КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 02. Состав и окружение языка программирования С

Преподаватель:
Поденок Леонид Петрович
505а-5, + 375 17 293 8039
prep@lsi.bas-net.by

ftp://student@lsi.bas-net.by/

Кафедра ЭВМ, 2021

Оглавление

Взаимодействие и правила поведения на занятиях

Язык общения — русский
Фото- и видеосъемка запрещается, болтовня и прочее мычание тоже
Использование мобильных гаджетов может вызвать проблемы

prep@lsi.bas-net.by
ftp://student@lsi.bas-net.by/

Старосты групп отправляют на **prep@** со своего личного ящика сообщение, в котором указывают свой телефон и ящик, к которому имеют доступ все студенты группы. В этом же сообщении в виде вложения приводят списки своих групп.

Формат темы этого сообщения: 010901 Фамилия И.О. Список группы

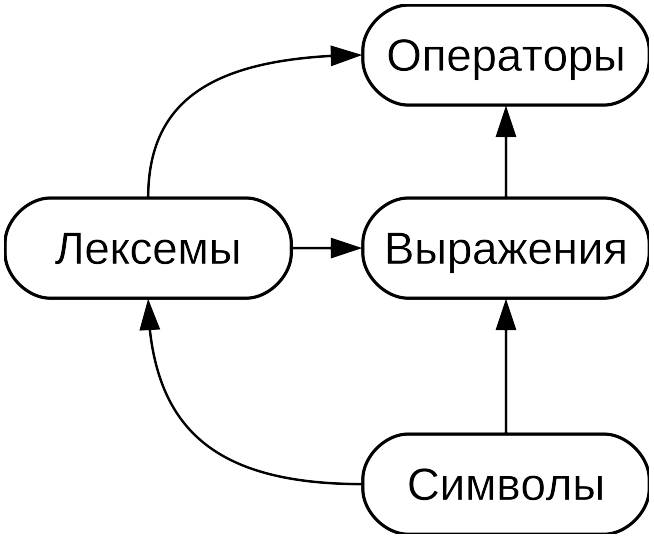
Subj: [010901 Фамилия И.О. Суть сообщения]

Правила составления сообщений
- текстовый формат сообщений;

Удаляется на сервере присланное в ящик prep@ все, что:
- без темы;
- имеет тему не в формате;
- содержит html, xml и прочий мусор;
- содержит рекламу, в том числе и сигнатуры web-mail серверов;
- содержит ссылки на облака и прочие гуглопомойки вместо прямых вложений.

Состав языка программирования

Естественный язык	Алгоритмический язык
Предложения Словосочетания Слова Символы	Операторы Выражения Лексемы Символы



Алфавит языка – набор неделимых знаков (символов), с помощью которых пишутся все тексты на языке.

Лексема – элементарная конструкция, минимальная единица языка, имеющая самостоятельный смысл.

Выражение задает правила **вычисления некоторого значения**.

Оператор задает законченное описание **некоторого действия**.

Для описания сложных действий требуется последовательность операторов.

Операторы могут объединяться в блоки (В языках C и C++ для этой цели используются фигурные скобки **{...}**).

Операторы бывают исполняемыми и неисполняемыми:

- исполняемые операторы задают действия над данными;
- неисполняемые операторы служат для описания данных – их называют операторами описания данных или просто описаниями/объявлениями/декларациями (declarations).

Помимо данных в С описываются (объявляются) и функции.

Каждый элемент языка определяется *синтаксисом* и *семантикой*.

- синтаксис устанавливает правила построения элементов языка.
- семантика определяет смысл и правила использования элементов языка.

Не всегда синтаксически правильное выражение генерирует однозначный и, более того, осмысленный код — эти случаи перечислены в стандарте языка и их следует всегда иметь ввиду.

Совокупность описаний и операторов (возможно объединенная каким-либо алгоритмом) представляет собой программу на алгоритмическом языке.

Для того, чтобы выполнить программу, ее необходимо перевести на язык, понятный процессору — в машинные коды. Этим занимается компилятор.

Процесс компиляции состоит из нескольких этапов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- создание на основе результатов анализов промежуточного кода;
- оптимизация промежуточного кода;
- создание объектного кода, в данном случае машинного.

Система типов

Система типов — совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным конструкциям, составляющим программу — таким как переменные, выражения, функции или модули.

Основная роль системы типов заключается в уменьшении числа ошибок в программах посредством определения интерфейсов между различными частями программы и последующей проверки согласованности взаимодействия этих частей.

Эта проверка может происходить статически на стадии компиляции или динамически во время исполнения, а также быть комбинацией обоих видов.

Статическая типизация — информационный объект (переменная, параметр подпрограммы, возвращаемое значение функции) связывается с типом в момент объявления объекта и тип не может быть изменен позже (переменная или параметр будут принимать, а функция — возвращать значения только этого типа).

