```

или

```
1 #if __cplusplus < 201402L
2   #error Want support of C++14 and higer
3 #endif
```

Как только препроцессор встречает данную директиву, работа компилятора прекращается.

C++ и макросы

Современные рекомендации по написанию программ на C++ призывают избегать макросов. Макросы сложны для анализа со стороны сред разработки; они не имеют области видимости; в них легко сделать, но трудно найти ошибки.
Нужны константы — так определяем типизированные эквиваленты, хочется написать макрос-функцию — отставить, пишем отдельную функцию. Как правило, для макрос всегда может быть заменён языковыми средствами C++.
При этом, пока нет альтернатив использованию директив и макросов для условной компиляции.

Ключевое слово static

В любое объявление переменной и объявление/определение функции может быть добавлен спецификатор **static**.

```
1 static int var = 202;
2 static double my_rate(int first, int last);
```

По сути это ключевое слово означает, что какие-то свойства объекта языка (переменной или функции) будут неизменными. Иногда для подобных объектов используется термин *статичная переменная/функция*. Разберёмся, какие именно свойства остаются неизменными.

Ключевое слово static

Для переменных **static** влияет на два аспекта:

- **время жизни**: любая переменная, объявленная с данным спецификатором, инициализируется один раз и существует до конца работы программы;
- **видимость в объектном файле**: если спецификатор применён к **глобальной** переменной, то её область видимости ограничивается текущей единицей трансляции. На практике это означает, что такая глобальная переменная может быть использована только кодом конкретного объектного файла. На локальные переменные **static** никак не влияет с точки зрения области видимости.

Ключевое слово static

Для лучшей демонстрации, используем локальную статическую переменную

```
1  void demo_fn()
2  {
3    static int exec_times = 0;
4    exec_times++;
5    printf("demo_fn called %d times\n");
6  }
7
8  demo_fn();
9  demo_fn();
10 demo_fn();
```

В результате вызова функции из примера трижды подряд в консоли появятся строки:

```
demo_fn called 1 times
demo_fn called 2 times
demo_fn called 3 times
```

Пример вывода программы показывает, что строка **(3)** исходного кода (инициализация переменной **exec_times**) нулём была выполнена только один раз. Это как раз демонстрация первого аспекта статических переменных: даже локальная переменная получила время жизни, равное времени работы программы.
Статические локальные переменные подходят для реализации **мемоизации** в функциях: запоминание ранее расчитанных результатов и отсутствие расчёта при повторном запросе. Для примера, предположим, что в программе очень много раз используется вычисление факториала для значений до **30**. А факториалы от 31, от 32 и больших значений встречаются очень редко.

Ключевое слово static

Реализация функции вычисления факториала с мемоизацией:

```
1  const size_t mem_count = 30;
2
3  double factorial(unsigned n)
4  {
5    static double computed[mem_count + 1]
6                  = {1.0, 1.0, 2.0, 6.0, 0.0};
7    if (n <= mem_count && computed[n] != 0.0) {
8      return computed[n];
9    }
10
11   double res = 1.0;
12   for (size_t i = 2; i <= n; i++) {
13     res *= i;
14   }
15
16   if (n <= mem_count) { computed[n] = res; }
17
18   return res;
19 }
```

Ключевое слово **static**

Основные моменты:

- в строке **(5)** объявлен локальный статический массив на 31 элемент для хранения факториалов от **0!** до **30!** включительно. Значения факториалов с нуля по тройку определяются заранее;
- в строке **(7)** происходит проверка, что запрошенное значение попадает в запоминаемый диапазон и что факториал уже был вычислен;
- в строке **(16)** после расчёта факториала заполняем массив вычисленных значений по мере необходимости.

Расчёт факториала наглядно подходит для демонстрации идеи мемоизации: сокращение числа вычислений за счёт использования дополнительной памяти (массив то надо хранить). Кому не всё равно, сравните время работы данной реализации по сравнению с вычислением без данного трюка, скажем, на 100 миллионах последовательных вычислений факториала от случайных чисел в диапазоне $[5; 35]$.

Ключевое слово **static**

Для функций **static** влияет только на её видимость с точки зрения объектного файла. Правило простое: если функция объявлена с данным ключевым словом, то она может вызываться **исключительно** кодом в текущем объектном файле. В этом случае говорят, что область видимости ограничена *текущей единицей трансляции*. Если **static** не указан, функцию может вызывать код из других объектных файлов (при определённых нюансах разбиения кода).
Здесь же отметим области видимости глобальных переменных: если глобальная переменная объявлена константной, то она по умолчанию видна только в своём объектном файле. Иначе, к переменной может обращаться код из других объектных файлов. Опять же, более наглядно увидите, когда доберётесь до рассмотрения разделения кода на несколько файлов.

Безусловный переход

На второй лекции рассматривались конструкции для управления логикой работы программы. Их немного было: цикл (три формы), условный переход (конструкции **if**-**else**) да конструкция **switch** (сопоставление целочисленных значений). Но была не упомянута ещё одна возможность влиять на процесс выполнения инструкций с помощью инструкции **безусловного перехода** **goto**. Данная инструкция работает только в рамках тела отдельной функции.

Безусловный переход

Для работы инструкции **goto** необходимы **метки** (labels). Это символьные обозначения каких-либо инструкций в коде на C++. По сути, другими словами, это символьное обозначение некоторой непустой строки внутри функции. Синтаксис **метки**

`<название>:`

Рассмотрим пример:

```
1  int main()
2  {
3    int val = 10;
4  my_label:              // Объявили метку
5    printf("val = %d\n", val);
6    val--;
7    if (val > 0) {
8      goto my_label; // Передали управление строке,
9    }                    // следующей после метки.
10
11   printf("The end\n");
12 }
```

Безусловный переход

В примере всё время, пока значение переменной **val** не дойдёт до нуля, поток управления из вызова **goto** в восьмой строке, будет возвращаться на пятую строку.
В настоящее время использование **goto** не рекомендуется. Данная инструкция и читаемость фрагментов кода понижает, и может приводить к неочевидным логическим ошибкам. Да и C++, в частности, не разрешает безусловный переход в случае, если при переходе будет пропущена явная инициализация переменных. К примеру, следущий код вызовет ошибку компиляции:

```
1 int main()
2 {
3   int val = 10; if (val != 0) { goto finish; }
4
5   int other_val = 101; // Пытаемся перейти через
6 finish:                // явную инициализацию
7   printf("The end\n");
8 }
```

Безусловный переход

Одно из распостранённых применений **goto** это выход из вложенных циклов.

```
1  for (int i = 0; i < i_limit; i++) {
2    for (int j = 0; j < j_limit; j++) {
3      for (int k = 0; k < k_limit; k++) {
4        if (some_condition) { goto after_loop; }
5      }
6    }
7  }
8
9  after_loop:
10 // code here...
```

И действительно, с помощью стандартной остановки через **break** никак не прервать внешние циклы.

Безусловный переход

Как вариант, подобную задачу можно реализовать с использованием дополнительной булевской переменной.

```
1  bool to_next = true;
2  for (int i = 0; to_next && i < i_limit; i++) {
3    for (int j = 0; to_next && j < j_limit; j++) {
4      for (int k = 0; to_next &&  k < k_limit; k++) {
5        if (some_condition) { to_next = false; break; }
6      }
7    }
8  }
9
10 // code here...
```

Так что, общий совет на счёт **goto**: иметь ввиду его существование нужно, использовать — только если не осталось других способов построить логику функции.