[16. Дополнительные приемы программирования.](#_Toc515770355)

[16.1. Совместимость типов.](#_Toc515770356)

[16.1.1. Определение совместимости типов в ANSI С.](#_Toc515770357)

[16.2. Идентичный тип.](#_Toc515770358)

[16.2.1. Перечисляемые типы.](#_Toc515770359)

[16.2.2. Типы массивов.](#_Toc515770360)

[16.2.3. Типы функций.](#_Toc515770361)

[16.2.4. Типы структур и объединений.](#_Toc515770362)

[16.2.5. Типы указателей.](#_Toc515770363)

[16.2.6. Совместимость нескольких исходных файлов.](#_Toc515770364)

[16.3. Макроопределения.](#_Toc515770365)

[16.3.1. Определение макросов.](#_Toc515770366)

[16.3.2. Макросы и параметры.](#_Toc515770367)

[16.3.3. Сложности при раскрытии макросов.](#_Toc515770368)

[16.3.4. Создание и использование собственных макросов.](#_Toc515770369)

[16.3.5. Макросы, поставляемые вместе с компилятором.](#_Toc515770370)

[16.3.6. Выбор макроопределения или функции.](#_Toc515770371)

[16.4. Директивы препроцессора.](#_Toc515770372)

[16.4.1. Директива #define.](#_Toc515770373)

[16.4.2. Директива #include.](#_Toc515770374)

[16.4.2.1. Заголовочные файлы.](#_Toc515770375)

[16.4.2.2. Правильное использование заголовочных файлов.](#_Toc515770376)

[16.4.2.3. Более эффективное использование заголовочных файлов.](#_Toc515770377)

[16.4.2.4. Новый стиль заголовков](#_Toc515770378)

[16.4.2.5. Предварительная компиляция заголовочных файлов.](#_Toc515770379)

[16.4.2.6. Файлы limits.h и float.h.](#_Toc515770380)

[16.4.3. Директивы #ifdef и #endif.](#_Toc515770381)

[16.4.4. Директива #undef.](#_Toc515770382)

[16.4.5. Директива #ifndef.](#_Toc515770383)

[16.4.6. Директива #if.](#_Toc515770384)

[16.4.7. Директива #else.](#_Toc515770385)

[16.4.8. Директива #elif.](#_Toc515770386)

[16.4.9. Директива #line.](#_Toc515770387)

[16.4.10. Директива #error.](#_Toc515770388)

[16.4.11. Директива #pragma.](#_Toc515770389)

[16.5. Оператор defined.](#_Toc515770390)

[16.6. Условная компиляция.](#_Toc515770391)

[16.7. Дополнительные операции препроцессора.](#_Toc515770392)

[16.7.1. Операция подстановки строки (#).](#_Toc515770393)

[16.7.2. Операция конкатенации (##).](#_Toc515770394)

[16.7.3. Операция подстановки символа (#@).](#_Toc515770395)

[16.8. Обработка ошибок: perror().](#_Toc515770396)

[16.9. Модели памяти.](#_Toc515770397)

[16.9.1. Модель tiny.](#_Toc515770398)

[16.9.2. Модель small.](#_Toc515770399)

[16.9.3. Модель medium.](#_Toc515770400)

[16.9.4. Модель compact.](#_Toc515770401)

[16.9.5. Модель large.](#_Toc515770402)

[16.9.6. Модель huge.](#_Toc515770403)

[16.10. Модификаторы функций.](#_Toc515770404)

[16.11. Модификаторы cdecl и pascal.](#_Toc515770405)

[16.12. Динамическое распределение памяти. Связанные списки.](#_Toc515770406)

[16.12.1. Особенности использования связанных списков.](#_Toc515770407)

[16.12.2. Простой связанный список.](#_Toc515770408)

1. Дополнительные приемы программирования.

Здесь рассматриваются дополнительные приемы программирования, общие для С и C++. Многие обсуждаемые темы, такие как совместимость типов и макросы, затрагивают такие средства языка, которые при разработке алгоритма программы нужно использовать с осторожностью. Другие описываемые средства, например поставляемые с компилятором макросы и условные команды препроцессора, помогают создавать более рациональные приложения. В конце темы описана концепция и синтаксические конструкции, необходимые для создания динамически связанных списков. После знакомства с нюансами программирования вы будете иметь достаточную подготовку по C/C++, чтобы без затруднений перейти к принципам объектно-ориентированного программирования.

Язык Си был разработан в помощь работающим программистам, а им нравится его препроцессор. Этот полезный помощник Он просматривает программу до компилятора (отсюда и термин «препроцессор») и заменяет символические аббревиатуры в программе на соответствующие директивы. Он отыскивает другие файлы, необходимые вам, и может также изменить условия компиляции. Однако эти слова еще не отражают истинную пользу и значение препроцессора, поэтому обратимся к примерам. Конечно, до сих пор мы снабжали все примеры директивами #define и #include, но теперь мы можем подытожить все, что изучили, и развить тему дальше.

## Совместимость типов.

Как вы уже хорошо знаете, С — слабо типизированный язык, а C++ имеет чуть большую типизацию (например, перечисляемые типы). Вы видели, как язык С может выполнять автоматические и явные преобразования типов, используя оператор явного приведения типа. В следующем разделе освещается не всегда очевидный способ интерпретации компилятором C/C++ совместимых типов.

### Определение совместимости типов в ANSI С.

Сама идея совместимых типов возникла в комитете ANSI С. Благодаря многим рекомендациям комитета в С появились средства, например прототипы функций, упрощающие поддержку языка. Комитет пытался определить набор правил или синтаксических конструкций, фиксирующих автоматические и неявные возможности языка.

Комитет ANSI С установил, что для того, чтобы два типа были совместимыми, они должны либо быть одного и того же типа, либо быть указателями, функциями или массивами, обладающими некоторыми свойствами; эти требования раскрыты в следующих параграфах.

## Идентичный тип.

Понятие "составной тип" связано с темой совместимости. Составной тип — это общий тип, образованный двумя совместимыми типами. Два любых совпадающих типа являются совместимыми, и их составной тип принадлежит к тому же типу.

Два арифметических типа идентичны, если они — одного типа. Краткие объявления одного и того же типа также идентичны. В следующем примере обе переменные, shivalue1 и shivalue2, имеют идентичный тип:

short shivalue 1;

short int shivalue2;

В следующем примере тип int совпадает с типом unsigned int:

int sivalue1;

unsigned int sivalue2;

Однако, типы int, short и unsigned отличаются друг от друга; при работе с символьными данными большое различие существует между типами char, signed char и unsigned char.

Как постановил комитет ANSI С, любой тип, которому предшествует какой-либо модификатор доступа, порождает несовместимые типы. Например, следующие два объявления относятся к несовместимым типам:

int ivalue1;

const int ivalue2;

Попробуйте определить, какие типы являются совместимыми в следующих объявлениях:

char \*pc1, \* рс2;

struct {int ix, iy;} stanonymous\_coord1, stanonymous\_coord2;

struct stxy {int ix, iy;} stanycoords;

typedef struct stxy STXY;

STXY stmorecoords;

Переменные pc1 и рс2 являются совместимыми указателями на символы, поскольку при объявлении дополнительный пробел между символом \* и именем рс2 является ненужным (незначащим).

Вас, наверное, не удивит то, что переменные stanonymous\_coord1 и stanonymous\_coord2 принадлежат к одному типу. Однако, тип переменной stanycoords не является идентичным типу предыдущих двух переменных. Хотя все три переменные содержат два одинаковых целочисленных поля, переменные stanonymous\_coord1 и stanonymous\_coord2 имеют анонимный структурный тип, а переменная stanycoords имеет теговый тип stxy.

Благодаря объявлению typedef, для компилятора типы struct stxy и STXY — идентичны, и поэтому переменная stanycoords идентична stmorecoords.

Важно помнить то, что компилятор рассматривает объявления typedef как синонимы типов, но не как совершенно новые типы. В следующем операторе определяется новый тип MYFLOAT, аналогичный типу float:

typedef float MYFLOAT;

### Перечисляемые типы.

Первоначально комитет ANSI С установил, что каждый перечисляемый тип совместим с зависящим от реализации интегральным типом; однако, в C++ перечисляемые типы с интегральными несовместимы. В С и C++ никакие определения перечисляемых типов, находящиеся в одном исходном файле, не являются совместимыми. Это правило аналогично тому, как отличаются структуры с тегами и структуры без тегов (анонимные). Вот почему в следующем примере типы ebflag1 и ebflag2 совместимы, а eflag1 — нет:

enum boolean {0,1} ebflag1;

enum {0,1} eflag1;

enum boolean ebflag2;

### Типы массивов.

Если два массива содержат элементы совместимых типов, то сами массивы считаются совместимыми. Если размер указан только для одного массива или же не указан совсем, то совместимость типов остается. Однако в случае, когда размеры указаны для обоих массивов, для того, чтобы они были совместимыми, размеры должны совпадать. Найдите все совместимые массивы в следующих объявлениях:

int imax20[20];

const int cimax20[20];

int imax10[10];

int iundefined[];

Безразмерный массив iundefined совместим с обоими массивами imax20 и imax10. Однако, два этих последних массива несовместимы, так как имеют различные размеры. Несовместимость массивов imax20 (элементы типа int) и cimax20 (элементы тип const int) объясняется тем, что несовместимы их элементы. Если у двух массивов указан размер, то составной тип совместимых массивов будет иметь тот же размер. В приведенном примере составной тип массивов iundefined и imax.20 — int[20].

### Типы функций.

Для того чтобы считать совместимыми две функции, имеющие прототипы, должны выполниться три условия. Две функции должны иметь один и тот же возвращаемый тип и одинаковое число параметров; при этом соответствующие параметры должны быть совместимых типов. Однако, имена параметров могут не совпадать.

### Типы структур и объединений.

При объявлении в программе новой структуры или объединения вводится новый тип, отличный (не совместимый) от любого другого типа в том же исходном файле. Поэтому в следующем примере переменные stanonymous1, stanonymous2 и stfloat1 отличаются друг от друга.

Однако же, ссылка на спецификатор типа, являющегося структурой, объединением или перечисляемым типом, принадлежит к тому же типу. Для связи ссылки и объявления типа используется поле тега, поэтому поле тега можно рассматривать как название типа. Вот почему переменные stfloat1 и stfloat2 имеют совместимые типы.

struct {float fvalue1, fvalue2;} stanonymous1;

struct {float fvalue1, fvalue2;} stanonymous2;

struct sttwofloats {float fvalue1, fvalue2;} stfloat1;

struct sttwofloats stfloat2;

### Типы указателей.

Два типа указателей считаются совместимыми, если они оба указывают на совместимые типы. Составной тип двух совместимых указателей такой же, как составной тип, на который осуществляется ссылка.

### Совместимость нескольких исходных файлов.

Поскольку каждое объявление структуры, объединения или перечисляемого типа рассматривается, как новый несовместимый тип, может возникнуть вопрос: что произойдет, если возникнет необходимость сослаться на эти типы из других файлов этой же программы.

Повторяющиеся объявления структур, объединений и перечисляемых типов совместимы в исходных файлах, если в них объявляются одни и те же элементы, в одном и том же порядке и элементы имеют совместимые типы. Для перечисляемых типов, однако, не требуется, чтобы константы перечисления объявлялись в том же порядке, хотя каждая константа должна иметь то же самое перечисляемое значение.

## Макроопределения.

Ранее было рассказано, как использовать директиву препроцессора #define для объявления символьных констант. Директиву препроцессора #define Ту же самую директиву можно использовать для определения макросов. Макрос — это фрагмент кода, который выглядит и работает так же, как функция.

Преимущество правильно написанного макроса заключается в скорости выполнения. Макрос раскрывается (заменяется его определением #define) во время работы препроцессора, при этом создается так называемый встраиваемый код. Именно поэтому макросы не вызывают дополнительных затрат времени, как это обычно случается при вызове функций. Однако, при каждой подстановке макроса увеличивается общий размер исполняемой программы.

Обратная ситуация — определения функций раскрываются только один раз и не имеет значения сколько раз они вызываются. Оптимальный выбор между скоростью выполнения и общим размером программы поможет вам решить, каким образом записывать конкретную процедуру.

Имеются и другие, более тонкие различия между макросами и функциями, касающиеся момента раскрытия кода. Их можно разбить на три категории. В С имя функции имеет значение адреса, по которому эта функция располагается. Поскольку макрос является встраиваемым средством и может раскрываться много раз, не существует единого адреса, связанного с ним. Поэтому макрос нельзя использовать в контексте, где требуется указатель на функцию. Кроме того, можно объявить указатель на функцию, но невозможно объявить указатель на макрос.

Компилятор С по-разному анализирует объявление функции и описание макроса #define, поэтому он не выполняет никаких проверок типов в макросе. В результате этого при передаче макросу неправильного количества аргументов или при ошибке в типе аргумента никаких сообщений от компилятора получено не будет.

Так как макросы раскрываются до начала фактической компиляции программы, некоторые макросы могут неправильно обрабатывать аргументы, если этот аргумент вычисляется в макросе несколько раз.

Макрос это фрагмент кода, который выглядит и работает так же, как функция. Однако функцией он не является. Имеется несколько различий между макросами и функциями:

* макрос заменяется своим определением во время работы препроцессора, то есть еще до компиляции программы. Поэтому макросы не вызывают дополнительных затрат времени, как при вызове функции
* использование макросов приводит к разрастанию исходного кода и увеличения размера исполняемой программы. Функции являются в некотором смысле антиподом макросов. Код, который они представляют, встраивается в программу только один раз, что приводит к сокращению кода программы. С другой стороны, при выполнении программы требуется дополнительное время для организации вызова функции;
* компилятор не выполняет никаких проверок типов в макросе. Поэтому при передаче макросу аргумента, не соответствующего подразумеваемому типу, или неверного числа аргументов, никаких сообщений об ошибке не будет.
* поскольку макрос является средством для встраивания фрагмента кода, не существует единого адреса, связанного с ним. Поэтому нельзя объявить указатель на макрос или использовать его адрес

Для определения макросов используется директива #define. Как и функции, макросы могут иметь параметры.

