Тема 7

Дополнительные возможности

Директивы препроцессора

Обработка исходных текстов программ компилятором в C++ выполняется в два этапа: препроцессорная обработка и собственно компиляция, т.е. построение объектного кода. На первом этапе происходит преобразование исходного текста, а на втором компилируется уже преобразованный текст. Препроцессор обрабатывает собственные *директивы*. Эти директивы задаются, как правило, в отдельной строке и начинаются с символа '#'.

Директивы препроцессора (продолжение)

Различают следующие директивы препроцессора:

- директивы компилятора **#pragma**, указывающие компилятору, как именно необходимо строить объектный код;
- директива включения файла **#include**, с помощью которой можно включить в текст программы текст из других файлов;
- директивы условной компиляции **#if, #else, #elif, #endif, #ifdef, #ifndef, defined;**
- директива определения лексем #**define;**
- директива отмены определения #**undef**

Директива #include

Различают следующие формы директивы #include

- **#include** <имяфайла>
- **#include** ”имяфайла”

Первый формат позволяет включить имя файла из тех папок, которые заданы как стандартные для хранения включаемых файлов. Конкретное местоположение этих папок зависит от реализации. Второй формат дает возможность записать произвольное имя файла в терминах операционной системы.

Допускается вложенное применение директивы **#include**

Директива #define

Директива #define позволяет описывать новые **лексемы**. Её формат:

```
#define имя_лексемы [(параметры)] значение_лексемы
```

- В дальнейшем в тексте программы все встреченные лексемы будут заменены соответствующим текстом.
- Текст лексемы должен заканчиваться символом конца строки. Поэтому для задания длинных и сложных лексем можно воспользоваться возможностью **склейки** строк: если в конце строки стоит символ '\', он считается признаком переноса на следующую строку.

Пример склейки строк для директивы #define

```
#define va_arg(list, mode) \
 *(((list).offset += ((int)sizeof(mode) + 7) & -8) , \
 (mode *)((list).a0 + (list).offset - \
(__builtin_isfloat(mode) && (list).offset <= (6 * 8)) \
  ? (6 * 8) + 8 : ((int)sizeof(mode) + 7) & -8) \
            ) \
        ) \
    )
```

Константы времени компиляции

Определение лексем даёт возможность задать т.н. **константы времени компиляции**. Для таких констант не определяется тип, не выделяется память, что может быть удобно в отдельных случаях. Так, описание

```
#define MAX_LENGTH 50
```

более понятно, на мой взгляд, чем

```
const int MAX_LENGTH = 50;
```

С другой стороны, при передаче этой константы в качестве параметра функции возможно, потребуется явно задать тип:

```
s = my_func((int) MAX_LENGTH);
```

Макросы

Директива #define с параметрами называется **макросом**.

Такие конструкции позволяют выполнить замещение лексем по-разному, в зависимости от фактических параметров. Иногда использование макросов оказывается более полезным, чем задание множества перегруженных функций.

Пример макроса

```
#define max(x, y) ((x)>(y) ? (x) : (y))
```

Запись в исходном тексте программы

```
cout << max(a+5, d-4);
```

преобразуется в

```
cout << ((a+5) > (d-4) ? (a+5) : (d-4));
```

Это будет правильно для всех типов, для которых определена операция ”>” , и её использование правомерно.

Если a и d – указатели на нуль-терминированные строки, использовать этот макрос нельзя!

Пример неправильного макроса

Обратите внимание, что параметры в теле макроса max взяты в скобки. Это сделано для того, чтобы избежать ошибок в определении порядка операций.
Классический пример

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

приводит к тому, что при вызове макроса

```
sqr(a+2)
```

получается неверное (хотя и синтаксически правильное) выражение

```
a+2 * a+2
```

Правильная запись такого макроса:

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

Директивы #undef и defined

Директива

```
#undef имя_лексемы
```

отменяет определение лексемы, заданное директивой **#define**. Наконец, директива

```
defined(имя_лексемы)
```

дает возможность выяснить, определена ли указанная лексема. Заметим, что последняя директива не записывается в отдельную строку и применяется только в других директивах препроцессора (например, в директивах условной компиляции).

Директивы условной компиляции

Директивы условной компиляции дают возможность включить в исходный текст те или иные строки, в зависимости от значения выражения (которое должно быть выражением времени компиляции), например:

```
#define DEBUG_MODE 1
// по завершении отладки 1 будет заменена на 0
…
#if defined(DEBUG_MODE) && DEBUD_MODE==1
    //вывод отладочной информации
#endif
```

Можно и так:

```
#define DEBUG_MODE
//эта строка будет закомментирована
…
#ifdef DEBUG_MODE
    //вывод отладочной информации
#endif
```

Проблема повторного определения

Поскольку язык C/C++ допускает вложенное использование директивы **#include**, может возникнуть ситуация, когда программист, сам того не желая, включит в свои исходные тексты одно и то же определение несколько раз.

Пример:

a.h	b.h	c.h

…
void my_funcA() {
…
}
…
	#include "a.h"
	…
		#include "a.h"
		…

При одновременном включении файлов "b.h" и "c.h" функция my_funcA() будет определена дважды!

Страж включения

Необходимо организовать проверку повторного определения (написать **страж включения**) в файле ”a.h”:

```
#ifndef _my_funcA_defined_
#define _my_funcA_defined_
void my_funcA() {
…
}
#endif /* _my_funcA_defined_ */
```

Имя идентификатора, используемого в страже включения, должно быть подобрано так, чтобы оно случайно не совпало ни с одним из других идентификаторов программы!