Ада, C, C++, C#, Java, Паскаль, Fortran, Matlab…

Динамическая типизация — переменная связывается с типом в момент присваивания ей значения или инициализации, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

Smalltalk, Python, Objective-C, Ruby, PHP, Perl, JavaScript, Lisp.

Динамическая типизация упрощает написание программ для работы с меняющимся окружением и при работе с данными переменных типов. При этом отсутствие информации о типе на этапе компиляции повышает вероятность ошибок в исполняемых модулях.

Окружение языка программирования С[1]

Система программирования транслирует исходные файлы С и выполняет программы на С в двух средах системы обработки данных, которые называются:

- среда трансляции;
- среда исполнения.

Их характеристики определяют и ограничивают результаты выполнения соответствующих С-программ, построенных в соответствии с синтаксическими и семантическими правилами.

Наборы символов

Соответственно, в языке определено два набора символов:

- набор символов, в котором записаны исходные файлы (исходный набор символов);
- набор символов среды исполнения (интерпретируется во время выполнения).

Каждый набор делится на ***базовый набор символов*** и набор из нуля или более специфичных для локали элементов, (которые не являются членами базового набора символов), называемых ***расширенными символами***.

Объединенный набор также называется ***расширенным набором символов***.

Члены набора символов среды исполнения в среде трансляции, например, в символьной константе или строковом литерале (компоненты исходного текста) представляются соответствующими элементами исходного набора символов или escape-последовательностями, состоящими из обратной косой черты \, за которой следует один или несколько символов.

---

1 Std ISO/IEC 9899-2011[2012]

В базовом наборе символов среды исполнения существует байт со всеми битами, установленными в 0 (нулевой символ) — он используется для завершения символьной строки.

Оба базовых набора — исходный, и среды исполнения должны иметь следующие члены:

- 26 заглавных букв латинского алфавита **(ABCDEFGHIJKLMNOPQRSTUVWXYZ)**
- 26 строчных букв латинского алфавита **(abcdefghijklmnopqrstuvwxyz)**
- 10 десятичных цифр (**0 1 2 3 4 5 6 7 8 9**)
- 29 графических символов[2] (**!"#%&'( )*+,-./:;<=>?[\]^_{|}~**)
- символ пробела и управляющие символы, представляющие горизонтальную табуляцию (**HT**), вертикальную табуляцию (**VT**) и подачу формы (**FF**).

Представление каждого члена базовых наборов символов, как исходного, так и среды исполнения, помещается в байте.

В исходных файлах присутствует индикатор конца ***каждой строки текста*** — такой индикатор конца строки обрабатывается, как если бы он был одним символом новой строки (LF, CR, LF+CR).

В базовом наборе символов среды исполнения присутствуют управляющие символы:

- запрос на внимание (alert);
- возврат на один символ (backspace);
- возврат каретки (carriage return);
- символ новой строки (new line).

---

2 Не попали сюда три символа — '`', '**$**' и '**@**'. Тем не менее, gcc при использовании опций -std=c99 и -std=c11 позволяет использовать $ в идентификаторах. Опция -std=c90 (ansi) генерирует ошибку.

Если в исходном файле встречаются любые другие символы (кроме идентификатора, символьной константы, строкового литерала, имени заголовка, комментария или токена препроцесора, который никогда не преобразуется в токен), поведение не определено.

Буква — это заглавная буква или строчная буква, как определено выше. Другие символы, которые являются буквами в других алфавитах, не являются буквами в терминах С.

Многобайтовые символы

Исходный набор символов может содержать многобайтовые символы, используемые для представления членов расширенного набора символов.

Набор символов среды исполнения также может содержать многобайтовые символы, которые не обязательно должны иметь ту же кодировку, что имеет исходный набор символов.

Для обоих наборов символов соблюдается следующее:

- всегда присутствует базовый набор символов, и каждый символ этого набора кодируется как один байт;
- наличие, значение и представление любых дополнительных членов зависит от локали;
- многобайтовый набор символов может иметь разное кодирование (для разных локалей);
- байт со всеми нулевыми битами всегда интерпретируется как нулевой символ.

Семантика отображения символов

Активная позиция — это то место на устройстве отображения, где должен появиться следующий символ, выводимый функцией **`fputc()`**.

Цель записи печатного символа (как определено функцией **`isprint()`**) в устройство отображения — отобразить графическое представление этого символа в активной позиции и затем переместить активную позицию на следующую позицию в текущей строке.

Направление написания зависит от локали.

Если активная позиция находится в конечной позиции строки (если такая существует), поведение устройства отображения не определено.

Неграфические символы в наборе исполнительных символов представляются в виде буквенных *escape-последовательностей* и предназначены для выполнения действий на устройствах отображения следующим образом:

`\a` (alert) Создает звуковое или видимое оповещение без изменения активной позиции.

`\b` (backspace) Перемещает активную позицию на предыдущую позицию в текущей строке. Если активная позиция находится в начальной позиции строки, поведение устройства отображения не определено.

`\f` (form feed) Перемещает активную позицию в начальную позицию в начале следующей логической страницы.