### Определение макросов.

Макросы определяются аналогично символьным константам. Единственное отличие в том, что строка подстановки substitution\_string обычно содержит несколько значений:

#define search\_string substitution\_string

Для того чтобы показать сходство описаний, в следующем примере директива препроцессора используется для определения символьной константы и макроса:

/\* #define — символьная константа \*/

#define iMAX\_ROWS 100

/\* #define — макрос \*/

#define NL putchar('\n')

Макрос NL заставляет препроцессор при просмотре исходного текста программы находить каждое вхождение символов NL и заменять его на оператор putchar(‘\n'). Следует обратить внимание на то, что в конце макроса отсутствует точка с запятой. Это объясняется тем, как макрос вызывается в исходном тексте программы:

int main(void)

{

…

…

…

NL;

Компилятор требует, чтобы вызов макроса заканчивался точкой с запятой. Предположим, что строка substitution\_string в макросе заканчивается точкой с запятой:

#define NL putchar('\n');

Тогда после раскрытия макроса в исходном тексте получится следующее:

int main(void)

{

…

…

…

putchar('\n'); ;

Директива #define, подобно всем директивам препроцессора, начинается с символа # в самой левой позиции. Она может появиться в любом месте исходного файла, а даваемое ею определение имеет силу от места появления до конца файла. Мы активно используем эту директиву для определения символических констант в наших программах, однако она имеет более широкое применение, что мы и покажем дальше. Вот пример, иллюстрирующий некоторые возможности и свойства директивы #define:

/\* простые примеры директивы препроцессора \*/

#include<stdio.h>

#define TWO 2 /\* по желанию можно использовать комментарии \*/

#define MSG "An old grey cat is singing a cheerful song"

/\*Старый серый кот поет веселую песню.\*/

/\* обратная косая черта продолжает определение на следующую строку \*/

#define FOUR TWO\*TWO

#define PX printf("X = %d.\n", x)

#define FMT "X = %d.\n"

void main()

{

int x = TWO;

PX;

x = FOUR;

printf(FMT, x);

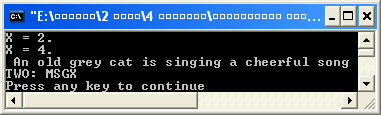
printf(" %s\n", MSG);

printf("TWO: MSGX\n");

}

Каждая строка состоит из трех частей. Первой стоит директива #define. Далее идет выбранная нами аббревиатура, известная у программистов как «макроопределение». Макроопределение не должно содержать внутри себя пробелы. И, наконец, идет строка (называемая «строкой замещения»), которую представляет макроопределение. Когда препроцессор находит в программе одно из ваших макроопределений, он почти всегда заменяет его строкой замещения. (Есть одно исключение, которое мы вам сейчас покажем.) Этот процесс прохождения от макроопределения до заключительной строки замещения называется «макрорасширением». Заметим, что при стандартной форме записи на языке Си можно вставлять комментарии; они будут игнорироваться препроцессором. Кроме того, большинство систем разрешает использовать обратную косую черту (' \' ) для расширения определения более чем на одну строку.

«Запустим» наш пример, и посмотрим за его выполнением.



Вот что произошло. Оператор

int x = TWO; превращается в int x = 2;

т. е. слово TWO заменилось цифрой 2. Затем оператор

РХ; превращается в printf("X равно %d.\n" , х);

поскольку сделана полная замена. Это новшество, так как до сих пор мы использовали макроопределения только для представления констант. Теперь же мы видим, что макроопределение может представлять любую строку, даже целое выражение на языке Си. Заметим, однако, что это константная строка; РХ напечатает только переменную, названную X.

12_01

Следующая строка также представляет что-то новое. Вы можете подумать, что FOUR заменяется на 4, но на самом деле выполняется следующее:

х = FOUR; превращается в х — TWO\*TWO; превращается в х = 2\*2;

и на этом все заканчивается. Фактическое умножение имеет место не во время работы препроцессора и не при компиляции, а всегда без исключения при работе программы. Препроцессор не выполняет вычислений; он только очень точно делает предложенные подстановки.

Заметим, что макроопределение может включать другие определения. (Некоторые компиляторы не поддерживают это свойство «вложения».)

В следующей строке

printf(FMT, х); превращается в printf("X равно %d.\n" , х)

когда FMT заменяется соответствующей строкой. Этот подход может оказаться очень удобным, если у вас есть длинная строка, которую вы используете несколько раз.

В следующей строке программы MSG заменяется соответствующей строкой. Кавычки делают замещающую строку константой символьной строки; поскольку программа получает ее содержимое, эта строка будет запоминаться в массиве, заканчивающемся нуль-символом. Так,

#define HAL 'Z' определяет символьную константу, а #define HAP "Z" определяет символьную строку: Z\0

Обычно препроцессор, встречая одно из ваших макроопределений в программе, очень точно заменяет их эквивалентной строкой замещения. Если эта строка также содержит макроопределения, они тоже замещаются. Единственным исключением при замене является макроопределение, находящееся внутри двойных кавычек. Поэтому

printf("TWO: MSG");

печатает буквально TWO: MSG вместо печати следующей строки:

2: "Старый серый кот поет веселую песню."

Если вам нужно напечатать эту строку, можно использовать оператор

printf(" %d: %s\n" , TWO, MSG);

потому что здесь макроопределения находятся вне кавычек.

Когда следует использовать символические константы? Вероятно, вы должны применять их для большинства чисел. Если число является константой, используемой в вычислениях, то символическое имя делает яснее ее смысл. Если число — размер массива, то символическое имя упрощает изменение вашей программы при работе с большим массивом. Если число является системным кодом, скажем для символа EOF, то символическое представление делает вашу программу более переносимой; изменяется только определение EOF. Мнемоническое значение, легкость изменения, переносимость: все это делает символические константы заслуживающими внимания.

Легко ли этого достичь? Рискнем и рассмотрим простую функцию, т. е. макроопределение с аргументами.

### Макросы и параметры.

В языке С макросы могут иметь аргументы. Такие макросы необходимо определять вместе с параметрами, назначение которых аналогично аргументам функции: они определяют места подстановки фактических параметров. В следующем примере показано, как определить и использовать макрос с параметрами:

/\* макроопределения \*/

#define READ\_RESPONSE(с) scanf("%c",(&c))

#define MULTIPLY(х,у) ((х)\*(у))

int main(void)

{

char cresponse;

int a = 10, b = 20;

…

…

…

READ\_RESPONSE(cresponse); /\* раскрытие макроса \*/

printf("%d",MULTIPLY(a,b));

В данном примере переменные х, у и с указывают места подстановки значений а, b и cresponse, соответственно. Два макроса, READ\_RESPONSE and MULTIPLY, иллюстрируют различные способы вызова макросов в программе. К примеру, MULTIPLY подставляется в оператор printf(), а READ\_RESPONSE используется самостоятельно.

Макроопределение с аргументами очень похоже на функцию, поскольку аргументы его заключены в скобки. Ниже приведено несколько примеров, иллюстрирующих, как определяется и используется такая «макрофункция». В некоторых примерах к тому же указаны возможные ловушки, поэтому читайте их внимательно.

/\* макроопределение с аргументами \*/

#include<stdio.h>

#define SQUARE(x) x\*x

#define PR(x) printf("x = %d.\n" , x)

void main()

{

int x = 4;

int z;

z = SQUARE(x);

PR(z);

z = SQUARE(2);

PR(z);

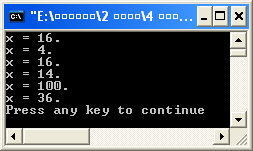
PR(SQUARE(x));

PR(SQUARE(x + 2));

PR(100/SQUARE(2));

PR(SQUARE(++ x));

}



z = SQUARE(2); // 16=4\*4

PR(z);

PR(SQUARE(x)); // 4=2\*2

PR(SQUARE(x + 2)); // 16=4\*4

PR(100/SQUARE(2)); // 100=100/2\*2

PR(SQUARE(++ x)); // 30=++x{5}\*++x{6}

Всюду, где в вашей программе появляется макроопределение SQUARE(x), оно заменяется на х\*х. В отличие от наших прежних примеров при использовании этого макроопределения мы можем совершенно свободно применять символы, отличные от х. В макроопределении 'х' замещается символом, использованным в макровызове программы. Поэтому макроопределение SQUARE(2) замещается на 2\*2. Таким образом, X на самом деле действует как аргумент. Однако, как вы вскоре увидите, аргумент макроопределения не «работает» точно так же, как аргумент функции. Вот результаты выполнения программы. Обратите внимание, что некоторые ответы отличаются от тех, которые вы могли бы ожидать.

z равно 16.

z равно 4.

SQUARE(x) равно 16.

SQUARE(x + 2) равно 14.

100/SQUARE(2) равно 100.

SQUARE(++ x) равно 30.

Первые две строки предсказуемы. Заметим, однако, что даже внутри двойных кавычек в определении PR переменная замещается соответствующим аргументом. ВСЕ аргументы в этом определении замещаются.

Третья строка представляет интерес:

PR(SQUARE(x));

она становится следующей строкой:

printf(" SQUARE(x) равно %d.\n" , SQUARE(x));

после первого этапа макрорасширения. Второе SQUARE(x) расширяется, превращаясь в х\*х, а первое остается без изменения, потому что теперь оно находится внутри двойных кавычек в операторе программы, и таким образом защищено от дальнейшего расширения. Окончательно строка программы содержит

printf(" SQUARE(x) равно %d.\n" , х\*х);

и выводит на печать

SQUARE(x) равно 16.

при работе программы.

Давайте еще раз проверим то, что заключено в двойные кавычки. Если ваше макроопределение включает аргумент с двойными кавычками, то аргумент будет замещаться строкой из макровызова. Но после этого он в дальнейшем не расширяется, даже если строка является еще одним макроопределением. В нашем примере переменная х стала макроопределением SQUARE(x) и осталась им.

Теперь мы добрались до несколько специфических результатов, Вспомним, что x имеет значение 4. Это позволяет предположить, это SQUARE(x + 2) будет равно 6\*6 или 36. Но напечатанный результат говорит, что получается число 14, которое, несомненно, никак не похоже на квадрат целого числа! Причина такого вводящего в заблуждение результата проста, и мы уже об этом говорили: препроцессор не делает вычислений, он только замещает строку. Всюду, где наше определение указывает на х, препроцессор подставит строку х + 2. Таким образом,

х\*х становится х + 2\*х + 2.

Единственное умножение здесь 2\*х. Если х равно 4, то получается следующее значение этого выражения:

4 + 2\*4 + 2 = 4 + 8 + 2 = 14.

Этот пример точно показывает очень важное отличие между вызовом функции и макровызовом. Вызов функции передает значение аргумента в функцию во время выполнения программы. Макровызов передает строку аргументов в программу до ее компиляции; это другой процесс, происходящий в другое время.

Можно ли ваше определение переделать так, чтобы SQUARE(x + 2) было равно 36? Конечно. Нам просто нужно больше скобок:

#define SQUARE(x) (х)\*(х)

Тогда SQUARE(x + 2) становится (х + 2)\*(х + 2), и мы получаем наше желанное умножение, так как перенесли скобки в строку замещения.

Однако это не решает всех наших проблем. Рассмотрим случаи, которые приводят к следующей строке на выходе:

100/SQUARE(2) превращается в 100/2\*2

Вычисления следует вести слева направо, т. е.

(100/2)\*2 или 50\*2 или 100.

Эту путаницу можно исправить, определив SQUARE(x) следующим образом:

#define SQUARE(x) (x\*x)

Это даст

100/(2\*2), что, в конечном счете, эквивалентно 100/4 или 25.

Чтобы выполнить два последних примера, нам необходимо определение

#define SQUARE(x) ((x)\*(x))

Это урок использования необходимого количества скобок для гарантии, что операции и соединения выполняются в правильном порядке.

Даже эти предосторожности не спасают последний пример от беды:

SQUARE(++ x) превращается в ++х\* ++х

и х увеличивается дважды — один раз до умножения и один раз после:

++х\* ++х = 5\*6 = 30

(Так как порядок выполнения операций не установлен, то некоторые компиляторы превратят это в 6\*5, но конечный результат будет тем же самым.)

Единственное лекарство в этом случае — не использовать ++х в качестве аргумента для макроопределения. Заметим, что ++х обычно работает как аргумент функции, так как ему присваивается значение 5, и затем это значение 5 передается функции.

Внешне использование макроса похоже на использование функции (из-за чего их иногда называют псевдофункциями). Поскольку, как отмечено выше, это все-таки разные объекты, принято имена макросов записывать прописными буквами.

По этой причине в качестве общего правила рекомендуется всегда заключать в скобки каждый параметр макроса. Кроме того, если вызов макроса может появляться в инструкциях, содержащих операторы приведения типа, например,

doubleValue = (double) SQUARE (x + 1);

то рекомендуется заключать в скобки все тело макроса:

#define SQUARE (x) ((х) \* (х))

И последнее замечание по поводу использования макросов. Обычно в тексте программы дополнительные пробелы не являются значащими. Они служат только для удобства чтения кода. В случае с макросами это не так. Например, пробел между именем макроса и его параметрами может привести к изменению смысла макроса, как в следующем макроопределении:

#define BAD\_MACRO (x) printf ("%d", x)

Оно будет раскрыто в следующую инструкцию:

(х) printf("%d", x);

Это приведёт к ошибке компиляции. Чтобы исправить ошибку, достаточно удалить лишний пробел между именем макроса и его параметром.

#define BAD\_MACRO(x) printf("%d", x)

Если определение макроса не умещается на одной строке, оно может быть продолжено в последующих строках. Для продолжения макроопределения в следующей строке достаточно в конце текущей строки поставить знак '\'.

Например,

#define MAKEWORD(a, b) ((WORD) (((BYTE) (a)) \

|(((WORD)((BYTE)(b))) << 8)))

В определении макроса могут участвовать другие макросы, то есть макросы могут быть вложенными. Следующий пример демонстрирует использование вложенных макросов:

#define PI 3.14159

#define SQUARE(x) ((x)\*(x))

#define CIRCLE\_AREA(x) (PI \* SQUARE(x))

Макрос может быть аннулирован в любом месте программы с помощью директивы #undef. Например,

#include <iostream>

using namespace std;

//Макроопределение

#define MULTIPLY(x, у) ((x)\*(у))

int main()

{

setlocale(LC\_ALL, "Rus");

int a = 2, b = 3;

printf("%d", MULTIPLY(a, b));

#undef MULTIPLY(x, у)

// Не закоментированный printf приведет к ошибке

//printf("%d", MULTIPLY(a, b));

getchar(); getchar();

return 0;

}



### Сложности при раскрытии макросов.

При использовании макросов одни символы или лексемы буквально заменяются на другие. Фактический синтаксический анализ объявления макроса, выражения внутри него и оператора, где он вызывается, выполняется после процесса раскрытия макроса. Иногда это может привести к неожиданным результатам. Например: следующее макроопределение кажется совершенно правильным:

#define SQUAREIT(x) x \* х

Предположим, что оператор вызывает его со значением, равным 5:

iresult = SQUAREIT(5) ;

Компилятор получает следующий оператор:

iresult =5 \* 5;

На первый взгляд все выглядит правильно. Предположим, однако, что этот же самый макрос вызывается в следующем операторе:

iresult = SQUAREIT(x + 1);

Вместо

iresult = (х + 1) \* (х + 1);

компилятор получит следующее:

iresult = х + (1 \* х) + 1);

В качестве общего правила можно посоветовать всегда заключать в скобки каждый параметр, появляющийся в теле макроопределения, как это было показано при описании макросов READ\_RESPONSE и MULTIPLY. Кроме того, если макровызов может выполняться в операции явного приведения типа, например в

dresult = (double)SQUAREIT(x + 1);

то лучше всего заключать в скобки все тело макроса:

#define SQUAREIT(x) ((x) \* (х))

Для большинства современных компиляторов С дополнительные пробелы внутри стандартных операторов С не являются значащими. Это не относится к макроопределениям. Посмотрите внимательно на следующий пример и найдите ошибку:

/\* неправильное макроопределение \*/

#define BAD\_MACRO (ans) scanf("%d",(&ans))

Напомним, что при выполнении директивы #define препроцессор ищет строку поиска и заменяет ее строкой подстановки. Эти две строки разделены несколькими пробелами. При раскрытии приведенного описания компилятор получит следующее:

(ans) scanf("%d",(&ans));

Создан неправильный оператор. Проблема заключается в пробеле между именем макроса BAD\_MACRO и (ans). Из-за этого пробела список параметров стал частью строки подстановки, а должен был быть на соответствующем месте в строке поиска. Для того чтобы исправить определение BAD\_MACRO, нужно удалить лишний пробел:

#define BAD\_MACRO(ans) scanf("%d",(&ans))

Для того чтобы определить степень вашего понимания проблем, встречающихся при использовании макросов, попробуйте предсказать результат следующих операторов:

int х = 5;

iresult = SQUAREIT(x++);

При использовании некоторых операций С, подобных инкременту (++) и декременту (--), ситуация становится не всегда предсказуемой. Результат приведенного выражения может оказаться равным 30, а не 25, как ожидалось; это вызвано тем, что различные реализации компиляторов С могут вычислять данное выражение различными способами. Например, при раскрытии синтаксиса макрос может выглядеть так:

/\* iresult = х \* х; \*/

iresult = 5\*5;

или так

/\* iresult = х \* (х+1); \*/

iresult =5\*6;

### Создание и использование собственных макросов.

При описании макросов могут использоваться другие макроопределения. Эта возможность может использоваться для улучшения читаемости текста программы. Для примера можно рассмотреть следующие последовательные макроопределения:

#define NL putchar('\n')

#define TAB putchar('\t')

#define F0RMAT1 NL, NL, TAB

#define F0RMAT2 NL, TAB, TAB

#define BEGIN\_PROMPT FORMAT1, printf("Want to begin?"); \

printf("\nType 1 for yes, 0 for no")

#define READ\_RESPONSE F0RMAT2,scanf("%d", (&c) )

#define FORMAT\_PRINT(ccontrol,ivalue, fvalue) \

printf("\n%c\t%d\t%8.2f", (ccontrol), (ivalue), (fvalue))

Вместо того, чтобы включать в основную программу все операции, определенные в макросах, достаточно написать следующее:

int main(void)

{

char cresponse;

int ivalue = 23;

float fvalue = 56.78;

…

…

…

BEGIN\_PROMPT;

READ\_RESPONSE(cresponse);

FORMAT\_PRINT(cresponse,ivalue,fvalue);

Читаемость текста программы улучшается, но нужно, однако, помнить о том, что при этом отключается автоматическая проверка типов, выполняемая компилятором, и возможны побочные эффекты, вызванные ошибками в синтаксисе вызывающих операторов.