Другой способ задания стража включения – использование директивы

```
#pragma once
```

Пространства имён

Пространства имён (называемые также поименованными областями) служат для логического группирования объявлений и ограничения доступа к ним.

Пространство имен объявляется с помощью оператора

```
namespace имяпространства { объявления }
```

В операторе namespace могут присутствовать не только объявления, но и определения программных объектов (тела функций, инициализаторы переменных и т.д.).

Пространства имён (продолжение)

Если имя пространства имён не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, отдельного для каждого программного модуля. Такое пространство будем считать безымянным. В безымянное пространство имён входят также все глобальные объявления.

Поименованная область может объявляться неоднократно, причем последующие объявления рассматриваются как дополнения к предыдущим. Более того, пространство имён является понятием, уникальным для всего проекта, а не одного программного файла, так что дополнение пространства имен может выполняться и за рамками одного файла.

Примеры пространств имён

```
namespace NS1 {
 int i=1;
 int k=0;
 void f1(int);
 void f2(double);
}
…
namespace NS2 {
 int i, j, k;
}
…
namespace NS1 {
 int i=2;              // Ошибка – повторное определение
 int k;                // Ошибка – повторное объявление
 void f1(void);        // Перегрузка
 void f2(double);      // А такое повторное объявление
   допустимо!
}
```

Работа с пространствами имён

Все программные объекты, описанные внутри некоторого пространства имен, становятся видимыми вне оператора namespace с помощью операции :: , трактуемой как **операция доступа к области видимости**. При этом можно использовать одноименные переменные из различных пространств и собственные переменные:

```
void my_func() {
  int i = 2 + NS1::i;
...
}
```

Оператор using

Если имя часто используется вне своего пространства, его можно объявить доступным с помощью оператора

```
using имяпространства :: имявпространстве;
```

после чего такое имя можно использовать без явного указания имени пространства, например

```
void my_func() {
  using NS2::k;
  int i = 2 + k;
}
```

Наконец, можно сделать доступными все имена из какого-либо пространства, записав оператор

```
using namespace имяпространства;
```

Приоритеты и конфликты имён

Имена объявленные где-нибудь явно или с помощью оператора using, имеют приоритет перед именами, доступными по оператору using namespace:

```
using namespace NS1;
using namespace NS2;
void my_func() {
…
 int i = 2 + k;         // двусмысленность
 int i = 2 + NS1::k;    // а так можно делать
 int i1 = 2 + j     // j берется из пространства NS2
}
```

Пользовательские типы данных

Программист может в дополнение к стандартным типам данных создавать свои собственные типы для того, чтобы адекватно учесть специфику решаемой задачи. Для создания новых типов может быть использован одиниз следующих способов:

- оператор typedef
- оператор enum
- оператор struct
- оператор class (будет рассмотрен в отдельной теме)
- оператор union (не будет рассматриваться далее)

Оператор typedef

С помощью оператора typedef можно присвоить новое имя уже созданному ранее типу.

Неформальное описание оператора typedef:

- определяем переменную (без инициализации)
- записываем перед этим определением слово typedef

В этом случае идентификатор будет обозначать не имя переменной, а имя нового типа!

Примеры оператора typedef

```
unsigned char byte;
// byte – переменная типа unsigned char
typedef unsigned char byte;
// byte – тип, совпадающий по характеристикам с
// unsigned char
// теперь можно писать
byte b1, b2=200;

typedef int massiv [100];
// massiv – это тип: массив из 100 элементов типа int
massiv m1, m2;
massiv* pm = &m1;
// а так писать нельзя!
massiv* pm = new massiv;

typedef int (*pf) (int, int);
// что такое pf?
```

Оператор enum

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения. Для этого удобно воспользоваться перечисляемым типом данных. Формат оператора enum:

enum [имя_типа] { список_констант }
[список_переменных];

- Если отсутствует имя типа – должен быть список переменных. Больше переменных этого типа создать нельзя!
- Если отсутствует список переменных – переменные этого типа можно создавать позднее!

Примеры оператора enum

```
enum Err {NO_ERR, ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;
…
switch (error) {
 case ERR_READ: /* операторы */ break;
 case ERR_WRITE: /* операторы */ break;
 case ERR_CONVERT: /* операторы */ break;
}

// Можно писать и так:
enum {NO_ERR, ERR_READ, ERR_WRITE, ERR_CONVERT}
       error = NO_ERR;

// задание значений констант
enum Err{NO_ERR=0, ERR_READ=5, ERR_WRITE=10,
  ERR_CONVERT=20};
Err error;
…
cout << error+3;
```

Оператор struct

Структура – составной тип данных (подобно массиву). Однако элементы структуры (поля) могут иметь разные имена и типы

Формат оператора struct:

struct [имя_типа] { описание полей} [список_переменных];

- Если отсутствует имя типа – должен быть список переменных. Больше переменных этого типа создать нельзя!
- Если отсутствует список переменных – переменные этого типа можно создавать позднее!

На самом деле структура – частный случай класса!

Примеры оператора struct

```
struct Student {
  char NZ[8];
  char FIO[40];
  int kurs;
  int grup;
};
Student ivanov = {”1023299”, ”Petr Ivanov”, 1, 10};

// Доступ к полям структуры
cout << ivanov.NZ << endl;
ivanov.kurs = 2;

// указатель на структуру
Student* st1 = new Student;
strcpy(st1->FIO, ”Ivan Petrov”);
```

операция -> - синоним операции *.

```
// можно писать и так:
strcpy((*st1).FIO, ”Ivan Petrov”);
```