acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . CfaqANSI :

ANSI C

Поиск

Поиск

Раздел «Язык Си»

Главная
Зачем учить С?
Определения

Инструменты:

Поиск
Изменения
Index
Статистика

Разделы

Информация
Алгоритмы
Язык Си
Язык Ruby
Язык Ассемблера
El Judge
Парадигмы
Образование
Сети
Objective C

Logon>>

5.1

Q: Что такое стандарт ANSI C ?

A: В 1983 году Американский институт национальных стандартов (ANSI) учредил комитет X3J11, чтобы разработать стандарт языка С. После длительной и трудной работы, включающей выпуск нескольких публичных отчетов, работа комитета завершилась 14 декабря 1989 г.созданием стандарта ANS X3.159-1989. Стандарт был опубликован весной 1990 г. В большинстве случаев ANSI C узаконил уже существующую практику и сделал несколько заимствований из C++ (наиболее важное - введение прототипов функций). Была также добавлена поддержка национальных наборов символов (включая подвергшиеся наибольшим нападкам трехзнаковые последовательности). Стандарт ANSI C формализовал также стандартную библиотеку.

Опубликованный стандарт включает "Комментарии" ("Rationale"), в которых объясняются многие решения и обсуждаются многие тонкие вопросы, включая несколько затронутых здесь. ("Комментарии" не входят в стандарт ANSI X3.159-1989, они приводятся в качестве дополнительной информации.)

Стандарт ANSI был принят в качестве международного стандарта ISO/IEC 9899:1990, хотя нумерация разделов иная (разделы 2 - 4 стандарта ANSI соответствуют разделам 5 - 7 стандарта ISO), раздел "Комментарии" не был включен.

5.2

Q: Как получить копию Стандарта?

A: ANSI X3.159 был официально заменен стандартом ISO 9899. Копию стандарта можно получить по адресу

```
American National Standards Institute
11 W.   42nd St., 13th floor
New York, NY   10036  USA
(+1) 212 642 4900
```

или

```
   Global Engineering Documents
   2805 McGaw Avenue
   Irvine, CA  92714  USA
   (+1) 714 261 1455
   (800)   854 7179  (U.S.   & Canada)
```

В других странах свяжитесь с местным комитетом по стандартам или обратитесь в Национальный Комитет по Стандартам в Женеве

```
     ISO Sales
     Case Postale 56
     CH-1211 Geneve 20
     Switzerland
```

Цена составляет в ANSI $130, в Global Engineering Documents - $160. Копии оригинального стандарта X3.159, включающие "Комментарии", попрежнему доступны за $205.00 (ANSI) или за $200.50 (Global Engineering Documents). Отметим, что комитет ANSI для поддержки своей деятельности получает доход от продажи отпечатанных копий стандарта, так что электронные копии *недоступны*.

Книга Герберта Шилдта с обманчивым названием "Комментарии к стандарту C" содержит лишь несколько страниц стандарта ISO 9899; опубликована издательством Osborne/McGraw-Hill, ISBN 0-07-881952-0 и продается примерно за $40. (Есть мнение, что различие в цене между официальным стандартом и комментарием Герберта Шилдта соответствует ценности комментария).

Текст "Комментариев" (не всего стандарта) теперь доступен через ftp ftp.uu.net (см. вопрос 17.12) в директории doc/standards/ansi/ X3.159-1989. "Комментарии" были также изданы издательством Silicon Press, ISBN 0-929306-07-4.

5.3

Q: Есть ли у кого-нибудь утилиты для перевода C-программ, написанных в в старом стиле, в ANSI C и наоборот? Существуют ли программы для автоматического создания прототипов?

A: Две программы, protoize и unprotoize осуществляют преобразование в обе стороны между функциями, записанными в новом стиле с прототипами, и функциями, записанными в старом стиле. (Эти программы не поддерживают полный перевод между "классическим" и ANSI C). Упомянутые программы были сначала вставками в FSF GNU компилятор C, gcc, но теперь они - часть дистрибутива gcc; смотри директорий pub/gnu на prep.ai.mit.edu (=18.71.0.3=8), или в других архивах FSF.

Программа unproto ((/pub/unix/unproto5.shar.Z на ftp.win.tue.nl - это фильтр, располагающийся между препроцессором и следующим проходом компилятора - на лету переводит большинство особенностей ANSI C в традиционный C.

GNU пакет GhostScript? содержит маленькую программу ansi2knr.