### Макросы, поставляемые вместе с компилятором.

Комитет ANSI С рекомендовал, чтобы во всех компиляторах С были описаны пять специальных макросов без параметров. Название каждого макроса начинается и заканчивается двумя символами подчеркивания; это показано в следующей таблице:

**Таблица. Предопределенные макросы ANSI**

|  |  |
| --- | --- |
| **Макрос** | **Описание** |
| \_\_DATE\_\_ | Строка, представляющая в форме mmm.dd.yyyy дату создания данного файла |
| \_\_FILE\_\_ | Имя текущего обрабатываемого файла |
| \_\_LINE\_\_ | Номер текущей строки обрабатываемого файла |
| \_\_STDC\_\_ | Определен, если установлен режим совместимости с ANSI С |
| \_\_TIME\_\_ | Время начала обработки текущего файла в формате hh.mm.ss |
| \_\_TIMESTAMP\_\_ | Строковая константа, представляющая собой дату и время последней модификации исходного файла в виде Ddd Mmm hh:mm:ss уууу (день месяц часы:минуты:секунды год) |

Предопределенные макроопределения вызываются аналогично тем, которые описаны пользователем. К примеру, для печати на экране имени программы, даты и текущего номера строки можно использовать следующий оператор:

printf("%s I %s I Line number: %d", \_\_FILE\_\_ , \_\_DATE\_\_ , \_\_LINE\_\_);

Следующий пример демонстрирует использование этих макросов:

#include <iostream>

using namespace std;

char\* Date = \_\_DATE\_\_;

char\* Time = \_\_TIME\_\_;

int main()

{

setlocale(LC\_ALL, "Rus");

FILE\* file;

//…

printf("Дата создания %s, "

"время создания %s\n", Date, Time);

//…

file = fopen("Ex.cpp","r+t");

if (file = NULL){

printf("Ошибка в вызове функции fopen() "

"в файле %s, строка %d\n",

\_\_FILE\_\_, \_\_LINE\_\_);

getchar(); getchar();

return 1;

}

else{

printf("Вызов функции fopen() выполнен корректно "

"в файле %s, строка %d\n",

\_\_FILE\_\_, \_\_LINE\_\_);

getchar(); getchar();

return 0;

}

//…

return 0;

}



### Выбор макроопределения или функции.

Многие задачи можно решать, используя макроопределение с аргументами или функцию. Что из них следует применять нам? На этот счет нет строгих правил, но есть некоторые соображения.

Макроопределения должны использоваться скорее как хитрости, а не как обычные функции: они могут иметь нежелательные побочные эффекты, если вы будете неосторожны. Некоторые компиляторы ограничивают макроопределения одной строкой, и, по-видимому, лучше соблюдать такое ограничение, даже если ваш компилятор этого не делает.

Выбор макроопределения приводит к увеличению объема памяти, а выбор функции — к увеличению времени работы программы. Так что думайте, что выбрать! Макроопределение создает «строчный» код, т. е. вы получаете оператор в программе. Если макроопределение применить 20 раз, то в вашу программу вставится 20 Строк кода. Если вы используете функцию 20 раз, у вас будет только одна копия операторов функции, поэтому получается меньший объем памяти. Однако управление программой следует передать туда, где находится функция, а затем вернуться в вызывающую Программу, а на это потребуется больше времени, чем при работе со «строчными» кодами.

Преимущество макроопределений заключается в том, что при их использовании вам не нужно беспокоиться о типах переменных (макроопределения имеют дело с символьными строками, а не с фактическими значениями). Так наше макроопределение SQUARE(X) можно использовать одинаково хорошо с переменными типа int или float.

Обычно программисты используют макроопределения для простых действий, подобных следующим:

#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y))

#define ABS(X) ( (X) < 0 ? - (X) : (X))

#define ISSIGN(X) ((Х) == ' +' || (X) == ' ' ? 1 : 0

(Последнее макроопределение имеет значение 1 (истинно), если х является символом алгебраического знака.) Отметим следующие моменты:

* В макроопределении нет пробелов, но они могут появиться в замещающей строке. Препроцессор «полагает», что макроопределение заканчивается на первом пробеле, поэтому все, что стоит после пробела, остается в замещающей строке.

12_02

* Используйте круглые скобки для каждого аргумента и всего определения. Это является гарантией того, что элементы будут сгруппированы надлежащим образом в выражении, подобном

forks = 2\*MAX(guests + 3, last);

* Для имен макрофункций следует использовать прописные буквы. Это соглашение не распространяется так широко, как соглашение об использовании прописных букв для макроконстант. Применение их предостережет вас от возможных побочных эффектов макроопределений.

Предположим, что вы разработали несколько макрофункций по своему усмотрению. Должны ли вы их переопределять каждый раз, когда пишете новую программу? Нет, если вы вспомните о директиве #include. Теперь рассмотрим ее.

## Директивы препроцессора.

На практике имеется 12 команд препроцессора, называемых иногда директивами. Они перечислены в следующей таблице. Две директивы из 12 вам уже знакомы — #include и #define.

#include #define #ifdef #endif

#undef #ifndef #if #else

#elif #line #error #pragma

Напомним, что препроцессор С обрабатывает исходный файл перед тем, как компилятор транслирует программу в объектный код. Выбирая правильные директивы, можно создать более эффективные заголовочные файлы, решить некоторые проблемы программирования и предотвратить наложения объявлений при наличии множества файлов.

Эти директивы обычно используются при программировании больших модулей. Они позволяют приостановить действие более ранних определений и создать файлы, каждый из которых можно компилировать по-разному.

Директивы препроцессора представляют собой команды компилятору, которые позволяют управлять компиляцией программы и сделать ее код более понятным. Все директивы начинаются с символа #. Перед начальным символом # и вслед за ним могут располагаться пробелы. Директивы обрабатываются во время первой фазы компиляции специальной программой - препроцессором.

### Директива #define.

Директива #define позволяет определить некоторый идентификатор или связать его с некоторой последовательностью лексем. В последнем случае он служит для определения макроса. Макросы мы рассмотрим чуть позже, а здесь приведем пример использования директивы #define для определения некоторого идентификатора или символической константы:

#define \_cplusplus

После того, как символическая константа определена в данном файле, она может быть использована в других директивах препроцессора. Отменить определение символической константы можно с помощью директивы

#undef

Например:

#undef \_cplusplus

### Директива #include.

Когда препроцессор «распознает» директиву #include, он ищет следующее за ней имя файла и включает его в текущий файл. Директива имеет вид:

#include <header\_name>

#include "header name"

#include macro identifier

#include <stdio.h> имя файла в угловых скобках

#include "mystuff.h" имя файла в двойных кавычках

Здесь header\_name должно быть именем файла с расширением или именем заголовка в новом стиле. Традиционно используется расширение .h или .hpp. Кроме того, в header name может указываться путь к файлу. Препроцессор удаляет директиву #include из текста файла с исходным кодом и направляет содержимое указанного в ней файла для обработки компилятором. Если указан путь к файлу, компилятор ищет его в указанном каталоге. Первая и вторая форма синтаксиса различаются используемым компилятором алгоритмом поиска файла, если полный путь к нему не задан. Форма <header\_name>, в которой имя заголовочного файла задается в угловых скобках, указывает компилятору, что заголовочный файл следует искать в подкаталоге \include компилятора. Форма "header\_name", в которой имя заголовочного файла задается в двойных кавычках, указывает, что заголовочный файл следует искать в текущем каталоге, затем - в подкаталоге \include компилятора.

Первая и вторая версии не подразумевают никакого раскрытия макроса. В третьей версии предполагается, что существует макроопределение, которое раскрывается в допустимое имя заголовочного файла.

В операционной системе UNIX угловые скобки сообщают препроцессору, что файл следует искать в одном или нескольких стандартных системных каталогах. Кавычки говорят ему, что сначала нужно смотреть в вашем каталоге (или в каком-то другом, если вы определяете его именем файла), а затем искать в «стандартных» местах.

#include <stdio.h> ищет в системном каталоге

#include "hot.h"ищет в вашем текущем рабочем каталоге

#include "/usr/biff/p.h" ищет в каталоге /usr/biff

В типичной микропроцессорной системе эти две формы являются синонимами, и препроцессор ведет поиск на указанном диске.

#include "stdio.h" ищет на стандартном диске

#include < stdio.h> ищет на стандартном диске

#include "a:stdio.h" ищет на диске а

Зачем включают файлы? Потому что они несут нужную вам информацию. Файл stdio.h, например, обычно содержит определения EOF, getchar() и putchar(). Две последние функции определены как макрофункции.

По соглашению суффикс .h используется для «заголовочных» файлов, т. е. файлов с информацией, которая располагается в начале вашей программы. Заголовочные файлы обычно состоят из операторов препроцессора. Некоторые файлы, подобно stdio.h, включены в систему, но вы можете создать и свой собственный.

Заголовочные файлы.

Предположим, например, что вам нравится использовать булевы переменные, т. е. вместо того чтобы иметь 1 как «истину» и 0 как "ложь", хотели бы использовать слова TRUE и FALSE. Можно создать файл, названный, скажем, bool.h, который содержал бы эти определения:

/\* файл bool.h \*/

#define BOOL int;

#define TRUE 1

#define FALSE 0

Вот пример программы, использующей этот заголовок:

/\* считает пустые символы \*/

#include <stdio.h>

#include <BOOL.h>

main()

{

int ch;

int count = 0;

BOOL whitesp();

while ( (ch = getchar() ) != EOF)

if ( whitesp(ch) )

count++ ;

printf(" There is %d empty symbols\n", count);

/\*Имеется %d пустых символов\*/

}

BOOL whitesp(c)

char c;

{

if(c==' '||с==' \n'||с ==' \t')

return(TRUE);

else

return(FALSE);

}

**Замечания по программе.**

* Если две функции в этой программе ('main()' и 'whitesp()') следовало скомпилировать раздельно, то нужно было бы использовать директиву #include "bool.h" с каждой из них.
* Выражение if (whitesp(ch)) аналогично if (whitesp(ch)) == TRUE, так как сама функция whitesp(ch) имеет значение TRUE или FALSE.
* Мы не создали новый тип BOOL, так как BOOL уже имеет тип int. Цель наименования функции BOOL — напомнить пользователю, что функция используется для логических вычислений (в противоположность арифметическим).
* Использование функции в предусматриваемых логических сравнениях может сделать программу яснее. Она может также сэкономить наш труд, если сравнение встречается в программе более одного раза.
* Мы могли бы использовать макроопределение вместо функции для задания whitesp().

Многие программисты разрабатывают свои стандартные заголовочные файлы, чтобы использовать их в программах. Некоторые файлы могут создаваться для специальных целей, другие можно использовать почти в каждой программе. Так как включенные файлы могут содержать директивы #include, можно, если хотите, создать сжатый, хорошо организованный заголовочный файл. Рассмотрим этот пример:

/\* заголовочный файл mystuff.h\*/

#include < stdio.h>

#include "bool.h"

#include "funct.h"

#define YES 1

#define NO 0

Во-первых, мы хотим напомнить вам, что препроцессор языка Си распознает комментарии, помеченные /\* и \*/, поэтому мы можем включать комментарии в эти файлы.

Во-вторых, мы включили три файла. По-видимому, третий содержит некоторые макрофункции, которые используются часто.

В-третьих, мы определили YES как 1, тогда как в файле bool.h мы определили TRUE как 1. Но здесь нет конфликта, и мы можем использовать слова YES и TRUE в одной и той же программе. Каждое из них будет заменяться на 1.

Может возникнуть конфликт, если мы добавим в файл строку

#define TRUE 2

Второе определение вытеснило бы первое, и некоторые препроцессоры предупреждали бы вас, что TRUE переопределено.

Директива #include не ограничивается заголовочными файлами. Если вы записали нужную функцию в файл sort .с, то можно использовать

#include "sort.c"

чтобы скомпилировать его совместно с вашей текущей программой.

Директивы #include и #define являются наиболее активно используемыми средствами препроцессора языка Си. Рассмотрим более сжато другие его директивы.

Правильное использование заголовочных файлов.

Поскольку заголовочные файлы состоят из синтаксически правильных символьных (ASCII) текстов программ на C/C++ и могут включаться в другие файлы в тех местах, где использована директива #include, многие начинающие программисты используют их не по назначению. Иногда эти файлы некорректно используются для описания целых функций или наборов функций. Хотя при этом компилятор не выдает никаких сообщений, логическая структура программы нарушается.

Заголовочные файлы применяются для описания объявлений, используемых совместно несколькими исходными файлами; они централизованно хранят объявления всех внешних переменных, прототипы функций, определения классов, структур, объединений, перечисляемых типов и встраиваемых функций. Переменные, функции и классы должны объявляться в файлах при помощи заголовочных файлов и директив #include.

Таким образом, устраняются два источника ошибок. Во-первых, гарантируется, что все файлы содержат одни и те же объявления. Во-вторых, если некоторое объявление нужно обновить, то необходимо сделать только одну замену в заголовочном файле. Исчезает вероятность того, что данное объявление в каком-нибудь отдельном файле осталось не обновленным. Часто в заголовочные файлы входят следующие компоненты:

* Директивы препроцессора
* Объявления констант const
* Прототипы функций
* Определения typedef
* Определения структур
* Перечисляемые типы
* Внешние ссылки extern

При разработке заголовочных файлов нужно проявлять некоторую осторожность. Предусматриваемые объявления должны быть логически связаны друг с другом. На компиляцию заголовочного файла требуется время, и, если этот файл слишком большой или содержит очень много несвязанных элементов, то издержки в виде увеличения времени компиляции могут поставить под сомнение целесообразность его использования.