`\n` (line feed) Перемещает активную позицию в начальную позицию следующей строки.

`\r` (carriage return) Перемещает активную позицию в начальную позицию текущей строки.

\t (horizontal tab) Перемещает активную позицию к следующей горизонтальной позиции табуляции в текущей строке. Если активная позиция находится на последней определенной горизонтальной позиции табуляции или превышает ее, поведение устройства отображения не определено.

\v (vertical tab) Перемещает активную позицию в начальную позицию следующей вертикальной позиции табуляции. Если активная позиция находится на последней определенной вертикальной позиции табуляции или превышает ее, поведение устройства отображения не определено.

Каждая из этих escape-последовательностей создает уникальное значение, определяемое реализацией, которое может храниться в одном объекте типа **char**.

Внешние представления в текстовом файле не обязательно должны быть идентичны внутренним представлениям.

Среда трансляции

Структура программы

Программа на C не обязательно должна быть оттранслирована вся одновременно. Текст программы хранится в модулях (unit), называемых *исходными файлами* (или *файлами препроцессора*).

Исходный файл вместе со всеми заголовками и исходными файлами, включенными в директиву препроцессора **`#include`**, называется *модулем предварительной трансляции* (preprocessing translation unit). Результат предварительной трансляции (препроцессинга) называется *модулем трансляции* (translation unit).

Ранее оттранслированные модули трансляции могут быть сохранены индивидуально или в библиотеках.

Отдельные единицы трансляции (модули трансляции) взаимодействуют между собой с помощью вызова функций, идентификаторы которых имеют *внешний тип связывания*, с помощью манипулирования объектами, идентификаторы которых имеют внешний тип связывания, или манипуляциями с файлами данных.

Модули трансляции могут быть оттранслированы отдельно, а затем с целью создания исполняемой программы связаны между собой.

```
// h.c
#include <stdio.h>

int main (int argc, char *argv[]) {

    printf("Hello World!\n");
}

$ gcc -o h h.c
$ gcc -std=c11 -W -Wall -Wextra -o h h.c
h.c: В функции «main»:
h.c:4:15: предупреждение: неиспользуемый параметр «argc» [-Wunused-parameter]
    4 | int main (int argc, char *argv[]) {
      |           ~~~~^~~~
h.c:4:27: предупреждение: неиспользуемый параметр «argv» [-Wunused-parameter]
    4 | int main (int argc, char *argv[]) {
      |                     ~~~~~~^~~~~~

$ gcc -std=c11 -W -Wall -Wextra -Wno-unused-parameter -o h h.c
$
```

Фазы трансляции

Программы на языках C и C++ обрабатываются следующим образом:

1) исходный текст сначала обрабатывается препроцессором, который исполняет директивы, которые содержатся в тексте и адресованы именно ему (**#**). В результате получается полный текст программы (модуль/единица трансляции).

2) Полный текст программы обрабатывается компилятором, который на этапе лексического анализа выделяет лексемы языка. Затем на этапе синтаксического анализа на основании грамматики языка распознаются выражения и операторы, построенные из лексем. На фазах 1 и 2 распознаются синтаксические ошибки, а правильный текст с точки зрения синтаксиса переводится в некоторую промежуточную форму.

3) Промежуточная форма может обрабатываться процессами оптимизации.

4) Оптимизированная промежуточная форма подается на стадию генерации кода, результатом которой является *объектный модуль*. Часто стадия генерации кода состоит из генерации текста на языке ассемблера целевого процессора, после чего следует стадия ассемблирования.

5) Объектный модуль может обрабатываться программой-библиотекарем, либо программой связывания (linker). В первом случае формируется библиотека объектных модулей, во втором – исполняемый модуль.

(##) Порядок применения синтаксических правил трансляции программ на языке С

Порядок применения синтаксических правил трансляции программ на языке С определяется следующими этапами:

1. Многобайтовые символы исходного физического файла отображаются, в зависимости от реализации, в исходный набор символов языка. При этом, если это необходимо, индикаторы конца строк заменяются на символы новой строки.

2. Каждый экземпляр символа обратной косой черты (\), за которым сразу следует символ новой строки, удаляется, в результате физические исходные строки объединяются для формирования логических исходных строк (нужно для «многострочных» макросов препроцессора).

Исходный файл, который не является пустым, должен заканчиваться символом новой строки, которому не должен предшествовать символ обратной косой черты.

3. Исходный файл раскладывается на лексемы препроцессора) и последовательности пробельных символов (включая комментарии). Каждый комментарий заменяется одним пробелом. Символы новой строки сохраняются.

4. Выполняются директивы препроцессора, расширяются вызовы макросов и выполняются выражения унарного оператора **`_Pragma`**.

Директива препроцессора **`#include`** вызывает рекурсивную обработку указанного заголовка или исходного файла от фазы 1 до фазы 4. Все директивы предварительной обработки затем удаляются.

5. Каждый элемент исходного набора символов и escape-последовательность в символьных константах и строковых литералах преобразуются в соответствующий элемент набора символов выполнения.