Есть несколько генераторов прототипов, многие из них - модификации программы lint. Версия 3 программы CPROTO была помещена в конференцию comp.sources.misc в марте 1992 г. Есть другая программа, которая называется ctxtract. См. вопрос 17.12.

В заключение хочется спросить: так ли уж нужно преобразовывать огромное количество старых программ в ANSI C? Старый стиль написания функций все еще допустим.

Примечание редактора: В библиотеке МФТИ еще остались старые издания K&R (твердый переплет, бежевого или серого цвета, год издания *до* 2000). Если есть возможность, *не* берите его. Это издание содержит старый стиль задания фукнций и тп. Молодые преподаватели информатики будут обескуражены вашими археологическими знаниями во время зачета.

5.4

Q: Я пытаюсь использовать ANSI- строкообразующий оператор #, чтобы вставить в сообщение значение символической константы, но вставляется формальный параметр макроса, а не его значение.

A: Необходимо использовать двухшаговую процедуру для того чтобы макрос раскрывался как при строкообразовании

```
#define str(x) #x
#define xstr(x) str(x)
#define OP plus
char *opname = xstr(OP);
```

Такая процедура устанавливает opname равным plus, а не OP. Такие же обходные маневры необходимы при использовании оператора склеивания лексем ##, когда нужно соединить значения (а не имена формальных параметров) двух макросов.

Смотри: ANSI Разд. 3.8.3.2, Разд. 3.8.3.5 пример с. 93.

5.5

Q: Не понимаю, почему нельзя использовать неизменяемые значения при инициализации переменных и задании размеров массивов, как в следующем примере:

```
const int n = 5;
int a[n];
```

A: Квалификатор const означает "только для чтения". Любой объект квалифицированный как =cons=t, представляет собой нормальный объект, существующий во время исполнения программы, которому нельзя присвоить другое значение. Следовательно, значение такого объекта - это *не* константное выражение в полном смысле этого слова. (В этом смысле C не похож на C++). Если есть необходимость в истинных константах, работающих во время компиляции, используйте препроцессорную директиву #define.

Смотри: ANSI Разд. 3.4.

5.6

Q: Какая разница между char const *p и char * const p ?

A: char const *p - это указатель на постоянную литеру (ее нельзя изменить); char * const p - это неизменяемый указатель на переменную (ее можно менять) типа char. Зарубите это себе на носу. См. также 10.4.

Смотри: ANSI Разд. 3.5.4.1 .

5.7

Q: Почему нельзя передать char ** функции, ожидающей const char ** ?

A: Можно использовать *указатель-на-T* любых типов T, когда ожидается *указатель-на-const-T*, но правило (точно определенное исключение из него), разрешающее незначительные отличия в *указателях*, не может применяться рекурсивно, а только на самом верхнем уровне.

Необходимо использовать точное приведение типов (т.е. в данном случае (const char **)) при присвоении или передаче указателей, которые имеют различия на уровне косвенной адресации, отличном от первого.

Смотри: ANSI Разд. 3.1.2.6 с. 26, Разд. 3.3.16.1 с. 54, Разд. 3.5.3 с. 65.

5.8

Q: Мой ANSI компилятор отмечает несовпадение, когда встречается с декларациями

```
extern int func(float);

int func(x)
float   x;
{...
```

A: Вы смешали декларацию в новом стиле extern int func(float); с определением функции в старом стиле int func(x) float x;. Смешение стилей, как правило, безопасно (см. вопрос 5.9), но только не в этом случае. Старый C (и ANSI C при отсутствии прототипов и в списках аргументов переменной длины) "расширяет" аргументы определенных типов при передаче их функциям. Аргументы типа float преобразуются в тип double, литеры и короткие целые преобразуются в тип int. (Если функция определена в старом стиле, параметры автоматически преобразуются в теле функции к менее емким, если таково их описание там.).

Это затруднение может быть преодолено либо с помощью определений в новом стиле,

```
int func(float x) { ... }
```

либо с помощью изменения прототипа в новом стиле таким образом, чтобы он соответствовал определению в старом стиле:

```
extern int func(double);
```

(В этом случае для большей ясности было бы желательно изменить и определение в старом стиле так, чтобы параметр, если только не используется его адрес, был типа double).

Возможно, будет безопасней избегать типов char, short int, float для возвращаемых значений и аргументов функций.

Смотри: ANSI Разд. 3.3.2.2 .

5.9

Q: Можно ли смешивать определения функций в старом и новом стиле?