Второй важный момент — это то, что заголовочный файл никогда не должен содержать нестатических определений. Если два файла в одной программе будут включать некий заголовочный файл, имеющий внешнее определение, то большинство редакторов связей (компоновщиков) не станут обрабатывать программу из-за наличия многократно описанных символов (multiple defined symbols). Поскольку в заголовочных файлах часто применяются постоянные значения, то по умолчанию режим компоновки идентификаторов const — статический, и поэтому внутри этих файлах можно определять константы.

Заголовочные файлы обычно используются для объявления типов данных, констант, прототипов функций, определения структур и перечисленных типов, а также внешних ссылок используемых совместно несколькими файлами исходного кода. Подключение объявлений, содержащихся в заголовочном файле, к файлу исходного кода программы производится директивой #include. Тем самым гарантируется, что все файлы, к которым подключен заголовочный файл, содержат одинаковые объявления. С другой стороны, если нужно изменить некоторое объявление, это достаточно сделать в одном месте - в соответствующем заголовочном файле. Реально осуществляемые проекты могут быть достаточно сложны. В них бывает трудно отследить повторное подключение заголовочного файла, которое может произойти неявно. Для предотвращения многократных включений заголовочного файла используется стандартный прием, основанный на комбинации директив препроцессора. Рассмотрим пример:

#ifndef \_HEADERFILENAME\_

#define \_HEADERFILENAME\_

//текст заголовочного файла

#endif // \_HEADERFILENAME\_

В этом примере в начале заголовочного файла проверяется, не объявлена ли уже символическая константа \_HEADERFILENAME\_ и, если - нет (это означает, что данный заголовочный файл еще не подключен), то эта символическая константа объявляется. Затем идет остальной текст заголовочного файла. Последняя строка этого файла содержит директиву #endef, закрывающую объявления, начатые директивой #ifndef.

Более эффективное использование заголовочных файлов.

Эффективность компиляции заголовочных файлов можно повысить, если использовать комбинации директив препроцессора. Лучше всего это показать на примере:

#ifndef \_INC\_IOSTREAM

#define \_INC\_IOSTREAM

#if !defined(\_INC\_DEFS )

#include <\_defs.h>

#endif

#if !defined(\_INC\_MEM )

#include <mem.h> // для доступа к memcpy и NULL

#endif

#endif /\* !\_INC\_IOSTREAM \*/

Перед тем как рассматривать отдельные операторы в данном примере, необходимо сказать, что при первом проходе компилятор строит таблицу символов. В частности, элементами такой таблицы являются псевдоимена заголовочных файлов. Использование псевдоимен позволяет компилятору отличать один символ от другого. Транслятор С добавляет в начало этих символов подчеркивание.

При компиляции проще всего управлять видимостью файла заголовков, если заключить код внутри его в следующие комбинации из трех операторов:

#ifndef \_INC\_myheader

#define \_INC\_myheader /\* начало видимости \_INC\_MYHEADER \*/

...

...

...

#endif /\* окончание условной видимости \_INC\_MYHEADER \*/

Это же самое было сделано в предыдущем примере, где вместо \_INC\_MYHEADER использовался параметр \_INC\_IOSTREAM. Первый раз, когда компилятор включает данный заголовочный файл, параметр \_INC\_IOSTREAM не определен. Этот фрагмент кода включается, после чего становятся видимыми все вложенные операторы. Начиная с данного момента, любые дополнительные операторы #include <iostream.h>, находящиеся в других файлах, использующихся для создания исполняемого модуля, будут обходить вложенный код.

Новый стиль заголовков

Исторически так сложилось, что в языке C++ при подключении заголовочных файлов использовался тот же синтаксис, что и в языке С для совместимости с разработанным на тот момент программным обеспечением. Однако при стандартизации языка этот стиль был изменен и теперь вместо заголовочных файлов (как это было в С) указываются некоторые стандартные идентификаторы, по которым компилятор сам находит необходимые файлы. Предопределенные идентификаторы представляют собой имя заголовка в угловых скобках без указания расширения (.h). Ниже приводится пример включения заголовков в стиле C++:

#include <iostream>

#include <stdlib>

#include <new>

Помимо этого, для включения в программу библиотек функций языка С в соответствии с новым стандартом заголовок преобразуется следующим образом: отбрасывается расширение .h и к имени заголовка добавляется префикс с. Таким образом, например, заголовок <string.h> заменяется заголовком <cstring>. Если же используемый компилятор не поддерживает объявления заголовков в новом стиле, можно по-прежнему использовать заголовки в стиле языка С, хотя это и не рекомендуется стандартом C++.

Предварительная компиляция заголовочных файлов.

Правильное использование заголовочных файлов позволяет увеличить скорость компиляции программы; для этой же цели служат предварительно компилированные заголовочные файлы. Предварительная компиляция наиболее удобна тогда, когда оттранслированный и не меняющийся фрагмент программы используется вместе с другим фрагментом, находящимся в стадии проектирования.

Предварительно компилированный заголовочный файл (.РСН) получается при запуске компилятора с опцией /Yc. Синтаксис следующий:

/Ycyourfile

Пробелы между опцией /Yc и именем файла yourfile не допускаются. При наличии ключа /Yc компилятор транслирует весь исходный файл, включая все заголовочные файлы. Предварительно компилированный файл запоминается с именем yourfile, совпадающим с названием исходного файла, и расширением .РСН.

Для автоматического создания и использования предварительно компилированных заголовочных файлов достаточно выбрать меню Options|Compiler options и щелкнуть на опциях Precompiled Headers и Automatic Use of Precompiled Headers.

Файлы limits.h и float.h.

Для того, чтобы создаваемые программы были переносимыми, согласно требованиям комитета ANSI C все компиляторы С документируют системно-зависимые диапазоны представления целых чисел и чисел с плавающей точкой. В табл. 14.1 перечислены находящиеся в файле limits.h общие определения, отвечающие требованиям комитета ANSI C.

Таблица. Значения, определенные в файле limits.h (ANSI C)

|  |  |  |
| --- | --- | --- |
| **#define** | **Значение** | **Описание** |
| #define CHAR\_BIT | 8 | Количество разрядов в типе char |
| #define SCHAR\_MIN | (-127) | Минимальное значение signed char |
| #define SCHAR\_MAX | 127 | Максимальное значение signed char |
| #define UCHAR\_MAX | Oxff | Максимальное значение unsigned char |
| #define SHRT\_MIN | (-32767) | Минимальное значение (signed) short |
| #define SHRT\_MAX | 32767 | Максимальное значение (signed) short |
| #define USHRT\_MAX | Oxfff | Максимальное значение unsigned short |
| #define INT\_MIN | (-32767) | Минимальное значение (signed) int |
| #define INT\_MAX | 32767 | Максимальное значение (signed) int |
| #define UINT\_MAX | Oxfff | Максимальное значение unsigned int |
| #define LONG\_MIN | (-2147483647) | Минимальное значение (signed) long |
| #define LONG\_MAX | 2147483647 | Максимальное значение (signed) long |
| #define ULONG\_MAX | Oxffffffff | Максимальное значение unsigned long |
| #define CHAR\_MIN | SCHAR\_MIN | Минимальное значение char |

Эти определения можно использовать в программе для того, чтобы гарантировать согласованность диапазонов используемых данных и указанных типов. Например, целое число в системе VAX занимает 4 байта, а в РС-совместимом компьютере только 2. Одно из решений данной проблемы — следующее:

if (PROGRAM\_NEEDED\_MAX > INT\_MAX) // требуемое максимальное значение

pvoid = new llong\_storage; // значение long  
else

pvoid = new iinteger\_storage; // значение int

В следующей таблице перечислены определения чисел с плавающей точкой, отвечающие требованиям комитета ANSI С.

Таблица Значения, определенные в файле float.h (ANSI С)

|  |  |  |
| --- | --- | --- |
| **#define** | **Значение** | **Описание** |
| #define FLT\_RADIX | 2 | Основание показателя |
| #define FLT\_ROUNDS | 1 | Округление при сложении: до ближайшего |
| /\* минимальное значение, при котором 1.0+FLT\_EPSILON!=1.0\*/  #define FLT\_EPSILON 1.192092896e-07F  /\* минимальное значение, при котором 1.0+DBL\_EPSILON!=1.0\*/  #define DBL\_EPSILON 2.2204460492503131e-016  /\* минимальное значение, при котором 1.0+LDBL\_EPSILON!=1.0\*/  #define LDBL\_EPSILON 1.084202172485504434e-019L | | |
| #define FTL\_DIG | 6 | Точность: количество десятичных цифр |
| #define DBL\_DIG | 15 | Точность: количество десятичных цифр |
| #define LDBL\_DIG | 18 | Точность: количество десятичных цифр |
| #define FLT\_MIN | 1.175494351e-38F | Min положительное значение |
| #define DBL\_MIN | 2.2250738585072014e-308 | Min положительное значение |
| #define LDBL\_MIN | 3.3621031431120935063e-4932L | Min положительное значение |
| #define FLT\_MIN\_EXP | (-125) | Min показатель с основанием 2 |
| #define DBL\_MIN\_EXP | (-1021) | Min показатель с основанием 2 |
| #define LDBL\_MIN\_EXP | (-16381) | Min показатель с основанием 2 |
| #define FLT\_MIN\_10\_EXP | (-10) | Min показатель с основанием 10 |
| #define DBL\_MIN\_10\_EXP | (-307) | Min показатель с основанием 10 |
| #define LDBL\_MIN\_10\_EXP | (-4931) | Min показатель с основанием 10 |
| #define FLT\_MAX | 3.402823466e+38F | Max значение |
| #define DBL\_MAX | 1.7976931348623158e+308 | Max значение |
| #define LDBL\_MAX | 1.189731495357231765e+4932L | Max значение |
| #define FLT\_MAX\_EXP | 128 | Max показатель с основанием 2 |
| #define DBL\_MAX\_EXP | 1024 | Max показатель с основанием 2 |
| #define LDBL\_MAX\_EXP | 16384 | Max показатель с основанием 2 |
| #define FLT\_MAX\_10\_EXP | 38 | Max показатель с основанием 10 |
| #define DBL\_MAX\_10\_EXP | 308 | Max показатель с основанием 10 |
| #define LDBL\_MAX\_10\_EXP | 4932 | Max показатель с основанием 10 |

### Директивы #ifdef и #endif.

Директива #ifdef позволяет проверить, определена ли символическая константа и, если – да, следующие за ней строки будут направлены для обработки компилятором. Она относится к так называемым условным директивам, поскольку проверяет выполнение некоторого условия и в зависимости от результата проверки изменяет процесс компиляции. Используемая совместно с данной (и другими условными директивами), директива #endif сообщает препроцессору о конце условного блока кода.

Приведем примеры использования этих директив:

#ifdef \_DEBUG

//Какая-то часть исходного кода,

//которая должна выполняться в отладочной

//версии программы

#endif

Другой случай использования этих директив, который вы часто можете встретить в стандартных заголовочных файлах это случай использования функций языка С. В этом случае необходимо изменить объявление функции, добавив к нему модификатор extern "С", запрещающий использование декорирования имей. Обычно это выглядит следующим образом:

#ifdef \_cplusplus

extern "С" {

#endif

//Объявления заголовочного файла,

//требующие запрета декорирования имен

#ifdef\_clusplus

}

#endif

Директивы #ifdef и #endif представляют собой две из нескольких условных директив препроцессора. Их можно использовать для выборочного включения в программу некоторых операторов. Директива #endif применяется со всеми условными командами препроцессора и означает конец условного блока. Например, если ранее было определено имя LARGE\_CLASSES, то в следующем фрагменте программы описывается новое имя MAX\_SEATS:

#ifdef LARGE\_CLASSES

#define MAX\_SEATS 100

#endif

Всякий раз, когда в программе на C++ используются стандартные функции С, директиву #ifdef можно применять для модификации объявления функции с тем, чтобы выполнялась необходимая внешняя компоновка extern "С”, запрещающая трансляцию имени функции. Обычно для этого используются следующие два фрагмента с директивами, обрамляющими компилируемый код:

/\* использован в graph.h \*/

#ifdef cplusplus

extern "С" { /\* позволяет использование в C++ \*/

#endif

/\* объекты трансляции \*/

#ifdef cplusplus

}

#endif

Упомянутые нами директивы позволяют выполнять условную компиляцию. Вот пример.

#ifdef MAVIS

#include "horse.h" /\* выполнится, если MAVIS определен \*/

#define STABLES 5

#else

#include "cow.h" /'выполнится, если MAVIS не определен \*/

#define STABLES 15

#endif

Директива ifdef сообщает, что если последующий идентификатор (MAVIS) определяется препроцессором, то выполняются все последующие директивы вплоть до первого появления #else или #endif. Когда в программе есть #else, то программа от #else до #endif будет выполняться, если идентификатор не определен.

Такая структура очень напоминает конструкцию if-else языка Си. Основная разница заключается в том, что препроцессор не распознает фигурные скобки { }, отмечающие блок, потому что он использует директивы #else (если есть) и #endif (которая должна быть) для пометки блоков директив.

Эти условные конструкции могут быть вложенными.

Директивы #ifdef и #if можно использовать с #else и #endif таким же образом. Директива #ifndef опрашивает, является ли последующий идентификатор неопределенным; эта директива противоположна #ifdef. Директива #if больше похожа на обычный оператор if языка Си. За ней следует константное выражение, которое считается истинным, если оно не равно нулю:

#if SYS == " IBM"

#include "ibm.h"

#endif

Одна из целей использования «условной компиляции» — сделать программу более мобильной. Изменяя несколько ключевых определений в начале файла, вы можете устанавливать различные значения и включать различные файлы для разных систем.

### Директива #undef.

Директива #undef сообщает препроцессору об отмене всех предыдущих определений для указанного идентификатора. В следующем примере для изменения значения константы MAX\_SEATS используется уже знакомая директива #ifdef и #undef:

#ifdef LARGE\_CLASSES

#undef MAX\_SEATS 30

#define MAX\_SEATS 100

#endif

Дополнительно можно заметить, что компилятор не выдаст ошибки, если будет попытка отменить имя, ранее не определявшееся; если некоторое имя было отменено, то его можно определить заново при помощи другой директивы #define.

Директива #undef отменяет самое последнее определение поименованного макроопределения.

#define BIG 3

#define HUGE 5

#undef BIG /\* BIG теперь не определен \*/

#define HUGE 10 /\* HUGE переопределен как 10 \*/

#undef HUGE /\* HUGE снова равен 5 \*/

#undef HUGE /\* HUGE теперь не определен \*/

Очевидно (мы надеемся), вы не будете компилировать файл, как в этом примере. Предположим, что у вас есть большой стандартный файл, определенный директивой #include, который вы хотели бы использовать, но для этого некоторые из его определений для одной из функций программы нужно будет временно изменить. Вместо того чтобы иметь дело с этим файлом, вы можете просто включить его, а затем окружить такую выделяющуюся функцию соответствующими директивами #define и #undef.

Или, предположим, что вы работаете с большой системой программ. Вы хотите задать макроопределение, но не уверены, использует ли ваше определение другие определения откуда-нибудь из другого места системы. В этом случае просто отмените ваше макроопределение в том месте, где оно вам больше не нужно, а оригинал этого макроопределения, если он есть, будет по-прежнему оставаться в силе для остальной части системы.