Если соответствующего символа нет, он преобразуется в определяемый реализацией символ, отличный от нулевого (wide).

6. Литеральные лексемы смежных строк объединяются.

7. Пробелы, разделяющие лексемы, больше не имеют значения. Каждая лексема препроцессора преобразуется в лексему языка. Полученные лексемы синтаксически и семантически анализируются и транслируются как единица трансляции.

8. Разрешаются все внешние ссылки на объекты и функции. Для разрешения внешних ссылок на функции и объекты, не определенные в текущем процессе трансляции, используются соответствующие компоненты библиотек.

Весь такой вывод транслятора собирается в образ программы, который содержит информацию, необходимую для выполнения в среде выполнения.

Диагностика

Диагностические сообщения выводятся только в случае нарушения синтаксиса или ограничений. В остальных случаях не выводится ничего.

Это значит, что внимательное чтение тех мест стандарта, где описаны случаи неопределенного поведения и прочие проблемы, является обязательным для программистов на C и C++.

Среда исполнения

Определены две среды исполнения — автономная (на «голом» железе) и размещающая (в операционной системе).

В обоих случаях запуск программы происходит, когда назначенная функция C вызывается средой выполнения. Перед запуском программы все объекты со статической продолжительностью хранения уже инициализированы (установлены их начальные значения).

Завершение программы возвращает управление среде выполнения.

Автономное окружение

В автономном окружении (в котором выполнение программы на C может происходить без какой-либо поддержки от операционной системы), имя и тип функции, вызываемой при запуске программы, определяются реализацией.

Любые библиотечные средства, доступные для автономной программы, кроме минимального набора, требуемого в разделе 4, определяются реализацией.

Результат завершения программы в автономной среде определяется реализацией.

Размещающее окружение

Размещающая среда не требуется, но должна соответствовать определенным спецификациям, если присутствует.

Запуск программы

Функция, вызываемая при запуске программы, называется **`main()`**.

Прототип для функции main() не объявляется.

Функция **`main()`** должна быть определена с типом возврата **`int`** и без параметров:

```
int main(void) { /* ... */ }
```

или с двумя параметрами[3] (именуемыми, например, как **`argc`** и **`argv`**, хотя могут использоваться любые имена, поскольку они являются локальными для функции, в которой они объявлены):

```
int main (int argc, char *argv[]) {/ * ... * /}
```

или эквивалентным образом, или каким-либо другим способом, определяемым конкретной реализацией. Если они объявлены, параметры главной функции подчиняются следующим ограничениям:

- Значение **`argc`** всегда неотрицательно;
- **`argv[argc]`** всегда является нулевым указателем.
- Если значение **`argc`** больше нуля, члены массива от **`argv[0]`** до **`argv[argc-1]`** включительно содержат указатели на строки, которым перед установкой программы передаются значения, определяемые реализацией среды хоста.

Это нужно, чтобы предоставить программе информацию, определенную до ее запуска, из какого бы места она не запускалась.

---

3 Стандарт POSIX позволяет использовать третий параметр — **`char *envp[]`**

- Если значение **argc** больше нуля, строка, на которую указывает **argv[0]**, представляет имя программы.

Если имя программы недоступно из среды хоста, **argv[0][0]** будет нулевым символом (пустая строка).

Если значение **argc** больше единицы, строки, на которые указывают **argv[1]** - **argv[argc-1]**, представляют параметры программы.

- Параметры **argc** и **argv**, а также строки, на которые указывает массив **argv[]**, могут изменяться программой и сохраняют свои последние сохраненные значения между запуском программы и ее завершением.

Исполнение программы

В размещающей среде программа может использовать все функции, макросы, определения типов и объекты из стандартной библиотеки.

Если тип возвращаемых данных из **main()** является типом, совместимым с **int**, возврат из вызова функции **main()** определяется аргументом функции **exit().**

При достижении **}**, которая прекращает исполнение **main()**, будет возвращено значение 0.

Минимальная программа на C

```
int main(void) {

}
```

Формальные системы описания синтаксиса

Форма Бэкуса-Наура

Форма Бэкуса-Наура (БНФ) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории.

БНФ-конструкция определяет конечное число *нетерминальных* символов и правила замены нетерминального символа на какую-то последовательность букв (лексем, *терминалов*) и нетерминальных символов.

Терминальные символы (терминалы) — это минимальные элементы грамматики, не имеющие собственной грамматической структуры.

Нетерминальные символы (символы) — это элементы грамматики, имеющие собственные имена и структуру .

Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики.

Процесс получения цепочки терминалов определяется поэтапно:

1) изначально имеется один символ (символы обычно заключаются в угловые скобки, а их название не несёт никакой информации).

2) этот символ заменяется на некоторую последовательность терминалов и символов, согласно одному из правил.

Процесс повторяется (на каждом шаге один из символов заменяется на последовательность, согласно правилу). В конце концов, получается цепочка, состоящая из терминалов и не содержащая символов. Это называется *выводом* цепочки из начального символа.