A: Смешение стилей абсолютно законно, если соблюдается осторожность (обратите особое внимание на вопрос 5.8). Заметьте, однако, что определение функций в старом стиле считается выходящим из употребления, и в один прекрасный момент поддержка старого стиля может быть прекращена.

5.10

Q: Почему объявление

```
extern f(struct x {int s;} *p);
```

порождает невнятное предупреждение "struct x introduced in prototype scope" (структура объявлена в зоне видимости прототипа)?

A: В странном противоречии с обычными правилами для областей видимости структура, объявленная только в прототипе, не может быть совместима с другими структурами, объявленными в этом же файле. Более того, вопреки ожиданиям тег структуры не может быть использован после такого объявления (зона видимости объвления простирается до конца прототипа). Для решения проблемы необходимо, чтобы прототипу предшествовало "пустое" объявление

```
struct x;
```

, которое зарезервирует место в области видимости файла для определения структуры x. Определение будет завершено объявлением структуры внутри прототипа.

Смотри: ANSI Разд. 3.1.2.1 с. 21,Разд. 3.1.2.6 с. 26, Разд. 3.5.2.3 с. 63.

5.11

Q: У меня возникают странные сообщения об ошибках внутри кода, "выключенного" с помощью #ifdef.

A: Согласно ANSI C, текст, "выключенный" с помощью #if, #ifdef, или #ifndef должен состоять из "корректных единиц препроцессирования". Это значит, что не должно быть незакрытых комментариев или кавычек (обратите особое внимание, что апостроф внутри сокращенно записанного слова смотрится как начало литерной константы). Внутри кавычек не должно быть символов новой строки. Следовательно, комментарии и псевдокод всегда должны находиться между непосредственно предназначенными для этого символами начала и конца комментария /* и */. (Смотрите, однако, вопросы 17.14 и 6.7).

Смотри: ANSI Разд. 2.1.1.2 с. 6, Разд. 3.1 с. 19 строка 37.

5.12

Q: Могу я объявить main как void, чтобы прекратились раздражающие сообщения "main return no value"? (Я вызываю exit(), так что main ничего не возвращает).

A: Нет. main должна быть объявлена как возвращающая int и использующая либо два , либо ни одного аргумента (подходящего типа). Если используется exit(), но предупреждающие сообщения не исчезают, Вам нужно будет вставить лишний return, или использовать, если это возможно, директивы вроде "notreached".

Объявление функции как void просто не влияет на предупреждения компилятора; кроме того, это может породить другую последовательность вызова/возврата, несовместимую с тем, что ожидает вызывающая функция (в случае main это исполняющая система языка C).

Смотри: ANSI Разд. 2.1.2.2.1 с. 7-8.

5.13

Q: В точности ли эквивалентен возврат статуса с помощью exit(status) возврату с помощью return?

A: Формально, да, хотя несоответсвия возникают в некоторых старых нестандартных системах, в тех случаях, когда данные, локальные для main(), могут потребоваться в процессе завершения выполнения (может быть при вызовах setbuf() или atexit()), или при рекурсивном вызове main().

Смотри: ANSI Разд. 2.1.2.2.3 с. 8.

5.14

Q: Почему стандарт ANSI гарантирует только шесть значимых символов (при отсутствии различия между прописными и строчными символами) для внешних идентификаторов?

A: Проблема в старых компоновщиках, которые не зависят ни от стандарта ANSI, ни от разработчиков компиляторов. Ограничение состоит в том, что только первые шесть символов *значимы*, а не в том, что длина идентификатора ограничена шестью символами. Это ограничение раздражает, но его нельзя считать невыносимым. В Стандарте оно помечено как "выходящее из употребления", так что в следующих редакциях оно, вероятно, будет ослаблено.

Эту уступку современным компоновщикам, ограничивающим количество значимых символов, обязательно нужно делать, не обращая внимания на бурные протесты некоторых программистов. (В "Комментариях" сказано, что сохранение этого ограничения было "наиболее болезненным". Если Вы не согласны или надеетесь с помощью какого-то трюка заставить компилятор, обремененный ограничивающим количество значимых символов компоновщиком, понимать большее количество этих символов, читайте превосходно написанный раздел 3.1.2 X3.159 "Комментариев" (см. вопрос 5.1), где обсуждается несколько такого рода подходов и объясняется, почему эти подходы не могут быть узаконены.

Смотри: ANSI Разд. 3.1.2 с. 21, Разд. 3.9.1 с. 96, Rationale Разд. 3.1.2 с. 19-21.