### Директива #ifndef.

Сходной с директивой #ifdef является директива #ifndef. Эта директива также проверяет существование указанного в ней идентификатора, однако следующие за ней строки кода передаются компилятору, если этот идентификатор не определен. Приведем пример ее использования:

#ifndef WINVER

#define WINVER 0x0400

#endif

Несомненно, становится понятным принцип работы условных директив. По команде #ifndef препроцессор определяет отсутствие указанного идентификатора и выполняет некоторые действия. Следующий фрагмент кода взят непосредственно из файла stdio.h:

#ifndef \_SIZE\_T\_DEFINED

typedef unsigned int size\_t;

#define \_SIZE\_T\_DEFINED

#endif

В данном случае в выполняемых по условию операторах используются директивы препроцессора typedef и #define. Этот фрагмент кода определяет тип size\_t, который согласно требованию комитета ANSI С является возвращаемым типом для оператора sizeof().

### Директива #if.

В директиве #if также присутствует лексема defined. В следующем фрагменте показано как при помощи директивы #if и конструкции defined выполняются те же действия, которые требовали вложенных директив #ifndef и #ifdef:

#if defined(LARGE\_CLASSES) && !defined (PRIVATE\_LESSONS)

#define MAX\_SEATS 30

#endif

#ifdef LARGE\_CLASSES

#ifndef PRIVATE\_LESSONS

#define MAX\_SEATS 30

#endif

Два этих примера дают один результат; однако, первый воспринимается быстрее. В директивах #ifdef и #ifndef возможно единственное условие проверки; директива #if в сочетании с defined позволяет использовать сложные выражения.

### Директива #else.

Директива #else применяется совместно с условными директивами и позволяет отделить часть кода, которая будет обрабатываться, если предыдущее условие не выполняется. Например:

#ifdef \_\_FLAT\_\_

#include <win32\windowsx.h>

#else

#include <win16\windowsx.h>

#endif

Применение директивы #else достаточно очевидно. Предположим, что некоторая программа переносится с компьютера VAX на PC. Для целых чисел VAX отводит 4 байта или 32 бита, а в PC выделяется только 2 байта или 16 бит. Следующий фрагмент кода, использующий директиву #else, обеспечивает одинаковый размер целых чисел в обеих системах:

#ifdef VAX\_SYSTEM

#define INTEGER short int

#else

#define INTEGER int

#endif

Понятно, что при выполнении программы на компьютере VAX в ней должен быть определен идентификатор VAX\_SYSTEM. Легко увидеть, что комбинации директив препроцессора дают интересные решения.

### Директива #elif.

Две другие условные директивы - #if и #elif - также используются для проверки условия. Имя последней из них является сокращением английских слов “else if”. Она позволяет проверить условие, альтернативное установленному в директиве #if. Приведем пример их использования:

#if \_MSC\_VER < 901

//…

#elif \_MSC\_VER < 1001

//...

#endif

При использовании условных директив нужно иметь в виду следующие правила:

* для каждой директивы #if должна присутствовать соответствующая директива #endif;
* директивы #elif и #else являются необязательными; если ни одно из выражений не истинно, компилируется секция кода, следующая за #else;
* значение выражения должно быть целой константой;
* в выражениях могут применяться операции сравнения =, >, >=, <,<=.

Директива #elif является сокращением от "else if"; она используется как альтернативный вариант при построении вложенных операторов #if. При помощи следующих операторов определяется какой размер класса определен и однозначно задается макрос BILL:

#if defined (LARGE\_CLASSES)

#define BILL printf("\nCost per student $100.00.\n")

#elif defined (PRIVATE\_LESSONS)

#define BILL printf ("\nYour tuition is $1000.00.\n")

#else

#define BILL printf("\nCost per student $150.00.\n")

#endif

Нужно заметить, что директивы не обязательно начинаются с первой позиции в строке. Отступы в командах препроцессора обеспечивают читаемость текста программы; это — одна из многих полезных рекомендаций, разработанных комитетом ANSI и принятых в С Visual C/C++.

### Директива #line.

Директива #line изменяет внутренний счетчик строк компилятора. Синтаксис ее использования следующий:

#line целая\_константга <"имя\_файла">

Если ваша программа состоит из частей, составленных из других программных файлов, часто оказывается полезным пометить такие части номерами строк исходного файла, из которого они взяты, а не последовательными номерами, которые получаются в результате в объединенном файле. Директива #line указывает, что следующий номер строки исходного кода начинается с целой константы из файла с заданным именем, указанной в этой директиве. Если имя файла однажды указано в этой директиве, все последующие директивы #line задают нумерацию относительно того же файла. По умолчанию нумерация осуществляется относительно текущего файла. Макросы раскрываются в аргументы директивы #line. Эта директива чаще всего используется различными утилитами, поставляемыми с компилятором. Приведем пример использования этой директивы:

//Исходный файл В

#line 1

void AnotherFunc()

{

…

}

//Исходный файл А

#include "SourceB"

void main()

{

//…

ANotherFunc();

}

В этом случае при включении в исходный файл кода из другого файла, нумерация его срок начинается с единицы, а не с очередного номера в файле А.

Директива #line отменяет автоматическую нумерацию строк, выполняемую компилятором; она может помочь при отладке программы. Предположим, что в файл из 400 строк вставлена процедура длиной 50 строк. Все внимание уделяется ошибкам, которые могут возникнуть в этом объединенном коде.

Обычно компилятор начинает нумерацию с начала файла. Если возникла ошибка, то компилятор печатает сообщение с указанием номера строки — к примеру, 289. Где расположена эта ошибка в этом объединенном файле?

Если же включить директиву #Iine в начало только что добавленной процедуры, то компилятор выдаст номер ошибочной строки относительно начала функции:

#line 1

int imy\_mergefunction(void)

{

...

...

...

}

### Директива #error.

Директива #error указывает компилятору, что нужно напечатать сообщение об ошибке и прекратить компиляцию. Обычно ее используют внутри условных директив. Вот типичный пример ее использования:

#if!defined(MYNAME)

#error Должна быть определена \

константа MYNAME

#endif

#include <stdio.h>

#include <iostream>

using namespace std;

//Макроопределение

#define MU 13

int main()

{

setlocale(LC\_ALL, "Rus");

int a = 2, b = 3;

printf("%d", MU);

#undef MU

#if!defined(MU)

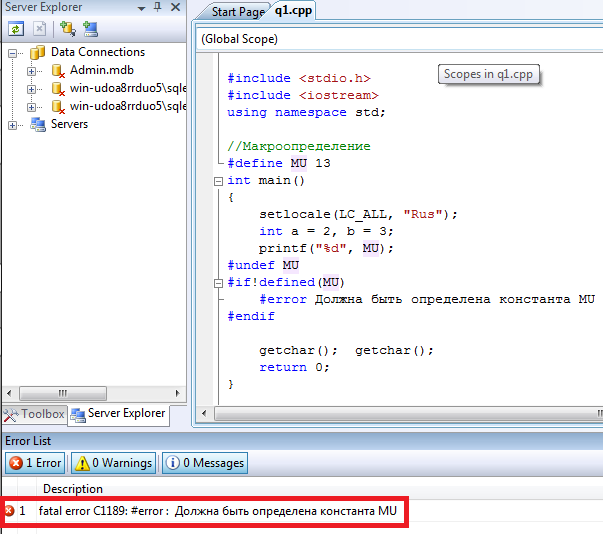
#error Должна быть определена константа MU

#endif

getchar(); getchar();

return 0;

}



Директива #еrrоr указывает компилятору на сообщение об ошибке, определенное пользователем и которое можно использовать для расширения собственных возможностей компилятора по расшифровке ошибок и выдаче сообщений. После того, как компилятор встречает директиву #error, он ищет в остальной части программы синтаксические ошибки, но объектный файл не создает. В следующем примере печатается предупреждение, если значение \_CHAR\_UNSIGNED не определено:

#if !defined( \_CHAR\_UNSIGNED )

#error /J option required. /\* нужна опция /J \*/

#endif

### Директива #pragma.

Директива #pragma определяет зависящие от реализации команды компилятора. Компилятор Visual C/C++ поддерживает следующие указания:

alloc\_text auto\_inline check\_pointer

check\_stack code\_seg comment

data\_seg function hdrstop

inline\_depth inline\_recursion init\_seg

intrinsic linesize loop\_opt

message native\_caller optimize

pack page pagesize

skip subtitle title

warning

Директива #pragma позволяет управлять возможностями компилятора. Синтаксис ее использования следующий:

#pragma имя\_директивы

К сожалению, имя директивы зависит от реализации этой директивы в компиляторе. Поэтому вам придется изучить документацию, относящуюся к конкретному компилятору. В таблице приведены несколько имен\_директив, встречающихся во всех известных нам компиляторах, и их назначение.

Таблица. Имена директив и их назначение

|  |  |
| --- | --- |
| **Директива** | **Описание** |
| Argsused | Подавляет предупреждение о неиспользуемом параметре для функции, следующей за директивой. |
| Hdrfile | Задает компилятору имя файла перекомпилированных заголовков. Синтаксис:  #pragma hdrfile "filename"  Здесь filename - имя файла перекомпилированных заголовков. |
| Hdrstop | Сообщает компилятору, что следующая за ней информация не включается в файл перекомпилированных заголовков. |
| Option | Директива предписывает компилятору использовать указанные в ней опции при компиляции следующего за ней кода. |
| Warn | Директива предписывает компилятору разрешить или подавить вывод предупреждающего сообщения. Синтаксис:  #pragma warn [+I - I.]xxx,  где xxx трехбуквенный идентификатор предупреждающего сообщения, используемый в командной строке с опцией -w. Если ему предшествует "+". то выдача сообщения разрешается, если "-", то запрещается. Если номеру сообщения предшествует точка, для него восстанавливается исходное состояние. |
| Warning | Директива предписывает компилятору разрешить или подавить вывод предупреждающего сообщения. Синтаксис:  #pragma warning(enable:хххх)  #pragma warning(disable:хххх)  #pragma warning(default:xxxx)  Первая форма разрешает вывод предупреждающего сообщения компилятора с номером хххх, вторая - запрещает, третья - восстанавливает поведение компилятора по умолчанию. |

Приведем примеры использования этой директивы:

#if defined (\_\_STDC\_\_)

//Следующая директива запрещает

//использование ключевых слов,

//не входящих в стандарт ANSI

#pragma warn -nak

#endif

//…

#ifndef \_DEBUG

//Следующая директива запрещает

//вывод предупреждающего сообщения

//с номером 4701

#pragma warning(disable:4701)

#endif

//…

#ifndef DEBUG

#if defined (\_\_STDC\_\_)

#pragma warn.nak

#endif

## Оператор defined.

В директивах #if и #elif может применяться оператор defined. Он позволяет проверить, был ли перед этим определен идентификатор или макрос с указанным в этом операторе именем. Можно также применять операцию логического отрицания (!) для проверки того, что идентификатор или макрос не определен. Следующий пример демонстрирует это:

#if defined(\_MFC\_COMPACT\_)

#if !defined(\_BORLANDC\_)

#undef \_ANONYMOUS\_STRUCT

#else

#define \_ANONYMOUS\_STRUCT

#endif

#endif

Могут использоваться и более сложные условия для проверки:

#if defined(\_\_LARGE\_\_) || \

defined(\_\_HUGE\_\_) || \

defined(\_\_COMPACT\_\_)

typedef long ptrdiff\_t;

#else

typedef int ptrdiff\_t;

#endif

Приведем пример использования директивы #elif:

#if defined (\_\_OS2\_\_)

#define \_RTLENTRY \_\_stdcall

#define \_USERENTRY \_\_stdcall

#elif defined (\_\_WIN32\_\_)

#define \_RTLENTRY \_\_cdecl

#define \_USERENTRY \_\_cdecl

#else

#define \_RTLENTRY \_\_cdecl

#define \_USERENTRY \_\_cdecl

#endif

## Условная компиляция.

Не всегда команды препроцессора располагаются только в заголовочных файлах; их можно использовать в исходном коде для более эффективной компиляции. Найдите небольшое отличие в следующем фрагменте программы (подсказка — размер исполняемой программы):

/\* компилируемый оператор if \*/

if(DEBUG\_ON) {

printf("Entering Example Function");

printf("First argument passed has a value of %d",ifirst\_arg);

}

/\* оператор сравнения \*/

#if defined(DEBUG\_ON)

printf("Entering Example Function");

printf("First argument passed has a value of %d",ifirst\_arg);

#endif

Первый оператор if компилируется всегда — это означает, что отладочная информация постоянно присутствует в исполняемом файле программы, отражаясь на его размере. Но что делать, если нежелательно продавать программу, имеющую вспомогательный код, нужный лишь на этапе проектирования? Для этого достаточно выполнить условную трансляцию этих избыточных операторов.

Во второй части фрагмента показана выборочная компиляция кода с использованием директивы #if defined. При отладке программы нужно определить DEBUGON; при этом компилятор будет видеть операторы, заключенные между командами #if...#endif. Однако, когда окончательная версия программы готова, определение параметра DEBUG\_ON удаляется. После этого компилятор перестает транслировать отладочные операторы и размер исполняемой программы уменьшается.

На простом примере можно увидеть, как при помощи директив #if...#endif отключать трансляцию двух операторов printf(). Скопируйте предыдущий фрагмент в какую-нибудь программу на С, не выполняющую никаких действий. Включите все необходимые заголовки и атрибуты (#include, main(). { и так далее). Не определяйте DEBUG\_ON. Скомпилируйте программу и убедитесь в том, что отсутствуют ошибки. После этого удалите оператор #include <stdio.h> и повторите трансляцию.

Теперь компилятор остановится на первом операторе printf(), заключенном в блоке операторов if...printf() Напечатается сообщение "Function 'printf’ should have a prototype." (Функция 'printf’ должна иметь прототип.).

Этого и следовало ожидать, поскольку оператор printf () внутри If всегда виден для компилятора. После этого достаточно удалить или закомментировать блок операторов if...printf() и повторить компиляцию. Компилятор не будет реагировать на операторы printf(), заключенные между директивами препроцессора #if...#endif — он их не увидит. Эти операторы станут видимыми на этапе компиляции только тогда, когда будет определен параметр DEBUG\_ON.

#include<stdio.h>

#include<iostream>

#define **MY\_TEST1 1**

#define **MY\_TEST2 2**

int **ifirst\_arg = 13;**

usingnamespace **std;**

int **main()**

{

/\* оператор сравнения \*/

#undef **MY\_TEST1**

//#undef MY\_TEST2

#ifdefined(MY\_TEST1)

printf("\nMY\_TEST1 is defined (%d)", MY\_TEST1);

#if !defined(MY\_TEST2)

printf("\nMY\_TEST2 is NOT defined");

#else

printf("\nMY\_TEST2 is defined (%d)", MY\_TEST2);

#endif

#else

**printf(**"\nMY\_TEST1 is NOT defined");

#ifdefined(MY\_TEST2)

**printf(**"\nMY\_TEST2 is defined (%d)", MY\_TEST2);

#else

printf("\nMY\_TEST2 is NOT defined");

#endif

#endif

#define **MY\_TEST3 3**

int **ifirst\_arg = 1;**

#undef **MY\_TEST3**

#define **MY\_TEST3 4**

/\* компилируемый оператор if \*/

if(MY\_TEST3)

**printf(**"\nMY\_TEST3 is defined (%d)", MY\_TEST3);

else

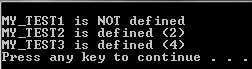
**printf(**"\nMY\_TEST3 is NOT defined");

printf("\n");

system("pause");

return **0;**

}



Подобным образом можно управлять видимостью различной информации — не только исполняемыми операторами.

#if defined(DEBUG\_ON)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Следующий фрагмент программы реализует \*/

/\* достаточно сложный алгоритм, требующий \*/

/\* комментариев и вывода отладочной информации \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf(" debug code goes here "); /\* далее следует \*/

#endif /\* отладочный код \*/

В данном примере имеется не только условный оператор для вывода отладочной информации, но и предусматриваются пояснительные комментарии. Некоторое дополнительное время, необходимое для написания условно-компилируемого кода, компенсируется простотой отладки программы и небольшим размером исполняемого кода.