БНФ-конструкция состоит из нескольких предложений вида

<определяемый символ> ::= <посл.1> | <посл.2> | . . . | <посл.n>

Используется для описания синтаксиса языков программирования, данных, протоколов (например, в документах RFC) и т. д.

Определяемый символ, символы определения и альтернативы могут быть любые.

Пример описания синтаксиса числа с плавающей точкой (: := заменено на :)

```
<число с плавающей точкой> : <число с фикс. точкой>
                           | <число с фикс. точкой> <масштаб> ;

<число с фикс. точкой> : <беззнаковое число с фикс. точкой>
                       | <знак> <беззнаковое число с фикс. точкой> ;

<беззнаковое число с фикс. точкой> : <целое>                    # 123
                                   | <целое> <точка>            # 123.
                                   | <точка> <целое>            # .123
                                   | <целое> <точка> <целое> ;  # 123.456
<целое> : <цифра>
        | <целое> <цифра> ; // рекурсия
<цифра> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

<точка> : . | , ;
<знак>  : + | - ;
<масштаб> : <символ масштаба> <множитель> ;
<символ масштаба> : e | E ;
<множитель> : <целое> | <знак> <целое> ;
```

12.453 0. .123

-12e12 .123E-10

Расширенная форма Бэкуса-Наура

Расширенная форма Бэкуса-Наура (РБНФ/EBNF) — формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие.

Существует множество различных вариантов РБНФ. На сей счет есть стандарт **ISO/IEC 14977**.

Описание грамматики в РБНФ представляет собой набор правил, определяющих отношения между терминальными символами (*терминалами*) и нетерминальными символами (*нетерминалами*).

Терминальные символы — это минимальные элементы грамматики, не имеющие собственной грамматической структуры. В РБНФ терминальные символы — это либо предопределённые идентификаторы (имена, считающиеся заданными для данного описания грамматики, например, зарезервированные слова), либо цепочки (*литералы*) — последовательности символов в кавычках или апострофах.

Нетерминальные символы — это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики. В РБНФ каждый нетерминальный символ имеет имя, которое представляет собой строку символов.

РБНФ отличается от БНФ упрощением записи правил (вместо **::=** используется **=** , исключены угловые скобки) и введением двух синтаксических элементов — условное вхождение (выражение в прямых скобках **[выражение]** и повторение (выражение в фигурных скобках).

БНФ

```
<число с фикс. точкой> : <беззнаковое число с фикс. точкой>
                       | <знак> <беззнаковое число с фикс. точкой>
```

РБНФ

```
число-с-фикс-точкой = [ знак ] беззнаковое-число-с-фикс-точкой .
```

YACC, LEX, BISON, FLEX

`yacc` — компьютерная программа, служащая стандартным генератором синтаксических анализаторов (парсеров) в Unix-системах.

Название является акронимом «Yet Another Compiler Compiler» («ещё один компилятор компиляторов»).

Yacc генерирует парсер на основе аналитической грамматики, описанной в нотации BNF (форма Бэкуса-Наура) или контекстно-свободной грамматики.

Каждое правило описывает допустимую синтаксическую структуру и дает ей имя.

На выходе yacc выдаётся код парсера на языке программирования Си.

Парсер, генерируемый с помощью yacc, требует использования лексического анализатора. Обычно **`yacc`** используется совместно с генератором лексических анализаторов, в большинстве случаев это **`lex`** либо **`flex`**.

Функциональность и требования для **`lex`** и **`yacc`**. определены стандартом **IEEE POSIX P1003.2**.

`bison` — компьютерная программа, совместимая с **`yacc`**, служащая генератором синтаксических анализаторов в рамках проекта GNU. Помимо кода на C, выдает код и на C++.

Паттерн совместного использования лексического анализатора и парсера легко найти в интернете.

Лексические элементы языка

Две категории лексем — лексемы языка и лексемы препроцессора

Лексемы (token)

- ключевое (зарезервированное) слово (keyword);
- идентификатор (identifier);
- константа (constant);
- строковый литерал (string-literal);
- разделитель (punctuator); // скобки, точка, запятая, пробельные символы, …

Лексемы препроцессора (preprocessing-token)

- имя файла заголовка (header-name);
- идентификатор (identifier);
- числа препроцессора (pp-number);
- символьная константа (character-constant);
- строковый литерал (string-literal);
- разделитель (punctuator);
- одиночные непробельные символы, которые не совпадают лексически с другими категориями лексем препроцессора[4].

В процессе препроцессирования лексемы препроцессора преобразуются в лексемы языка.

---

4 Дополнительная категория, метки, используются внутри на этапе 4 трансляции (см. 6.10.3.3) и не встречается в исходных файлах.

Ограничения

Каждая *лексема препроцессора*, преобразованная в *лексему языка*, должна иметь лексическую форму ключевого слова, идентификатора, константы, строкового литерала или разделителя.