Примечание редактора: Не волнуйтесь. Ограничение в 6 значимых символов давно потеряло силу.

5.15

Q: Какая разница между memcpy и memmove?

A: memmove гарантирует правильность операции копирования, если две области памяти перекрываются. memcpy не дает такой гарантии и, следовательно, может быть более эффективно реализована. В случае сомнений лучше применять memmove.

Смотри: ANSI Разд. 4.11.2.1, 4.11.2.2, Rationale Разд.4.11.2 .

5.16

Q: Мой компилятор не транслирует простейшие тестовые программы, выдавая всевозможные сообщения об ошибках.

A: Видимо, Ваш компилятор разработан до приема стандарта ANSI и поэтому не способен обрабатывать прототипы функций и тому подобное. См. также вопросы 5.17 и 17.2.

Примечание редактора: Где вы нашли этот компилятор?

5.17

Q: Почему не определены некоторые подпрограммы из стандартной ANSI-библиотеки, хотя у меня ANSI совместимый компилятор?

A: Нет ничего необычного в том, что компилятор, воспринимающий ANSI синтаксис, не имеет ANSI-совместимых головных файлов или стандартных библиотек. См. также вопросы 5.16 и 17.2.

5.18

Q: Почему компилятор "Frobozz Magic C", о котором говорится, что он ANSI-совместимый, не транслирует мою программу? Я знаю, что текст подчиняется стандарту ANSI, потому что он транслируется компилятором gcc.

A: Практически все компиляторы (а gcc - более других) поддерживают некоторые нестандартные расширения. Уверены ли Вы, что отвергнутый текст не применяет одно из таких расширений? Опасно экспериментировать с компилятором для исследования языка. Стандарт может допускать отклонения, а компилятор - работать неверно. См. также вопрос 4.4.

5.19

Q: Почему мне не удаются арифметические операции с указателем типа void * ?

A: Потому что компилятору не известен размер объекта, на который указывает void *. Перед арифметическими операциями используйте оператор приведения к типу (char *) или к тому типу, с которым собираетесь работать. (Смотрите, однако, вопрос 2.18).

5.20

Q: Правильна ли запись a[3]="abc" ? Что это значит?

A: Эта запись верна в ANSI C (и, возможно, в некоторых более ранних компиляторах), хотя полезность такой записи сомнительна. Объявляется массив размера три, инициализируемый тремя буквами 'a','b',и 'c' без завершающего стринг символа '\0'; Массив, следовательно, не может использоваться как стринг функциями strcpy, printf %s и т.п.

Смотри: ANSI Разд. 3.5.7 с. 72-3.

Примечание редактора: Используйте явную инициализацию без указания размера массива char a[] = "abc" и перестаньте считать длину строки на пальцах. Размер массива будет вычислен автоматически, исходя из рамера инициализатора. В данном случае - 4 char, '\0' хранится последним элементом массива, никаких проблем с функциями.

5.21

Q: Что такое #pragma и где это может пригодиться?

A: Директива #pragma обеспечивает особую, точно определенную "лазейку" для выполнения зависящих от реализации действий: контроль за листингом, упаковку структур, подавление предупреждающих сообщений (вроде комментариев /* NOTREACHED */ старой программы lint) и т.п.

Смотри: ANSI Разд. 3.8.6 .

5.22

Q: Что означает #pragma once? Я нашел эту директиву в одном из головных файлов.

A: Это расширение, реализованное в некоторых препроцессорах, делает головной файл идемпотентным, т.е. эффект от однократного включения файла равен эффекту от многократного включения. Эта директива приводит к тому же результату, что и прием с использованием #ifndef, описанный в вопросе 6.4.

5.23

Q: Вроде бы существует различие между зависимым от реализации, неописанным (unspecified) и неопределенным (undefined) поведением. В чем эта разница?

A: Если говорить кратко, то при зависимом от реализации поведении необходимо выбрать один вариант и документировать его. При неописанном поведении также выбирается один из вариантов, но в этом случае нет необходимости это документировать. Неопределенное поведение означает, что может произойти все что угодно. Ни в одном из этих случаев Стандарт не выдвигает требований; в первых

двух случаях Стандарт иногда предлагает (а может и требовать) выбор из нескольких близких вариантов поведения. Если Вы заинтересованы в написании мобильных программ, можете игнорировать различия между этими тремя случаями, поскольку всех их необходимо будет избегать.

Смотри: ANSI Разд.1.6, особенно "Rationale".

-- TatyanaDerbysheva - 06 Jan 2011