Возможности C++. Сложные типы данных. Структуры

Директивы препроцессора

Препроцессор — первая фаза компиляции. На этапе препроцессора выполняются инструкции, называемые директивами. Директивы начинаются со знака #.

#include

Включает содержимое подключаемого файла в тот файл, в котором указана. Используется в двух вариантах:

- `#include <имя файла>` — поиск указанного файла происходит в системных каталогах. Предназначен для подключения модулей стандартных библиотек.
- `#include "имя файла"` — поиск указанного файла осуществляется в каталоге проекта. Если он не найдет, поиск будет продолжен в системных каталогах. Способ используется для создания многофайловых проектов, а также для подключения модулей внешних библиотек.

Как правило, имя подключаемго файла имеет расширение *.h. Такие файлы называют заголовочными (сокращение от header — заголовок). Исключение составляют ряд системных файлов. При этом файлы стандартной библиотеки языка C начинаются с буквы и не имеют расширения, однако можно записывать их и в старом формате. Например, cmath и math.h — одно и то же. Файлы стандартной библиотеки языка C++ расширений не имеют (iostream и др.).

#define

#define определяет макроимя или макроподстановку. Например,

```
#define PI 3.1415926
```

Достоинства макроподстановок:

1. Не нужно следить за типами данных (в отличие от функций).
2. Сокращение кода.

Недостатки макроподстановок:

1. Нужно следить за порядком действий.
2. Ухудшают читаемость кода.

#undef

`#undef` отменяет определение макроимени. Нужна для отмены #define.

Директивы условной компиляции

Директивы условной компиляции похожи по синтаксису на множественное ветвление.

```
#if
#elif
#else
#endif
```

Вместо #if могут использоваться:

- `#ifdef макроимя` — если макроимя определено, то ...
- `#ifndef макроимя` — если макроимя не определно, то ...

Директивы условной компиляции используются для управления компиляцией. Например, существуют так называемые «стражи включения», которые более подробно будут рассмотрены ниже.

#pragma

Директива #pragma предназначена для выполнения целого ряда действий, в том числе подключения библиотек. Использование директивы может различаться для разных компиляторов. Например, в MS Visual Studio директива `#pragma once` выполняет те же функции, что и страж включения, то есть позволяет включать код подключаемого файла только один раз. Пример см. далее.

Предопределенные макросы

Предопределенные макросы — это макроимена, у которых значения известны заранее. Примеры предопределенных макросов:

- __LINE__ — номер строчки в коде.
- __DATE__ — дата компиляции.
- __TIME__ — время компиляции.

См. 6_1_Directives.cpp

Области видимости переменных

Область видимости переменной — часть программы, где эту переменную можно использовать. В C++ выделяют следующие области видимости:

- Переменные, объявленные внутри любого блока, называются локальными. Область видимости локальной переменной -- от места объявления до конца блока. Локальные переменные часто объявляют в заголовках и внутри циклов. Тогда заданное имя существует только внутри цикла, что позволяет использовать его повторно в других циклах в совершенно другом смысле, возможно даже с другим типом данных.
- Переменная, объявленная вне любого блока, называется глобальной. Область видимости -- файл, в котором она объявлена. Не следует злоупотреблять глобальными переменными, т.к. возникают сложности с их передачей в другие программные модули.
- `extern` позволяет подключить глобальную переменную для другого файла. На практике используется редко.
- Параметры функции имеют в качестве области видимости всю функцию. Параметры внутри функции называют формальными, передаваемые при вызове -- актуальные.
- Класс (см. далее).
- Именованная область.

Пространства имен

Именованная область (пространство имен) — удобный способ ограничить использование набора имен в рамках программы. Синтаксис объявления:

```
namespace [имя]
{
     // тут весь код, относящийся к этому пространству
}
```

Синтаксис использования:

- `using namespace имя;` — подключение пространства имен полностью;
- `имя::элемент` — использование отдельного элемента из заданного пространства.

При написании программного кода всегда следует руководствоваться одним из фундаментальных принципов программирования — переменная всегда должна иметь минимально возможную область видимости.