## Дополнительные операции препроцессора.

Существуют три операции, которые можно использовать только в директивах препроцессора: подстановка строки (#), конкатенация (##) и подстановка символа (#@).

### Операция подстановки строки (#).

Если перед параметром макроса используется символ #, то компилятор вместо значения этого параметра подставляет его имя. В результате имя аргумента преобразуется в строку. Эта операция необходима потому, что параметры не заменяются, если они входят в литеральную строку, записанную в макросе в явном виде. Следующий пример иллюстрирует синтаксис операции подстановки строки:

#define STRINGIZE(ivalue) printf(#ivalue " is: %d",ivalue)

...

...

...

int ivalue = 2;

STRINGIZE(ivalue);

Результат работы макроса:

ivalue is: 2

Если перед параметром макроса указана операция подстановки символа (#), то компилятор при раскрытии макроса вместо параметра подставит его имя. Как и любая строка, результат этой операции может объединяться со смежными строками, если отделяется он них только пробелами. Рассмотрим пример:

#include <stdio.h>

#include <iostream>

using namespace std;

#define SHOWVAL(val)\

printf(#val "= %d \n", (int)(val))

int main()

{

setlocale(LC\_ALL, "Rus");

int icount = 10;

//...

SHOWVAL(icount);

getchar(); getchar();

return 0;

}

При выполнении программа выводит на экран:

icount = 10;



### Операция конкатенации (##).

Одно из применений операции конкатенации — динамическое создание имен переменных и макроопределений. Операция объединяет синтагмы, и, удаляя все пробелы, образует новое имя. Если операция ## используется в макроопределении, то она обрабатывается после подстановки параметров макроса, перед тем, как этот макрос проверяется на наличие каких-либо дополнительных операций. Например, в следующем примере показано, как создавать имена переменных с помощью команд препроцессора:

#define IVALUE\_NAMES(icurrent\_number) ivalue ## icurrent\_number;

...

...

...

int IVALUE\_NAMES(1);

Компилятор получит следующее объявление:

int ivalue1;

Заметим, что препроцессор удалил пробелы; поэтому ivalue1 не будет выглядеть как ivalue1. Эту операцию можно объединять с другими директивами препроцессора для получения сложных определений. В следующем примере операция конкатенации используется для создания имени макроопределения, по которому препроцессор вызывает соответствующий макрос:

#define MACRO1 printf("MACR01 invoked.")

#define MACR02 printf("MACR02 invoked.")

#define MAKE\_MACRO(n) MACRO ## n

...

...

...

MAKE\_MACRO(1);

Результат выполнения этого примера следующий:

MACRO1 invoked. // вызван MACR01

Операция конкатенации (или склейки) ## вызывает объединение двух строк в одну. Перед объединением строк удаляются разделяющие их пробелы. Если для полученной в результате склейки строки существует макроопределение, препроцессор выполняет раскрытие макроса. Например:

#define MACRO1 printf ("Вызов MACRO1.")

#define MACRO2 printf("Вызов MACRO2.")

#define CREATE\_MACRO(n) MACRO ## n

//... CREATE\_MACRO(1);

В результате раскрытия макроса препроцессором компилятор получит следующую инструкцию:

printf("Вызов MACRO1.");

Если операция конкатенации используется в макроопределении, то вначале препроцессор выполняет подстановку аргументов вместо формальных параметров, затем выполняет операцию конкатенации. После этого выполняются остальные операции макроопределения. Например:

#define DEFVAL(i) int var ## i

//…

DEFVAL(1);

В результате раскрытия макроса препроцессором компилятор получит следующую инструкцию:

int var1;

### Операция подстановки символа (#@).

Операция подстановки символа предшествует формальным параметрам в макроопределении. При этом фактический параметр рассматривается как отдельный символ, заключенный в простые кавычки. Например:

#define CHARIZEIT(cvalue) [#@cvalue](mailto:#@cvalue)

...

...

...

cletter = CHARIZEIT(z);

Компилятор получит следующее выражение:

cletter = 'z';

## Обработка ошибок: perror().

Среди многих интересных функций, имеющих прототипы в файле stdio.h, есть функция с именем реrrоr(). Эта функция выводит в поток stderr системное сообщение о последней ошибке, произошедшей при вызове библиотечной процедуры. Для этого используются переменные errno и \_sys\_errlist, предопределенные в файле stdlib.h. Массив \_sys\_erriist представляет собой массив строк сообщений об ошибках. Переменная errno — это индекс массива строковых сообщений, который автоматически устанавливается равным номеру возникшей ошибки. Число элементов в массиве определяется другой константой, \_sys\_nerr, также определенной в файле stdlib.h.

Функция реrrоr() имеет единственный параметр — строку символов. Обычно передаваемый аргумент — это строка, идентифицирующая файл или функцию, где произошла ошибка. Использовать эту функцию просто; это показано в следующем примере:

/\*14PERROR.C

Программа на С, демонстрирующая функцию perror(), имеющую прототип

в файле STDIO.H\*/

#include <stdio.h>

void main(void)

{

FILE \*fpinfile;

fpinfile = fopen("input.dat", "r");

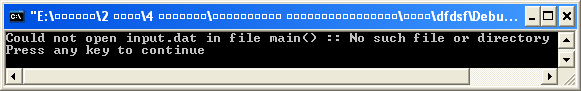
if (!fpinfile)

/\* Невозможно открыть input.dat в файле main() : \*/

perror("Could not open input.dat in file main() :");

}

Печатается следующее сообщение:



## Модели памяти.

Компилятор Visual C/C++ поддерживает шесть стандартных моделей памяти — tiny (миниатюрная), small (малая), medium (средняя), compact (компактная), large (большая) и huge (гигантская) — а также пользовательскую модель, которая используется в тех приложениях, где имеются особые требования к хранению данных и кода. Тщательно выбирая подходящую модель памяти для приложения, можно улучшить степень использования системных ресурсов и скорость выполнения программы. Ниже кратко описываются отличительные особенности каждой из шести стандартных моделей памяти.

Существует ограниченное число задач, решение которых требует минимального доступа к системным ресурсам вычислительного средства. На практике чаще всего приходится учитывать особенности архитектуры компьютера с тем, чтобы разрабатываемое приложение было максимально гибким. Основные ограничения на использование системных средств ЭВМ накладывает организация памяти.

Память компьютеров на базе процессоров семейства Intel 80x86 представляется в виде сегментов или частей от 0 байт до 64 Кбайт.

Как известно, в состав процессора входят регистры - быстро доступные ячейки собственной памяти. Для различных арифметических операций используются регистры общего назначения: АН. AL, BH, BL, СН. CL, DH и DL, которые в случае необходимости могут группироваться в сдвоенные: АХ (из АН и AL), ВХ (из ВН и BL). СХ (из СН и CL) и DX (из DH и DL).

Доступ к ячейке оперативной памяти организован через обращение к значению сегмента и указание смещения внутри данного сегмента. Процессор имеет четыре сегментных регистра, посредством которых осуществляется манипулирование памятью:

* CS (сегмент кода) предназначен для обеспечения адресации к сегменту, содержащему исполняемые программные коды;
* DS (сегмент данных) используется ятя адресации глобальных и статических переменных;
* ES (дополнительный сегментный регистр), как и DS. содержит сегментное значение для доступа к глобальным и статическим переменным, является вспомогательным регистром;
* SS (сегмент стека) служит для адресации к сегментному значению стека программы и, как следствие, к локальным переменным;

При адресации памяти в DOS процессор преобразует значение сегментного регистра и смещения в линейный адрес. В случае работы в операционной системе Windows сегментный регистр служит в качестве селектора дескрипторной таблицы, содержащей действительный адрес.

К счастью, компиляторы ведущих фирм предусматривают так называемые расширения языка, помогающие обходить те или иные ограничения системы. В этой главе мы познакомимся с основными из их них.

Доступ к переменным или функциям в программе также осуществляется через сегментные регистры. В регистр DC заносится базовое значение, после чего процессору тем или иным способом передается значение смещения. Поскольку размер сегмента не должен превышать 64 Кбайт, очень вероятно, что в реальности коды или данные будут располагаться в различных сегментах. Таким образом, каждый раз при обращении к переменным, расположенным в различных сегментах, программе необходимо будет загружать в регистр DC новое значение сегмента. Совершенно очевидно, что для обращения к разным сегментам требуется гораздо больше времени, так как каждый раз приходится загружать новое базовое значение. В лучшем случае все данные или все коды программы будут умещаться в один единственный сегмент и для доступа к любой переменной или функции будет достаточно загрузить смещение лишь один раз. Однако в данном случае размер программы был бы ограничен величиной одного сегмента, то есть 64 Кбайтами. В настоящее время такое ограничение неприемлемо за редким исключением, да чаще всего и не нужно.

Для того чтобы имелась возможность выбора оптимальной конфигурации конечного программного продукта. C++ предлагает к использованию различные модели памяти. Модели памяти служат для управления выделением ресурсов при выполнении разработанного приложения. Существует шесть моделей памяти: Tiny, Small, Medium, Compact, Large и Huge.

DOS приложения могут использовать любую из перечисленных моделей, в то время программы, разработанные для функционирования в 16-разрядной ОС Windows (например, Windows 3.11) - только одну из четырех: Small, Medium, Compact или Large. В сущности, выбор одной из них задается определенным ключом компилятора, однако может меняться для той или иной функции или переменной опционально в программе (по мере надобности). Помимо определения требований, предъявляемых разрабатываемой программой к объему памяти, выбор определенной модели влияет на размер указателей, динамической памяти, а для DOS программ - еще и на размер области стека. В связи с вышеизложенным указатели бывают ближними и дальними.

Ближний указатель содержит только значение смещения, в то время как значение сегмента адреса находится в сегментном регистре. Этот тип указателей используется по умолчанию в моделях с ближней динамической памятью Tiny, Small и Medium. При этом сегментная часть адреса данных размещается в регистре СS. Программный код размером до 64 Кбайт применяет модель с короткими указателями кода (Tiny, Small или Compact), размещая сегментную часть адреса в регистре CS. Таким образом, функция или переменная будет доступна только в том сегменте кода, в котором она откомпилирована.

Дальние указатели содержат как сегментную часть адреса, так и смещение. Этот тип указателей используется по умолчанию в моделях памяти Compact, Large и Huge - для данных, и в Medium, Large и Huge - для кодов программы.

В модели Tiny в регистры CS, DS, SS и ES заносится одинаковое значение сегментной части адреса. Код программы, статические, динамические данные и стек в общей сложности могут занимать всего лишь 64 Кбайта. По умолчанию переменная типа указатель в такой модели памяти занимает два байта (ближний указатель) и содержит смещение внутри данного сегмента памяти. Очень небольшое число задач может решаться при использовании такой модели - ее применяют при существенном недостатке памяти.

Модель Small предусматривает код программы размером до 64 Кбайт (один сегмент) и еще один сегмент под статические, динамические данные и стек. Указатели в такой модели также занимают два байта и содержат смещение внутри используемого сегмента. Эта модель применяется для решения небольших и средних задач.

В модели Medium код разрабатываемого приложения может занимать до 1 Мбайта (1024 Кбайт), в то время как данные и стек размещаются в пределах одного сегмента. Таким образом, в коде программы используются дальние указатели, занимающие четыре байта. Адресация данных подразумевает использование ближних указателей (по два байта). Эту модель следует использовать при разработке больших приложений, оперирующих с небольшим количеством данных.

Модель Compact отводит под код программы один сегмент (до 64 Кбайт) и по одному сегменту под статические данные и стек. Для адресации в коде программы задействуют, в то время как адресация данных осуществляется посредством дальних указателей. Компактная модель разработке небольших и средних приложений, большого объема статических данных.

В модели Large код программы занимает до 1 Мбайта, а статические данные со стеком - по одному сегменту до 64 Кбайт. В модели используются дальние четырехбайтные указатели и для адресации кода и для адресации данных. Модель Large часто используют на практике при решении больших задач.

Модель Huge допускает превышение статическими данными объема 64 Кбайт. В остальном эта модель аналогична модели Large и использует четырехбайтные дальние указатели.

**Таблица. Модели памяти 16-разрядных приложений**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DOS | | | | | | |
| Модель | | Код | Данные | Стек | | Указатель по умолчанию |
| Tiny | | код + данные + стек + динамическая память до 64 Кбайт | | | | near |
| Small | | 64 Кбайт | данные + стек + динамическая память до 64 Кбайт | | | near |
| Medium | | 1 Мбайт | данные + стек + динамическая память до 64 Кбайт | | | near |
| Compact | | 64 Кбайт | 64 Кбайт | 64 Кбайт | | fаr |
| Large | | 1 Мбайт | 64 Кбайт | 64 Кбайт | | fаr |
| Huge | | 1 Мбайт | > 64 Кбайт | 64 Кбайт | | fаr |
| 16-bit Windows EXE | | | | | | |
| Small | 64 Кбайт | | данные + стек + динамическая память до 64 Кбайт | | | near |
| Medium | 1 Мбайт | | данные + стек + динамическая память до 64 Кбайт | | | near |
| Compact | 64 Кбайт | | данные + стек до 64 Кбайт | | | fаr |
| Large | 1 Мбайт | | данные + стек до 64 Кбайт | | | far |
| 16-bit Windows DLL | | | | | | |
| Small | 64 Кбайт | | 64 Кбайт | | \_\_ | far |
| Medium | 1 Мбайт | | 64 Кбайт | | — | far |
| Compact | 64 Кбайт | | 64 Кбайт | | — | far |
| Large | 1 Мбайт | | 64 Кбайт | | — | far |
| 32-bit Windows | | | | | | |
| Flat | 1 Гбайт | | 1 Мбайт | | 1 Мбайт | far |

Для изменения длины указателей, применяемой по умолчанию, используются модификаторы:

near - для генерирования компилятором ближних вызовов;

far - для генерирования дальних вызовов.

Следует понимать, что ближний указатель всегда может быть преобразован компилятором в дальний, однако дальний указатель не может быть преобразован в ближний. Синтаксис спецификаторов вызова имеет следующий вид:

тип near \*имя\_переменной;

тип far \*имя\_переменной;

что читается как "объявить указатель \*имя\_переменной типа 'тип' как ближний / дальний". Если же поместить модификатор дальнего или ближнего вызова между символом звездочка (\*) и именем переменной, компилятор поймет такую запись в ином контексте и в результате объявит указатель на переменную в том же (near) или обязательно в другом (far) сегменте (то есть модификатор в этом случае влияет на месторасположение указателя, а не на его размер).

Необходимо отметить, что модификаторы в библиотеках компиляторов различных фирм могут в синтаксисе включать ведущие символы подчеркивания (один или два). Так модификаторы:

near

\_nеar

\_\_near

подразумевают одно и то же. В дальнейшем изложении данного материала нет жесткой привязки к какому-либо типу написания, однако, следует иметь этот факт в виду и руководствоваться при разработке приложений технической документацией к конкретному компилятору. Кроме того, модификаторы с одним ведущим символом подчеркивания зачастую используются компилятором для внутренних нужд, и пользоваться ими не рекомендуется.