Это значит, что можно определить в качестве **{** и **}** что-нибудь другое, например, **BEGIN** и **END**, как это в паскале и прочей Модуле:

```
#include <stdio.h>
int main(void) {

  int a = 123;
  printf("a(in file scope): %i\n", a);

#define BEGIN {
#define END   }

  BEGIN                                 // {
    int a = 999;                        //   int a = 999;
    printf("a(in BEG-END): %i\n", a); //   printf("a(in BEG-END): %i\n", a);
  END                                   // }

}
$ gcc -o test -W -Wall test.c
$ ./test
a(in file scope): 123
a(in BEG-END)   : 999
```

Лексемы препроцессора могут быть разделены пробелами, которые состоят из комментариев (`/*... */`, `//`) или символов пробела (пробел, символ горизонтальной табуляции, символ новой строки, символ вертикальной табуляции и символ перевода формата), или из того и другого.

Пробелы могут появляться в лексеме препроцессора только как часть имени заголовка или между символами кавычек в символьной константе или строковом литерале.

Ключевые слова

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

Вышеуказанные лексемы (с учетом регистра) зарезервированы для использования в качестве ключевых слов и не должны использоваться иначе.

Ключевое слово `_Imaginary` зарезервировано для указания мнимых типов.

`char` — типы

`break` — управление потоком вычислений

`const` — квалификаторы

Идентификаторы

Идентификатор может обозначать:
- объект;
- функцию;
- тег[5] или член структуры, объединения или перечисления;
- имя определения типа (typedef name);
- имя метки;
- имя макроса;
- параметр макроса.

Один и тот же идентификатор может обозначать разные объекты в разных точках программы.

Член перечисления называется константой перечисления (*enumeration*).

Еще до семантической фазы трансляции программы любые вхождения имен макросов в исходном файле заменяются последовательностями *лексем препроцессора*, которые составляют их макроопределения.

5 тег структуры — имя ее типа

Области применения/видимости идентификаторов

Для каждого отдельного объекта, который обозначается идентификатором, идентификатор является видимым (то есть может использоваться) только в пределах определенной области текста программы, называемой ее областью действия или областью видимости (scope).

Различные объекты, обозначенные одним и тем же идентификатором, либо имеют разные области действия или находятся в разных пространствах имен.

Существует четыре вида областей действия:

- функция;
- файл;
- блок;
- прототип функции.

Прототип функции — это объявление функции, которая объявляет типы ее параметров и тип возврата.

Имя метки — единственный вид идентификатора, который имеет область действия функции. Он может использоваться (в операторе **`goto`**) в любом месте функции, в которой он появляется, и неявно объявляется своим синтаксическим внешним видом (за именем метки следует **`:`** и какой-нибудь оператор).

Идентификатор имеет область действия, определяемую размещением его объявления.

Если объявление (декларация) или спецификатор типа, который объявляет идентификатор, появляется вне какого-либо блока или списка параметров, идентификатор имеет ***область видимости файла***, которая заканчивается в конце модуля трансляции.

Если объявление или спецификатор типа, которые объявляют идентификатор, появляются внутри блока или в списке объявлений параметров в определении функции, идентификатор имеет ***область видимости блока***, которая заканчивается в конце соответствующего блока.

```
int p = 3, q = 5;  // объявление переменных с инициализацией

int foo(int, int); // объявление функции (прототип)

int main(int argcnt, char *argvalue[]) {

    int rc = foo(6, 7);
    return(0);
}

int foo(int p, int s) {   // параметр p "экранирует" p
                          // из области видимости файла
    int r = 123;          // область видимости r – локальный блок
    return p + q + s + r; // q – область файла, p и s – локальный блок
}
```

Если объявление или спецификатор типа, который объявляет идентификатор, появляется в списке объявлений параметров в прототипе функции (не входит в определение функции), идентификатор имеет ***область действия прототипа функции***, которая заканчивается в конце декларатора функции.

```
int p = 3;
int q = 5;
int s = 7;                  // параметры p, q и s видны в пределах файла

int foo(int p, int s);      // параметры p и s видны только в пределах (...)

int main(int argc, char *argv[]) {

    int rc = foo(6, 7);
    return(0);
}

int foo(int p, int s) {     // параметр p "экранирует" p
                            // из области видимости файла
    int r = 123;            // область видимости r – локальный блок
    return p + q + s + r; // q – область файла, p и s – локальный блок
}
```

Если идентификатор обозначает две разные сущности в одном и том же пространстве имен, области действия могут перекрываться.

Если это так, область действия одной сущности (внутренняя область) заканчивается строго перед областью действия другой сущности (внешняя область).

