Лекция 11

Препроцессор языка «С».

Директивы препроцессора.

Модули и модульное программирование

Препроцессор

Препроцессор – программа, осуществляющая обработку текста программы перед ее непосредственной компиляцией. Обработка осуществляется согласно специальным указаниям, называемым директивами препроцессора.

Формат записи директивы имеет вид:

#директива *параметры директивы*

Препроцессор

Описание директивы препроцессора всегда начинается с новой строки и заканчивается в конце строки.

Поэтому:

- *на одной строке может быть записана только одна директива.*
- *одна директива может быть записана в нескольких строках. Для этого на конце каждой строки (кроме последней) ставится символ ‘\’.*

Пример:

***#директива** описание директивы *
*продолжение описания директивы *
завершение описания директивы

Директива include

Директива include осуществляет вставку в программу текста из другого файла. В основном эта директива используется для подключения заголовочных файлов.

Имя файла указывается заключенным в знаки ‘<‘ и ‘>’, если файл находится в каталоге **include** среды разработки. Как правило в настройках среды разработки можно указать перечень каталогов которых будет производится поиск.

//Подключение файла *stdio.h* из каталога **include**

#include <stdio.h>

//Подключение файла *types.h* из подкаталога *sys* каталога *include*

#include <sys\types.h>

Директива include

Имя файла указывается заключенным в двойные кавычки, если файл находится в произвольном месте, а не в стандартном каталоге.

В таком случае, если файл находится в текущем каталоге (там же, где и программа), то указывается только имя файла. Например:

#include “data.h”

Если файл находится не в текущем каталоге, то можно указать относительное или абсолютное имя файла. Например:

#include “data\data.h” //В подкаталоге **data** текущего каталога

#include “..\data.h” //В родительском каталоге текущего каталога

#include “e:\data\data.h” //В каталоге **data** на диске **e**.

#include “\data\data.h” //В каталоге **data** на текущем диске.

Директива define

Директива define предназначена для проведения замен и создания макросов.

Создание автозамен:

#define *идентификатор* строка-подстановка *перевод_строки*

Примеры:

#define *PI 3.14159265358979323*

#define *expr pow(x, 2+3*y)*

Использование:

```
double  x = 2.0*PI, y = 2.0;
double res = expr;
```

Директива define

Создание макросов

#define *идентификатор([параметр-идентификатор[, …]])*
строка-подстановка *перевод_строки*

Пример

#define sqr(x) pow(x,2.0) sqr(a) → pow(a,2.0)

Неправильное объявление макроса:

#define sqr(x) x * x

Использование: sqr(x+1) → x + 1 * x + 1 //2*x+1

Правильное объявление макроса:

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

Использование: sqr(x+1) → (x + 1) * (x + 1) //(x+1)^2

Директива define

Параметр макроса может быть преобразован к строке (stringizing). Для этого используется специальный формат записи параметра макроса в описании реализации: #имя_параметра

Пример:

#define prn(str) puts(#str)

Использование: prn(hello!) → puts(“hello!”)

Пример:

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

```
int value = 10;
printval(value);        //printf(“value=%d\n”,value) → value=10
```

Директива define

Можно создавать макросы в которых параметр становится частью лексемы программы (token-pasting). Для этого в описании реализации макроса к параметру обращаются в формате: ##имя_параметра.

При этом действует следующее ограничение: этот параметр не может быть отдельной лексемой или являться ее началом.

Примеры:

#define nvar(n) int g_var**##**n = n;

- nvar(1) → int g_var1 = 1;
- nvar(2) → int g_var2 = 2;

Директива define

В макрос можно передавать неограниченное количество параметров. Для этого в описании заголовка макроса в списке параметров указывается троеточие, а обращение к каждому параметру макроса осуществляется с помощью специального идентификатора __VA_ARGS__

Пример:

#define prn_varargs(...) puts(#__VA_ARGS__)

- prn_varargs(0); → puts(“0”);
- prn_varargs(one, two, three); → puts(“one, two, three”);

Пример:

```
#define vars(type,...) type __VA_ARGS__
vars(int,v1,v2,v3);    //int v1, v2, v3
v1 = 1; v2 = 2; v3 = 3;
printf("%d %d %d\n",v1,v2,v3);
```

Директива define

Примеры:

```
#define ver(maj,min)  #maj "." #min
```

char *version=ver(1,0); → char *version="1.0";

```
#define var(type, name, value) type name = value;
```

var (double,val,10.0); → double val=10.0;

```
#define exchange(type,val1,val2) {\
 type tmp = val1;\
 val1 = val2;\
 val2 = tmp;\
}
```

```
int v1 = 1, v2 = 2;
exchange(int,v1,v2);
printf("%d %d\n",v1,v2);
double x = 1.0, y = 2.0;
exchange(double,x,y);
printf("%lf %lf\n",x,y);
```

Директива define

Автозамены, макросы и простые определения, сделанные с помощью директивы #define можно отменять с помощью директивы #undef. Затем их можно снова определить.

Пример:

```
#define prn puts("One!")
  prn;  //puts(“One!”);
#undef prn
#define prn puts("Two!")
  prn; //puts(“Two!”);
#undef prn
  prn;  //Ошибка – prn не определено
```

Директива error

Директива error используется для создания сообщения об ошибке во время компиляции.