Ниже приводится несколько вариантов применения модификатора:

// Символьная переменная в сегменте

// по умолчанию:

char А = 'а';

// Символьная переменная в дальнем

// сегменте данных:

char far В = 'b';

// Указатель на символьную переменную,

// размешается в сегменте по умолчанию:

char\* рА = &A;

// Дальний указатель на символьную

// переменную А, размещается

// в сегменте данных по умолчанию:

char far\* pfA = &A;

// Обычный указатель на символьную

// переменную, размещаемый в дальнем

// сегменте данных:

char\* far pBl = &B;

// Дальний указатель на символьную

// переменную, размещаемый в дальнем

// сегменте данных:

char far\* far pB2 = &B;

### Модель tiny.

При использовании модели памяти tiny создаются программные файлы с расширением .СОМ. Такие программы содержат один 64 Кб сегмент для кода и данных. Доступ ко всем объектам кода и данных осуществляется при помощи near (ближних) адресов. Эти программы не могут использовать библиотеки, содержащие far-функции, например, библиотеки графики. Файлы с расширением .СОМ можно запускать только под DOS. Кроме этого, компилятор Microsoft Visual C/C++ не создает р-код для модели tiny. Программы с этой моделью памяти используют память аналогично описанным в следующем параграфе программам с моделью памяти small. Однако, при компоновке tiny-приложений объектный файл связывается с библиотекой CRTCOM.LIB; в результате исполняемый файл получается с расширением .СОМ, а не .ЕХЕ.

### Модель small.

Когда выбрана опция small, программа может иметь два сегмента: один для данных и один для кода. После компиляции программы с моделью памяти small получают расширение .ЕХЕ. Эта опция используется по умолчанию, когда не указана другая модель памяти. Размер обоих сегментов — данных и кода — ограничен 64 Кб. Длина программы с моделью памяти small не может превышать 128 Кб. По умолчанию в таких программах используются near-адреса как для кода, так и для данных.

### Модель medium.

Программа с моделью памяти medium может иметь один сегмент для данных, но множество сегментов — для кода. Поэтому такие программы могут содержать более 64 Кб кода, но не более 64 Кб данных. По мере необходимости программный код может занимать столько места, сколько необходимо, однако при этом размер всего блока данных не должен превышать 64 Кб. По умолчанию в программах с моделью памяти medium используются far (дальние) адреса для кода и near (ближние) адреса — для данных. Установки по умолчанию можно отменить при помощи ключевого слова \_near.

### Модель compact.

При использовании модели памяти compact программы могут иметь несколько сегментов данных, но только один сегмент кода. Эта модель памяти наиболее подходит для тех приложений С, у которых большой объем данных, но короткий код. Модель памяти compact позволяет данным занимать столько места и сегментов, сколько нужно. По умолчанию в программах с этой моделью памяти используются near-адреса для кода и far-адреса для доступа к данным. В приложении можно переопределить эти установки при помощи ключевого слова \_near или \_huge для данных и ключевого слова \_far для кода.

### Модель large.

Как нетрудно догадаться, приложения с моделью памяти large могут иметь по несколько сегментов для кода и данных. Однако, размер отдельного объекта данных не должен превышать 64 Кб. Эта модель памяти подходит для основных программ, работающих с большими объемами данных. По умолчанию в программах с моделью памяти large far-адреса используются как для кода, так и для данных. Эти установки можно переопределить в приложении при помощи ключевого слова \_near или \_huge для данных и ключевого слова \_far для кода.

### Модель huge.

Модели памяти huge и large схожи. Главное отличие заключается в том, что при использовании модели huge снимаются ограничения на размер отдельных объектов данных. Однако, имеются ограничения на размер элементов массива, если этот массив больше, чем 64 Кб. Элементы массива не могут пересекать границы сегментов — это гарантирует эффективную адресацию для каждого элемента. Поэтому размер отдельного элемента массива не может превышать 64 Кб. Кроме того, если массив больше 128 Кб, то размер каждого элемента в байтах должен быть равным степени числа 2. Однако, если размер массива равен 128 Кб или меньше, то его элементы могут иметь любую длину до 64 Кб максимум. Необходимую модель памяти можно устанавливать непосредственно в среде разработки. Для этого достаточно выбрать пункт главного меню Option Language Options, а затем указать С or C++ Compiler options.... Следующее диалоговое окно будет иметь пункт Memory Model. Для выбора нужно щелкнуть по стрелке вниз.

Если проводятся некоторые арифметические операции над указателями, они изменяют только значение смещения, оставляя значение сегмента неизменным. В этом случае для указателей, расположенных, например, в конце сегмента, возможна ситуация, когда при увеличении указателя, вместо того, чтобы обратиться к значению другого сегмента, произойдет переход через границу последнего и будет произведено обращение к нулевому смещению того же самого сегмента. То же утверждение верно для указателей, расположенных в начале сегмента. Для того чтобы компилятор сгенерировал код, позволяющий выходить указателям за текущий сегмент (нормализация указателя), применяется модификатор huge. В этом случае синтаксис имеет следующий вид:

тип huge \*имя\_указателя

Приведенный ниже пример демонстрирует использование модификатора huge.

#include <stdio.h>

int main ()

{

long huge \*p = NULL;

for(int i=0; i<10; i++, p++)

printf("p содержит адрес %Fp\n", p);

return 0;

}

#include <math.h>

#include <stdio.h>

#include <iostream>

using namespace std;

int main()

{

//setlocale(LC\_ALL, "Rus");

long huge \*p= NULL;

for(int i= 0; i< 10; i++, p++)

printf("p содержит адрес %Fp\n", p);

getchar(); getchar();

return 0;

}

В рассмотренной программе подключается заголовочный файл stdio.h, содержащий прототип функции форматированного вывода данных printf () - здесь с ее помощью производится отображение указателя в виде сегмент: смещение. Далее определяется указатель типа long huge и в цикле производится инкремент и вывод 10 значений адресов дальних указателей. Из результатов работы программы видно, что осуществляется нормализация указателя, и величина смещения не превышает значения 15, в то время как значение сегмента изменяется.

Во время выполнения приложения неизбежны вызовы каких-либо функций. Параметры функций и переменные, объявленные внутри, являются локальными объектами и располагаются внутри стека. Кроме того, в стеке хранится информация (например, адрес возврата, состояние программы до вызова функции, выполняющейся в текущий момент), необходимая программе при возврате из вызываемых функций.

Глобальные же объекты хранятся в сегменте данных, а так как размер данных для всех моделей памяти кроме huge не может превышать одного сегмента, количество глобальных переменных в программе ограничено. Если компилятор при сборке приложения указывает, что используется слишком много глобальных объектов, часть из них можно описать как дальние (far). Если же необходимо определить переменную, о которой изначально известно, что ее размер выходит за пределы 64 Кбайт, используют модификатор huge. Так, например, глобальный массив большого размера можно описать следующим образом:

long huge AKR[30000];

Таким образом, с помощью спецификаторов far и huge можно определять переменные, если изначально известно, что размер приложения будет выходить за рамки отведенных 64 Кбайт.

## Модификаторы функций.

В зависимости от того, какая модель памяти применяется, функции и указатели на функцию являются ближними или дальними. Как и в случае с указателями на переменные, размер указателя на функцию можно изменить с помощью модификаторов near и far. Так, например, функции, обращение к которым осуществляется операционной системой или драйвером какого-либо устройства, всегда должны быть объявлены как дальние (far) и, если используется модель памяти с короткими указателями для кода (Tiny, Small или Compact), такие функции должны специфицироваться с помощью модификатора far.

Рассмотрим пример использования модификаторов функций.

#include <iostream.h>

void \_\_near PrintNear(char\*);

void \_\_far PrintFar(char\*);

void (\_\_near\* nPrint)(char\*) = PrintNear;

void (~far\* fPrxnt) (char\*) = PrintFar;

int main()

{

nPrint("NEAR");

fPrintC'FAR");

return 0;

}

void \_\_near PrintNear(char\* nMess)

{

cout<< nMess<<’\n’;

cout <<"вызов ближней функции \n";

}

void\_ far PrintFar(char\* fMess)

{

cout << fMess << ’\n’

cout << "Вызов дальней функции\n";

}

#include <stdio.h>

#include <iostream>

using namespace std;

void \_\_near PrintNear(char\*);

void \_\_far PrintFar(char\*);

void (\_\_near\* nPrint)(char\*) = PrintNear;

void (~far\* fPrint) (char\*) = PrintFar;

int main()

{

nPrint("NEAR");

fPrint("FAR");

return 0;

}

void \_\_near PrintNear(char\* nMess)

{

cout << nMess << '\n';

cout << "Вызов ближней функции \n";

}

void \_\_far PrintFar(char\* fMess)

{

cout << fMess << '\n';

cout << "Вызов дальней функции\n";

}

После объявления функций PrintNear() и PrintFar() следует определение указателей на эти функции короткого nPrint и длинного fPrint соответственно. Тело функции main() содержит вызов ближней и дальней функции через их указатели.

## Модификаторы cdecl и pascal.

Как уже доминалось ранее, при вызове некоторой функции ее параметры обычно помещаются в стек, а при возврате из функции параметры из стека удаляются. Помещение и удаление параметров из стека осуществляется в соответствии с определенными правилами. Существует два варианта помещения аргументов в стек; слева направо (соглашение о вызове языка Pascal) и справа налево (соглашение о вызове языка С). По умолчанию компилятор использует соглашение, указанное в параметрах (при вызове из командной строки или в опциях интегрированной среды разработки приложения), однако программист всегда может явно указать ятя каждой функции, какое из соглашений вызова необходимо применять. Это осуществляется с помощью модификаторов cdecl и pascal.

Оба модификатора соответствующим образом влияют на внутреннее имя функции при декорировании имен (будет рассматриваться позже), тем самым сообщая компилятору использовать то или иное соглашение о вызове.

Модификатор cdecl указывает компилятору на то, что параметры вызываемой функции должны помещаться в стек в порядке, обратном следованию при вызове, то есть справа налево (первым передается аргумент, стоящий последним в списке параметров, затем предыдущий и т.д.). При этом не делается никаких предположений об ответственности функции за очистку стека. Внутреннее имя функции эквивалентно объявляемому имени с добавлением символа подчеркивания и использованием соглашения языка С о различении верхнего и нижнего регистров:

MyFunction - имя из прототипа функции

\_MyFunction - внутреннее имя функции

Модификатор pascal наоборот, требует прямой передачи параметров в стек, слева направо. Кроме того, данный спецификатор сигнализирует, что именно вызываемая функция ответственна за очистку стека. Данное соглашение используют на практике, если функция вызывается много раз из разных мест. Однако функции с переменным числом параметров не могут использовать данного соглашения. Внутреннее имя при использовании данного вида соглашения совпадает с именем прототипа, но с преобразованием в верхний регистр:

MyFunction - имя из прототипа функции

\_MYFUNCTION - внутреннее имя функции

Если по умолчанию используется соглашение языка Pascal, следует об этом помнить, так как имя главной функции main() будет преобразовано не в \_main, а в MAIN, что может вызвать сообщение компилятора об ошибке, обойти которую поможет использование модификатора cdecl:

int \_\_cdecl main()

{

…

return 0;

}

Иногда необходимо ускорить вызов функции. Этого можно достичь путем передачи параметров функции не через стек, а непосредственно через регистры общего назначения, указав перед именем функции спецификатор fastcall. Поскольку число регистров ограничено, следует применять быстрый вызов только для функций, число параметров которых не больше трех. При этом тип аргументов должен соответствовать типу char, int или long, либо быть коротким указателем. Внутреннее имя функции совпадает с именем ее прототипа с добавлением символа @:

MyFunction - имя из прототипа функции

@MyFunction - внутреннее имя функции

С++ предлагает еще и использование стандартного соглашения о вызове, которое может применяться только в 32-разрядных приложениях. Это своеобразный конгломерат соглашений языков С и Pascal, который специфицируется с помощью модификатора stdcall. При стандартном соглашении о вызове параметры помещаются в стек справа налево, как в случае с cdecl, однако вызванная функция сама отвечает за очистку стека. Внутреннее имя функции полностью совпадает с именем, указанным в прототипе:

MyFunction - имя из прототипа функции

MyFunction - внутреннее имя функции

Другим способом ускорения работы той или иной части функции является использование встроенного ассемблерного кода. Ключевое слово asm определяет следующий за ним однострочный (до конца строки) или блочный (многострочный, заключенный в фигурные скобки) ассемблерный оператор, который имеет возможность взаимодействия с другими переменными, используемыми в программе. Ниже рассмотрен пример, позволяющий двум переменным быстро обменяться своим содержимым. Но вначале простой пример вставки ассемблерного кода в текст программы на языке c++

#include<iostream>

usingnamespace **std;**

int **main()**

{

short **a=0;**

\_\_asm

**{**

**mov ax, a**

**inc ax**

**mov a, ax**

**}**

**cout << a <<** "\n"; //1

**system(**"pause");

}



#include<stdio.h>

#include<iostream>

usingnamespace **std;**

int **main()**

{

**setlocale(LC\_ALL,** "Rus");

int **var1 = 55;**

int **var2 = 99;**

**cout <<** "До обмена:\n";

**cout << var1 <<** '\n';

**cout << var2 <<** '\n';

\_\_asm

**{**

**push var1**

**push var2**

**pop var1**

**pop var2**

**}**

**cout <<** "После обмена:\n";

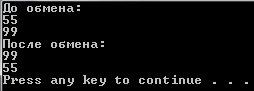
**cout << var1 <<** '\n';

**cout << var2 <<** '\n';

**system(**"pause");

return **0;**

}



В примере сначала объявляются и инициализируются две целочисленные переменные, а затем выводится их содержимое. Далее в блоке ассемблерной вставки первая, а затем и вторая переменная помещаются в стек, после чего в качестве первой переменной извлекается значение второй переменной (помещена в стек последней), а после этого из стека извлекается значение первой переменной с присвоением var2. Рис. 0601 иллюстрирует обмен значений переменных посредством стека.



**Рис.0601** Обмен значений переменных через стек

Следует, однако, помнить, что операции с ассемблерными вставками небезопасны: если в них используется изменение содержимого регистров общего назначения, это может неожиданным образом повлиять на ход выполнения всей программы. Для использования тех или иных регистров правильно будет при входе в ассемблерную вставку предварительно сохранить, например, в стеке, значения изменяемых регистров, а на выходе восстановить исходное значение. Ниже рассмотрен тот же пример с использованием регистра общего назначения АХ:

#include<stdio.h>

#include<iostream>

usingnamespace **std;**

int **main()**

{

**setlocale(LC\_ALL,** "Rus");

shortint **var1 = 55;**

shortint **var2 = 99;**

**cout <<** "До обмена:\n";

**cout << var1 <<** '\n';

**cout << var2 <<** '\n';

\_\_asm

**{**

// сохраняем в стеке первоначальное

// значение регистра АХ:

**push ax**

// осуществляем обмен

// значениями переменных

**mov ax, var1**

**push ax**

**mov ax, var2**

**mov var1, ax**

**pop var2**

// восстанавливаем

// первоначальное значение

// регистра АХ:

**pop ax**

**}**

**cout <<** "После обмена:\n";

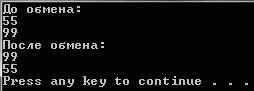
**cout << var1 <<** '\n';

**cout << var2 <<** '\n';

**system(**"pause");

return **0;**

}



C++ также предусматривает непосредственные операции над регистрами и флагами процессора без использования ассемблерной вставки. Для этого зарезервированы следующие ключевые слова:

|  |  |  |
| --- | --- | --- |
| \_АХ | \_АН | \_AL |
| \_BX | \_BH | \_BL |
| \_CX | \_CH | \_CL |
| \_DX | \_DH | \_DL |
| \_SI | \_DI | \_BP |
| \_SP | \_ES | \_SS |
| \_CS | \_DS | \_FLAGS |

Кроме этого, если в опциях компилятора разрешено использование инструкций процессора Intel 80x386. в программе также можно использовать дополнительные ключевые слова:

|  |  |
| --- | --- |
| \_ЕАХ | \_ESI |
| \_ЕВХ | \_EDI |
| \_ЕСХ | \_ESP |
| \_EDX | \_ЕВР |

Ниже приведен пример, демонстрирующий использование псевдорегистров.