Во внутренней области видимости обозначает сущность, объявленную во внутренней области; сущность, объявленная во внешней области видимости, оказывается скрыта (и не видна) во внутренней области видимости.

```
int p = 3, q = 5, s = 7;   // параметры p, q и s видны в пределах файла

int foo(int p, int s) {    // параметры p и s "экранируют" p и s
                           // из области видимости файла
    int r = 123;           // область видимости r – функция foo()
    {
        int r = 321;       // область видимости r – блок
        printf("%d\n", r); // печатает r из области видимости блока
    }
    printf("%d\n", r);     // печатает r из области видимости функции
    return p + q + s + r;  // q – область файла, p, s – область ф-ции
}
```

Теги структуры, объединения и перечисления имеют область действия, которая начинается сразу после появления тега в спецификаторе типа, который объявляет данный тег.

```
struct foo_s {
    struct foo_s *next;
    struct foo_s *prev;
    int a;
    int b;
};
```

Каждая константа перечисления имеет область действия, которая начинается сразу после появления ее определяющего перечислителя в списке перечислителей.

```
enum color {
    red,
    green,
    blue = red;
};
```

Любой другой идентификатор имеет область действия, которая начинается сразу после завершения его объявления.

Типы связывания идентификаторов (linkage)

Идентификатор, объявленный в разных областях или в одной и той же области более одного раза, может создаваться для ссылки на один и тот же объект или функцию с помощью процесса, называемого ***связыванием*** (компоновкой, сборкой).

Существует три типа связывания:

- внешнее;
- внутреннее;
- никакого.

В **наборе единиц трансляции и библиотек**, составляющих целую программу, каждое объявление определенного идентификатора с *внешним типом связывания* обозначает один и тот же объект или функцию.

В пределах **одной единицы трансляции** каждое объявление идентификатора *с внутренним типом связывания* обозначает один и тот же объект или функцию.

Идентификатор имеет внутренний тип связывания если объявление идентификатора с областью видимости файл для объекта или функции содержит спецификатор класса хранения[6] **`static`**.

Каждое объявление идентификатора *без типа связывания* обозначает уникальную сущность.

---

6 Длительность хранения объектов

Для идентификатора, объявленного с помощью спецификатора класса хранения **extern** в области видимости, в которой видно предыдущее объявление этого идентификатора, если в предыдущем объявлении указана внутренний или внешний тип связи, тип связи идентификатора в более позднем объявлении такой же, как тип связи, указанный в предыдущем обяъвлении (не меняет предыдущего).

Если не видно никакого предыдущего объявления или если в предыдущем объявлении не указан тип связи, то идентификатор имеет внешний тип связи.

Если объявление идентификатора для функции не имеет спецификатора класса хранения, его тип связи определяется точно так, как если бы он был объявлен с помощью спецификатора класса хранения **extern**.

Если объявление идентификатора для объекта не имеет спецификатора класса хранения и имеет область действия файла, его тип связи является внешним.

Следующие идентификаторы не имеют никакого типа связи:

- идентификатор, объявленный как что-либо, кроме объекта или функции;
- идентификатор, объявленный как параметр функции;
- идентификатор в области блока для объекта, объявленного без спецификатора класса хранения **extern**.

Если внутри единицы трансляции появляется один и тот же идентификатор как с внутренним, так и с внешним типом связи, поведение не определено.

Пространства имен идентификаторов

Если в какой-либо точке исходного модуля (единицы трансляции) видно более одного объявления определенного идентификатора, неоднозначность использования, которое ссылается на разные сущности, устраняется синтаксическим контекстом.

Таким образом, существуют отдельные пространства имен для различных категорий идентификаторов, а именно:

- имен меток;

- тегов структур, объединений и перечислений (устраняющих неоднозначность, следуя любому из ключевых слов **`struct`**, **`union`** или **`enum`**);

- членов структур или объединений — каждая структура или объединение имеет отдельное пространство имен для своих членов;

- всех остальных идентификаторов, называемых обычными идентификаторами (объявленными в обычных объявлениях или в качестве констант перечисления).

Длительность хранения объектов

У объекта есть срок хранения, который определяет его время жизни. Существует четыре длительности хранения:

- статическое (**`static`**);
- потоковое (**`_Thread_local`**);
- автоматическое;
- выделенное[7].

Время жизни объекта — это часть выполнения программы, в течение которой гарантированно для него зарезервирована память.

Объект существует, имеет постоянный адрес и сохраняет свое последнее сохраненное значение в течение всего времени его жизни.

Если на объект ссылаются вне его времени жизни, поведение не определено.

Значение указателя становится неопределенным, когда объект, на который он указывает (или только что прошедший), достигает конца своего времени жизни.

Объект, идентификатор которого объявлен с помощью спецификатора класса хранения **`_Thread_local`**, имеет ***длительность хранения потока***. Его время жизни — это время полного выполнения потока, для которого он создан, и значение, которое он хранит, инициализируется при запуске потока.

Для каждого потока существует отдельный объект, и использование объявленного имени в выражении относится к объекту, связанному с потоком, который выполняет вычисление выражения.

---

7 Выделенное хранение описано в 7.22.3

Объект, идентификатор которого объявлен без привязки и без статического спецификатора класса хранения, имеет автоматическую продолжительность хранения, как и некоторые составные литералы. Результат попытки косвенного доступа к объекту с автоматической продолжительностью хранения из потока, отличного от того, с которым связан объект, определяется реализацией.