Формат:

#error *строка_описания_ошибки*

Пример:

```
#define prn puts("One!")
  prn;
#undef prn
#ifndef prn
  #error prn must be defined!
#endif
  prn;
```

Директива pragma

Директива pragma осуществляет указание некоторых особенностей компилятору.

- **#pragma optimize(** [{ **time** | **size** | **none** }] **)**
- **#pragma message(** *string* **)**
- **#pragma startup** *function*
- **#pragma exit** *function*
- **#pragma code_seg(** [**"***name***"**] **)**
- **#pragma data_seg(** [**"***name***"**] **)**
- **#pragma const_seg(** [**"***name***"**] **)**
- **#pragma once**

Директива pragma

```
void start(void)
{
  printf("START function!\n");
}

void finish(void)
{
  printf("FINISH function!\n");
}

#pragma startup start
#pragma exit finish

int main(int argc, char *argv[])
{
  printf("MAIN function!\n");
  return 0;
}
```

Результат:
START function!
MAIN function!
FINISH function!

Директива pragma

```
void start1(void)
{
  printf("START1 function!\n");
}

void start2(void)
{
  printf("START2 function!\n");
}

void finish1(void)
{
  printf("FINISH1 function!\n");
}

void finish2(void)
{
  printf("FINISH2 function!\n");
}
```

```
#pragma startup start1
#pragma exit finish1
#pragma startup start2
#pragma exit finish2

int main(int argc, char *argv[])
{
  printf("MAIN function!\n");
  return 0;
}
```

Результат:

```
START2 function!
START1 function!
MAIN function!
FINISH2 function!
FINISH1 function!
```

Директивы условной компиляции

1	2	3	4
#if условие *операторы* #endif	#if условие *операторы 1* #else *операторы 2* #endif	#if условие1 *операторы 1* #elif условие2 *операторы 2* #endif	#if условие1 *операторы 1* #elif условие2 *операторы 2* #else *операторы 3* #endif

Директивы условной компиляции

Примеры:

```
#define A 5
#if A>10
  puts(“Message 1”);
#else
  puts(“Message 2”);
#endif
```

```
#define A 2
#if A==1
  puts("A=1!");
#elif A==2
  puts("A=2!");
#else
  puts("uncknown A!");
#endif
```

Директивы условной компиляции

```
#ifndef идентификатор
…
#endif
```

```
#ifdef идентификатор
…
#endif
```

Директивы условной компиляции

Примеры:

```
#define DEBUG_MODE
…
#ifdef DEBUG_MODE
  puts(“Режим отладки”);
#endif
```

```
#define TEST_MODE
int main(int argc, char *argv[])
{
#ifdef TEST_MODE
  freopen("in.txt","r",stdin);
  freopen("out.txt","w",stdout);
#endif
  int val;
  scanf("%d",&val);
  printf("%d",val);
  return 0;
}
```

Диагностика

Библиотека <assert.h>

void assert(int выражение)

Отмена действия – определить имя NDEBUG до подключения библиотеки assert.h

Диагностика

Ввести два целых числа. Разделить первое число на второе. Если второе число – ноль, то остановить программу используя макрос assert.

```
#include <stdio.h>
#include <assert.h>
int main(int argc, char *argv[])
{
  int a,b;
  printf("Puts numers A and B: ");
  scanf("%d %d",&a,&b);
  assert(b!=0);
  printf("%d\n", a/b);
  return 0;
}
```

Модуль

Модуль (библиотека) – совокупность типов данных, переменных, констант и функций для работы с этими типами данных.

Основное предназначение:

- повторное использование разработанного ранее кода,
- улучшение процесса разработки программ.

Структура модуля

Две основные части:

- интерфейс (заголовок модуля – файл **.h**);
- реализация (реализация модуля файл **.c**).

Заголовок модуля

Заголовок модуля – интерфейсная часть, представленная в виде файла с расширением **.h**.

Основное содержание:

- описание внешних типов данных;
- описание внешних переменных и констант;
- описание прототипов внешних функций.

Реализация модуля

Реализация модуля – файл с расширением **.c**

Основное содержание:

- описание внутренних типов данных;
- описание внутренних и внешних переменных и констант;
- реализация внешних и внутренних функций.

Правила описания внешних переменных

Объявление внешней переменной с возможной ее инициализацией осуществляется в файле текста программы модуля, а в файле заголовков такая переменная описывается как внешняя (класс памяти *extern*) без какой-либо инициализации.

Подключение модулей

Подключение модуля в программу осуществляется двумя действиями:

- подключение файла заголовка модуля с помощью директивы **#include**;
- подключение файла текста программы модуля в проект.

Частные случаи модулей

1. Модуль содержит только часть реализацию: единственный модуль в программе, содержащий только функцию **main**.

2. Модуль содержит только часть заголовок: в модуле производится описание глобальных типов данных.

Использование условной компиляции

При многократном подключении модуля необходимо организовать однократность его компиляции. Это осуществляется с использованием директив условной компиляции в файле заголовка модуля в формате:

#ifndef имя_модуля
#define имя_модуля
… текст заголовка модуля …
#endif
ИЛИ
#pragma once