Многомодульные программы

При создании больших проектов на C++ необходимо разделять код на модули. Каждый модуль содержит набор функций и других программных элементов, связанных между собой. При создании модулей рекомендуется разделять логику работы программы и ее интерфейс, то есть делать отдельный модуль для функций, выполняющих полезную нагрузку, и отдельных для интерфейса, использующего эти функции для решения конкретных пользовательских задач. Например, при разработке модуля работы с массивами рекомендуется функции заполнения массива с клавиатуры и вывода массива на экран помещать в отдельном модуле.

В C++ модуль, как правило, состоит из двух файлов. В первом файле с расширением *.cpp находятся функции, а во втором файле с расширением *.h — их заголовки, а также объявления глобальных переменных, констант, структур и т.д. Также в этом модуле помещаются подключения всех используемых файлов. Имена файлов (до расширения) одного модуля должн совпадать. Если имя.cpp — модуль, то имя.h — заголовочный файл, он же интерфейс модуля.

Подключение модулей производится с помощью директивы `#include`

Для подключения модуля с именем name необходимо написать строчку `#include "name.h"`

в файле name.cpp этого модуля, а также в том файле, где производится подключение модуля.

C++ с технической точки зрения позволяет обходиться только файлами *.h, но это противоречит правилам разработки.

Следует помнить, что `#include` полностью копирует код из подключаемого файла. Если `#include "name.h"` встречается в двух файлах (а это произойдет обязательно), то при сборке проекта возникнет конфликт. Для его разрешения используется один из двух способов:

1. В MS Visual Studio используется директива `#pragma once`
2. Более универсальный способ — использование так называемых стражей включения.

Суть метода заключается в использовании директив условной компиляции. Если некоторое макроимя еще не определено, то мы его определяем и далее включаем содержимое нужного файла. Если же макроимя уже определено, подключение выполнено не будет. Для того, чтобы способ работал, достаточно назначить для каждого модуля свое макроимя. Как правило, в этом качестве используется название модуля, написанное большими буквами.

Пример использования стражей включения:

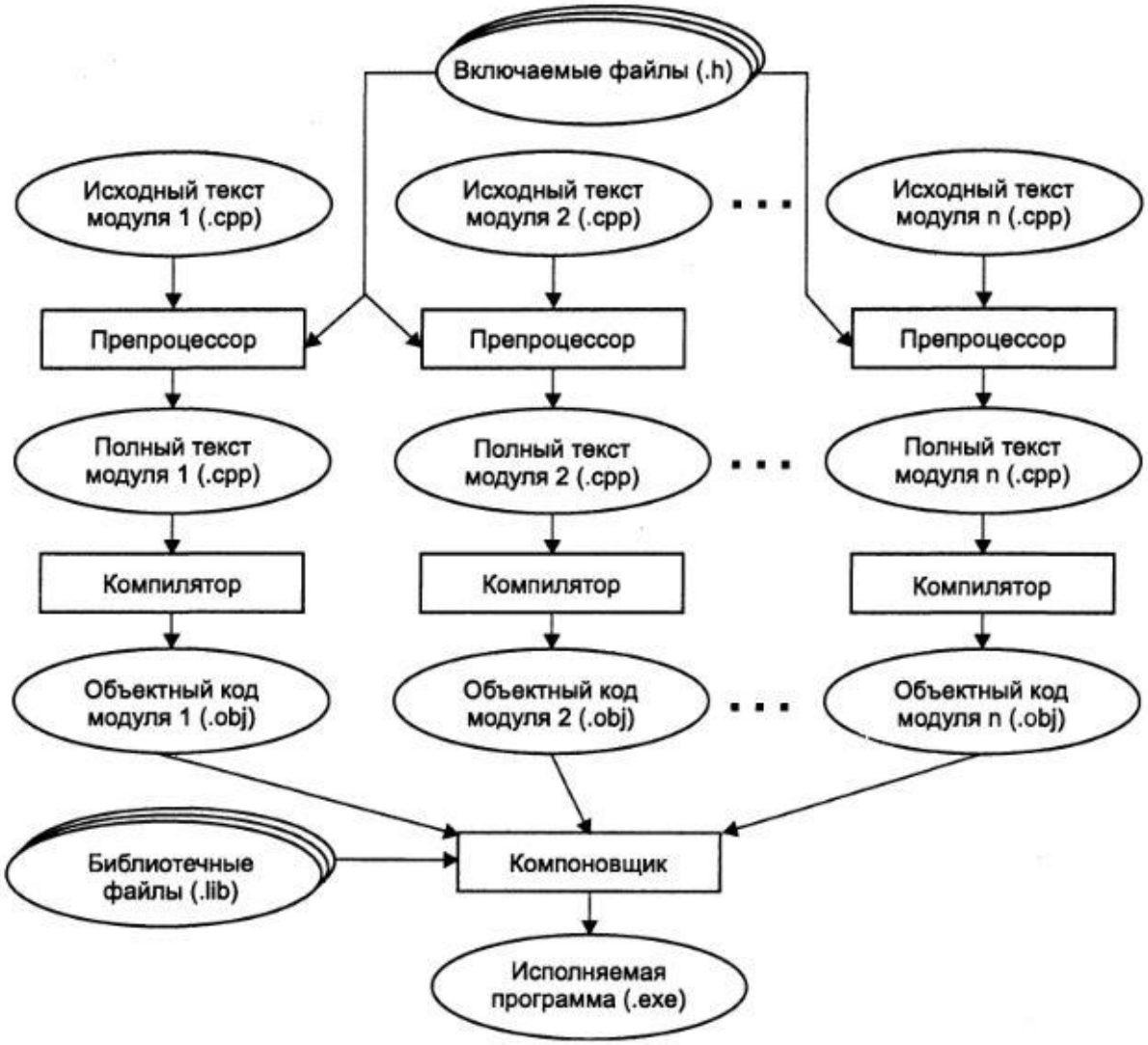
```
#ifndef ИМЯ
#define ИМЯ
#include "имя.h"

// текст файла




#endif
```