#include <iostream.h>

int main ()

{

int varl = 55;

cout << \_AX << ' \n';

cout << '\n';

\_AX = varl;

cout << \_AX << '\n ';

return 0;

}

#include <stdio.h>

#include <iostream>

using namespace std;

int main()

{

int var1 = 55;

cout << \_AX << '\n';

cout << '\n';

\_AX = var1;

cout << \_AX << '\n';

return 0;

}

Первоначально значение регистра АХ неоднозначно, поэтому первый оператор вывода покажет какое-то неопределенное значение, далее идет присвоение регистру значения переменной varl и вывод этого значения на экран. Следует учесть, что псевдорегистры нужно использовать с осторожностью, так как многие операции могут менять значения регистров еще до того, как в программе следует использование последних. Так, например, если в вышеприведенный пример после присвоения

\_АХ = var1;

добавить строку

cout << "Привет!";

значение регистра будет изменено и на экране второй оператор вывода отобразит неопределенное значение.

Внутри блока ассемблерной вставки можно помещать метки и вызовы других функций программы. При этом перед вызовом функции необходимо самостоятельно размещать аргументы в стеке, а после вызова может потребоваться его (стека) очистка.

Следующий пример иллюстрирует вызов внешней функции из ассемблерной вставки (для модели памяти Large).

#include<stdio.h>

#include<iostream>

usingnamespace **std;**

void **ShowSum(**int, int);

int **main()**

{

int **a = 2;**

int **b = 3;**

\_\_asm

**{**

// Поместить параметры в стек:

**push a**

**push b**

// Вызов дальней функции

// (модель Large!)

//call far ptr ShowSum

// Пример ближнего вызова:

**call near ptr ShowSum**

// Очистка стека:

**pop cx**

**}**

**system(**"pause");

return **0;**

}

void **ShowSum(**int **x,** int **y)**

{

**cout << (x + y);**

}



Здесь в ассемблерной вставке следует помещение аргументов функции в стек, а также вызывается функция, выводящая сумму полученных аргументов.

В качестве иллюстрации использования ассемблерных вставок с метками рассмотрим пример, осуществляющий вывод на экран символов от А до Z:

#include<stdio.h>

#include<iostream>

usingnamespace **std;**

int **main()**

{

char **letter =** 'A';

char **stop =** 'Z';

start:

\_\_asm

**{**

**mov ah, letter**

**mov al, stop**

**cmp ah, al**

**jle print**

**jmp exit**

**}**

print:

**cout << letter;**

**letter++;**

**cout <<** " ";

goto **start;**

**system(**"pause");

**exit;**

return **0;**

}



В этом примере символьной переменной letter присваивается значение символа 'А', переменной stop - значение символа ' Z '. Далее в регистр ah заносится значение переменной letter, а в регистр al - значение переменной stop, после чего команда cmp ah, al сравнивает значения регистров между собой. Если ah <= al, осуществляется переход на метку print и переменная letter выводится на печать. Вслед за этим значение переменной letter увеличивается, и цикл повторяется с метки start. Если же значение ah больше значения al, следует выход из программы.

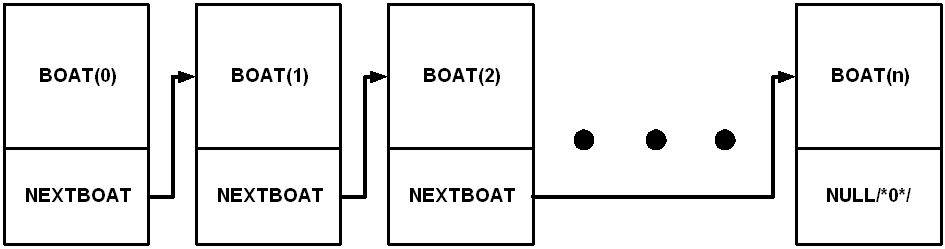
## Динамическое распределение памяти. Связанные списки.

Связанные списки зачастую являются наилучшим выбором при создании алгоритмов, эффективно использующих память. Во всех приведенных выше примерах, работающих с массивами структур, определялось общее количество структур. Например, константа MAX\_BOATS устанавливалась равной 50. Это означает, что программа может обрабатывать данные для 50 катеров максимум. Если в реестре появятся 70 или 100 катеров, то для работы с таким количеством эту программу нужно модифицировать. Это вызвано тем, что память для структур выделяется статически (не нужно путать с модификатором класса памяти static). В этом контексте понятие "статически" относится к переменной, которая создается транслятором на этапе компиляции. Такие переменные существуют в своей области действия; при выполнении программы нельзя их уничтожить или создать еще несколько подобных переменных. Поэтому сразу же видны недостатки статического распределения памяти.

Один из способов решить данную проблему — задать количество структур больше, чем требуется. Если константа MAX\_BOATS будет равна 5000, то это превысит количество катеров в любой компании. Однако, при этом компьютер должен выделить памяти в 100 раз больше, чем ранее; поэтому такой подход при программировании неэффективен.

Более приемлемый подход — выделять память динамически, по необходимости. При этом память для хранения структур запрашивается по мере роста списка. Для динамического распределения памяти можно использовать связанные списки.

Связанный список — это набор структур, каждая из которых имеет некоторый элемент, или указатель, ссылающийся на другую структуру в этом списке. Указатель служит для связи между структурами. Эта концепция напоминает массив, однако она позволяет динамически увеличивать список. На рисунке показан простой связанный список для программы, работающей с реестром катеров фирмы Nineveh Boat Sales.



Реализация стандартного связанного списка

Связанный список для данного примера включает указатель на следующий катер в реестре:

struct stboat {

char sztype[15];

char szmodel[15] ;

char sztitie[20] ;

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

struct stboat \*nextboat;

} Nineveh, \*firstboat,\*currentboat;

Пользовательский структурный тип stboat называется ссылочной структурой, поскольку он имеет поле, содержащее адрес другой, аналогичной структуры. Указатель nextboat (следующий катер) хранит адрес следующей связанной структуры. Это позволяет указателю \*nextboat первой структуры ссылаться на вторую структуру и так далее. Такова концепция связанного списка структур.

### Особенности использования связанных списков.

Для того чтобы прикладная программа динамически отслеживала размер данных, необходимо иметь средства для выделения памяти каждому новому элементу, добавляемому в список. В С для выделения памяти используется функция malloc(); в C++ — new(). В следующем параграфе программа выделяет память для первой структуры при помощи оператора

firstboat=(struct stboat \*) new (struct stboat);

Следующий фрагмент программы показывает, как аналогичный оператор можно применить для последующего выделения памяти каждой дополнительной структуре. Процесс продолжается в цикле while до тех пор, пока имеются данные для обработки:

while (datain(&Nineveh) == 0)

{

currentboat -> nextboat = (struct stboat \*) new (struct stboat);

if (currentboat -> nextboat == NULL) return(1);

currentboat = currentboat -> nextboat;

\*currentboat = Nineveh;

}

Для того чтобы продемонстрировать передачу структуры в качестве параметра, в начале цикле while адрес структуры stboat, &Nineveh, передается в функцию datain(). Эта функция заносит в структуру действительные данные или возвращает значение 1, если пользователь нажал символ "Q", означающий окончание ввода. Если функция datain() не возвращает 1, то указателю currentboat->nextboat присваивается адрес динамически созданной структуры stboat. Обратим внимание на то, что выполняется приведение типа (struct stboat \*) адреса, возвращаемого функцией new(), для того, чтобы согласовать его с типом принимающей переменной. Оператор if проверяет, был ли успешным вызов функции new(). (При ошибке new() возвращает значение null.)

Назначение переменной currentboat — хранить адрес последней действительной структуры stboat в списке; поэтому оператор, следующий за if, присваивает переменной currentboat адрес нового элемента списка, а именно — новый адрес nextboat.

Последний оператор в цикле копирует содержимое структуры Nineveh типа stboat в новую динамически созданную структуру, на которую ссылается указатель \*currentboat. Указатель в последней структуре списка установлен на null — это означает конец связанного списка. Найдите соответствующее место в приведенной ниже программе.

### Простой связанный список.

В следующей программе показано, как реализовать пример с реестром катеров фирмы Nineveh Boat Sales при помощи связанных списков. Сравните эту программу с той, что была приведена ранее в теме "Создание массива структур". Пример на С аналогичен этому, за исключением того, что в нем реализован статический массив. Проанализируйте оба листинга и определите, какие элементы изменились, а какие нет.

// 14LIST.CPP

// Программа на С++ - простой пример связанного списка

#include "stdafx.h"

#include <stdio.h>

#include <strstream>

#include <iostream>

using namespace std;

//#include <stdlib.h>

//#include <iostream.h>

struct stboat

{

char sztype[15];

char szmodel[15];

char sztitle[20];

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fholesale;

struct stboat \*nextboat;

} Nineveh, \*firstboat, \*currentboat;

void boatlocation(struct stboat \*node);

void output\_data(struct stboat \*boatptr);

int datain(struct stboat \*Ninevehptr);

int main(void)

{

firstboat = (struct stboat \*) new (struct stboat);

if(firstboat == NULL) exit(1);

if(datain(&Nineveh) != 0) exit(1);

\*firstboat = Nineveh;

currentboat = firstboat;

while(datain(&Nineveh) == 0)

{

currentboat -> nextboat = (struct stboat \*) new (struct stboat);

if(currentboat -> nextboat == NULL) return(1);

currentboat = currentboat -> nextboat;

\*currentboat = Nineveh;

}

currentboat -> nextboat = NULL; // указывает на конец списка

boatlocation(firstboat);

return(0);

}

void boatlocation(struct stboat \*node)

{

do

{

output\_data(node);

} while ((node = node -> nextboat) != NULL);

}

void output\_data(struct stboat \*boatptr)

{

cout << "\n\n\n";

cout << "The model year for the boat: " << boatptr -> iyear << "\n"

<< "The make of the boat: " <<boatptr -> sztype << "\n"

<< "The model of the boat: " << boatptr -> szmodel << "\n"

<< "The number of hours on the boat motor: " << boatptr -> lmotor\_hours << "\n";

cout << "The comment about the boat: " << boatptr -> szcomment << ".\n";

cout << "Grab the deal by asking your Nineveh selesperson for ";

cout << boatptr->sztitle << " ONLY! $"

<< boatptr -> fretail << ".\n";

}

int datain(struct stboat \*Ninevehptr)

{

char newline;

cout << "\n[Enter new boat information - a Q quits]\n\n";

cout << "Enter the make of the boat.\n";

cin >> Ninevehptr -> sztype;

if(\*(Ninevehptr -> sztype) == 'Q') return (1);

cout << "Enter the model of the boat.\n";

cin >> Ninevehptr -> szmodel;

cout << "Enter the title number for the boat.\n";

cin >> Ninevehptr -> sztitle;

cout << "Enter the model year for the boat.\n";

cin >> Ninevehptr -> iyear;

cout << "Enter the number of hours on the boat motor.\n";

cin >> Ninevehptr -> lmotor\_hours;

cout <<"Enter the retail price of the boat.\n";

cin >> Ninevehptr -> fretail;

cout << "Enter the wolesale price of the boat.\n";

cin >> Ninevehptr -> fholesale;

cout << "Enter a one line comment about the boat.\n";

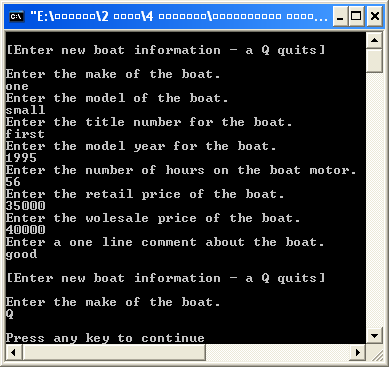
cin.get(newline);

cin.get(Ninevehptr -> szcomment,80,'.');

cin.get(newline); // Обрабатывает возврат коретки

return(0);

}



Обратите внимание на то, что указатели на структуру stboat передавались всем трем функциям:

void boatlocation(struct stboat \*node);

void output\_data(struct stboat \*boatptr);

int datain(struct stboat \*Ninevehptr);

Функция boatlocation() проверяет наличие данных в связанном списке, после чего вызывает функцию output\_data(). Это происходит в цикле do...while, который заканчивается тогда, когда указателю node присваивается null-адрес. Это происходит только при попытке обращения к элементу, следующему за последней структурой stboat в списке. Функция output\_data() осуществляет форматированный вывод каждой структуры из связанного списка.

Во многих языках высокого уровня связанные списки позволяют создавать программы, весьма эффективно использующие память; зачастую эти программы — самые сложные для отладки. Однако, как вы узнаете далее, объектно-ориентированные классы C++ обеспечивают еще большую эффективность.

**Что вы должны были узнать.**

Как определять символьные константы директивой #define:

#define FINGERS 10

Как включать другие файлы: #include "albanian.h"

Как определить макрофункцию: #define NEG(X) (-(X))

Когда использовать символические константы: часто.

Когда использовать макрофункции: иногда.

Опасность применения макрофункций: побочные эффекты.

**Вопросы и ответы.**

**Вопросы.**

1. Ниже приведены группы операторов, содержащих по одному и более макроопределений, за которыми следуют строки исходных кодов, использующих эти макроопределения. Какой результат получается в каждом случае? Правилен ли он?
   1. #define FPM 5280 /\* футов в миле \*/

dist = FPM \* miles;

* 1. #define FEET 4

# define POD FEET + FEET

plort = FEET \* POD;

* 1. # define SIX = 6;

nex = SIX;

* 1. # define NEW(X) X + 5

у = NEW(y);

berg = NEW(berg) \* lob;

est = NEW(berg) / NEW(y);

nilp = lob \* NEW(-berg);

1. Подправьте определение в вопросе 1.г, чтобы сделать его более надежным.
2. Определите макрофункцию, которая возвращает минимальное из двух значений.
3. Задайте макроопределение, в котором есть функция whitesp(c), считающая в программе пустые символы. Определите макрофункцию, которая печатает представления и значения двух целых выражений.

**Ответы.**

* 1. dist = 5280 \* miles; правильно.
  2. plot = 4 \* 4 + 4; правильно, но если пользователь на самом деле хотел иметь 4 \* (4 + 4), то следовало применять директиву #define POD (FEET + FEET).
  3. nex == 6; неправильно; очевидно, пользователь забыл, что он пишет для препроцессора, а не на языке Си.
  4. у = у + 5; правильно, berg = berg + 5 \* lob; правильно, но, вероятно, это нежелательный результат.

est = berg + 5/y + 5; то же самое.

nilp = lob-berg + 5; то же самое.

1. #define NEW(X) ( (X) + 5)
2. #define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )
3. #define WHITESP(C) ( (C) == ' ' || (C) == '\n' || (C) == '\t')
4. #define PR2(X,Y) printf("X равно %d и Y равно %d.\n", X, Y)

Так как в этом макроопределении X и Y никогда не используются никакими другими операциями (такими, как умножение), мы не должны ничего заключать в скобки.

**Упражнение.**

Создайте заголовочный файл определений препроцессора, которые вы хотите применять.