Для объекта, тип которого не является массивом переменной длины, время жизни простирается от входа в блок, с которым он связан, до тех пор, пока выполнение этого блока не закончится каким-либо образом.

Вход во внутренний блок или вызов функции приостанавливает, но не прекращает выполнение текущего блока.

Если вход в блок ***рекурсивный***, каждый раз создается новый экземпляр объекта. Начальное значение объекта в этом случае не определено.

Если для объекта указана ***инициализация***, она выполняется каждый раз, когда поток управления при выполнении блока доходит до объявления или составного литерала.

В противном случае значение объекта становится неопределенным.

Универсальные имена символов

universal-character-name:

\u *hex-quad*

\U *hex-quad hex-quad*

hex-quad:

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

Универсальные имена символов могут использоваться в идентификаторах, символьных константах и строковых литералах для обозначения символов, которые не входят в базовый набор символов.

Имя универсального символа **\Unnnnnnnn** обозначает символ, чей восьмизначный короткий идентификатор (как указано в ИСО/МЭК 10646) равен **nnnnnnnn**.

Аналогично, имя универсального символа **\unnnn** обозначает символ, чей четырехзначный короткий идентификатор равен **nnnn** (и чей восьмизначный короткий идентификатор **0000nnnn**).

Ограничения

Имя универсального символа не должно указывать символ, у которого короткий идентификатор (ASCII) меньше **00A0**, кроме **0024** (**$**), **0040** (**@**) или **0060** (**‘**), и ни один из символов в диапазоне от **D800** до **DFFF** включительно[8].

8 Символы базового набора и кодовых позиций, зарезервированные ISO/IEC 10646 для управляющих символов, символа DELETE, и S-зоны (зарезервировано для UTF–16).

Константы

- целые константы;
- константы с плавающей точкой;
- константы перечислений;
- символьные константы.

Семантика

Каждая константа имеет тип, определяемый ее формой и значением.

Ограничения

Каждая константа должна иметь тип, а значение константы должно находиться в диапазоне представимые значения для его типа.

Строковые литералы

Синтаксис:

string-literal:

[encoding-prefix] **"** *[s-char-sequence]***"**

encoding-prefix:

{ **u8** | **u** | **U** | **L** }

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

любой член исходного набора символов за исключением двойной кавычки **"**, *обратной наклонной черты* ****, *или escape-последовательности символа новой строки* **\n**

Строковый литерал представляет собой последовательность из нуля или более много-байтовых символов, заключенных в двойные кавычки, например как **"xyz"**.

Строковый литерал UTF-8 имеет такой же вид, за исключением префикса **u8**.

Широкий строковый литерал имеет такой же вид, за исключением префикса **L**, **u** или **U**.

Одинарная кавычка может быть представлена либо самой собой **'**, либо escape-последовательностью **\'**. Двойная кавычка **"** должна быть представлена escape-последо-вательностью **\"**.

Ограничения

Последовательность смежных строковых литералов не должна включать в себя ***одновременно*** строковые литералы и строковые литералы UTF-8.

Семантика*

Последовательности многобайтовых символов, заданные любой последовательностью смежных символов и одинаковыми префиксами строковых литеральных ***лексем***, объединяются в одну последовательность многобайтовых символов.

Если какой-либо из токенов имеет префикс кодирования, результирующая многобайтовая последовательность символов обрабатывается как имеющая тот же префикс, в противном случае он рассматривается как строковый литерал.

К каждой многобайтовой символьной последовательности, полученной из строкового литерала добавляется нулевой байт, после чего данная многобайтовая символьная последовательность используется для инициализации массива статической длительности и постоянной длины, достаточных для содержать эту последовательность.

Для строковых и символьных литералов элементы массива имеют тип **`char`** и инициализируются отдельными байтами из многобайтовой последовательности символов.

Для строковых литералов UTF-8 элементы массива имеют тип **`char`** и инициализируются символами многобайтовой последовательности символов, как они кодируются в UTF-8.

Для широких строковых литералов, начинающихся с буквы **`L`**, элементы массива имеют тип **`wchar_t`** и инициализируются последовательностью широких символов соответствующей многобайтовой последовательности символов.

Для широких строковых литералов, начинающихся с буквы **u** или **U**, элементы массива имеют тип **char16_t** или **char32_t** соответственно и инициализируются последовательностью широких символов, соответствующей последовательности многобайтовых символов (определяется вызовами **mbrtoc16()** или **mbrtoc32()** в соответствии с текущей локалью).

Изменение элементов этих массивов → неопределенное поведение программы.

Пример 1

Пара смежных символьных строковых литералов

```
"\x12" "3"
```

создает строковый литерал, содержащий два символа со значениями **'\x12'** и **'3'**.

Пример 2

Каждая из последовательностей

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

эквивалентна строковому литералу

```
L"abc"
```