Процесс создания исполняемого файла из исходных кодов



Процесс создания исполняемого файла представлен на рисунке выше и включает в себя следующие этапы:

1. Препроцессор — первая фаза компиляции. На этом этапе выполняются директивы препроцессора, в том числе в текст файлов включается код из подключаемых файлов. Далее мы получаем полные тексты программных модулей.
2. Компиляция. На этом этапе полный текст каждого модуля обрабатывается компилятором. Происходит поиск синтаксических ошибок. В случае их отсутствия файлы преобразуются в объектный код.
3. Компоновка (сборка, линковка). На этом этапе компоновщик собирает вместе объектные коды, подключает внешние файлы и в итоге выдает исполняемый файл.

Переименование типов

Для того, чтобы дать типу данных новое имя, используется команда typedef. Синтаксис:

```
typedef тип новое_имя;
```

Переименование типов удобно использовать для создания коротких имен у типов с длинными называниями, а так же при работе со сложными типами данных.

Примеры: переименование типов

```
typedef unsigned int UINT;
typedef char Str[100];
```

Объявление переменных

```
UINT i, j;
Str a, b[10];
```

Перечисления

Перечисления удобны для задания наборов константных значений. Синтаксис

```
enum [имя]{константы};
```

Константам внутри перечисления автоматически последовательно присваиваются значения 0, 1, 2, ... Значения можно задавать и явно с помощью операции присваивания, в таком случае они могут быть любыми целыми числами.

Пример: объявление перечисления

```
enum Err{ERR_READ, ERR_WRITE, ERR_CONVERT};
```

Использование

```
Err error;
switch(error){
    case ERR_READ: //
    //...
}
```

Структуры

Предназначены для хранения связанных наборов данных разных типов. Синтаксис:

```
struct [имя]{
    тип_1 имя_1;
    тип_2 имя_2;
    ...
    тип_n имя_n;
} [переменные];
```

Структуры могут не иметь имен, но в таком случае следует обязательно задавать переменные, т.к. другие переменные объявить уже не получится. Мы будем использовать только структуры с именами.

Обращение к элементам структуры осуществляется через . (точку) или через -> (стрелочку). Стрелочка -> используется для динамических структур данных, это будет изучаться в следующем разделе. Элементы структуры называются **полями** или **атрибутами**.

Метод — встроенная функция структуры. Метод является элементом структуры и объявляется внутри нее. Метод доступен только для переменных, имеющих тип этой структуры.
См. 6_2_Structures.cpp