Программирование на языке C++

Зариковская Наталья Вячеславовна

Лекция 1

Исторические замечания

- Язык С первоначально разрабатывался как язык системного программирования (70е год. Деннис Ритчи). Он был использован для построения ОС UNIX . Постепенно приобретая популярность, он стал языком широкого пользования и получил значительное развитие. Язык С++, поглотивший язык С, воплотил в себе все основные тенденции современного стиля программирования (ООП - объектно-ориентированное программирование) фиксирует поведение реальных объектов таким образом, что детали его реализации скрыты.
- Язык программирования С++ стал развитием языка С по трем основным направлениям :
- 1) поддержка абстракции данных (типов данных, определяемых пользователем);
- 2) поддержка объектно-ориентированного проектирования и программирования;
- 3) различные улучшения конструкций С.
- В то же время язык сохранил все достоинства С (например, переносимость), поскольку тот вошел в него как подмножество.
- К настоящему времени язык С++ является основным языком высокого уровня для создания программного обеспечения, в первую очередь больших программных комплексов. Он реализован на самых разных ЭВМ - от персональных до суперкомпьютеров. На него ориентируется обучение программированию на Западе.

Первое знакомство с языком

- У вас, как и у любых начавших изучать новый язык, неизбежно возникают два вопроса :
- 1) что такое программа на C++ и как ее написать?
- 2) если программа уже написана, как ее оттранслировать и запустить?

Организация программы на C++

- Программа на C++ представляет собой совокупность из одного или нескольких файлов.
- Файл представляет собой обычный текстовый файл с программным текстом. Каждый такой файл компилируется отдельно, а затем полученный объектный код (машинные команды) соединяется воедино с помощью компоновщика (редактора связей). Подробнее о процессе компиляции мы поговорим на лабораторных и практических занятиях.
- Программа на языке C++ состоит из файлов двух типов: файлов заголовков и файлов кода. Файлы заголовков имеют расширение «.h», а файлы кода - расширение «.c» или «.cpp». Файлы заголовков содержат классы, шаблоны, структуры, объединения, перечисления, объявления функций, описания typedef , определения констант, функции inline и директивы препроцессора.
- При компиляции, объявления заголовочных файлов включаются в любой файл, где исполь зуются внешние функции, классы и объекты, содержащиеся в нём. Заголовочные файлы включаются с помощью директивы препроцессора #include, которая имеет две формы:
- #include "имя заголовочного файла"
- #include <имя заголовочного файла>

Организация программы на C++

- Если имя заголовочного файла указано в кавычках, то его поиск осуществляется в текущем каталоге пользователя.
- Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы.
- Файлы кода содержат реализацию программы пользователя и имеют расширение «.c» или «.cpp». Они состоят из одной или более функций.
- Одна из функций, с которой начинается выполнение программы, должна иметь имя main. Функция main отличается от других функций тем, что её нельзя вызывать изнутри программы, а её параметры задаются операционной системой. Параметры в функции main могут отсутствовать. Вызов функции (активизация) выполняется двумя способами: классическим - по имени; косвенно - через указатель на функцию (тема ‘указатели’). Классический способ вызова функции производится при помощи указания имени в скобках, за которым указывается список аргументов - параметров.
- Список аргументов - параметров представляет собой значения (переменных, констант, указателей - констант или адресов), которые необходимо передать функции для успешного решения задачи. Функция активизируется всегда, когда управляющая программа встречается с именем функции. После выполнения соответствующей функции управление передаётся обратно в вызывающую среду (за исключением особых ситуаций), которая продолжает свою работу.

Организация программы на C++

- Функция в C++ это логически самостоятельная именованная часть программы, состоящая из нуля (в простейшем случае) или более операторов, объединённых в исполнимый модуль для решения определённой задачи. Стандарт языка определяет следующий формат определения функции:
- **[тип] имя_функции (сп. форм. пар.| void)**
- **{**
- **[описание данных]**
- **[операторы тела функции]**
- **[return] [выражение]**
- **},**
- где спецификация тип - задаёт тип возвращаемого функцией значения. Если указание типа отсутствует, то считается, что функция возвращает значение int. Если вместо типа стоит клю чевое слово void, то считается, что функция не возвращает в вызывающую программу никакого значения. Особый случай, когда используется тип void * - родовой указатель. В этом случае результат работы функции есть указатель.

Организация программы на C++

- Имя_функции - идентификатор произвольного вида, являющийся указателем на функцию, значение которого равно адресу точки входа в функцию.
- Список формальных параметров - это последовательность объявлений формальных парамет ров, разделённых запятыми. В C++ допускается использование функций без формальных параметров. Такой случай возникает, когда в функцию не передаются никакие аргументы. Тогда, поле формальных параметров может быть пустым, или содержать ключевое слово void. Допускается определение формальных параметров по умолчанию.
- При вызове функции с формальными параметрами компилятор вставляет в код программы последовательность машинных команд, обеспечивающих запись, перечисленных в списке аргументов в стек.
- Описание данных и операторы тела функции представляют собой последовательность опера торов, обеспечивающих решение функциональной задачи.

Организация программы на C++

- Оператор return - необязательный оператор, обеспечивающий выход из функции. Если оператор return используется совместно с выражением, то выход из функции сопровождается передачей вычисленного значения в точку вызова. Тип результата должен совпадать или быть совместимым с типом функции. При отсутствии оператора return выход из функции происходит после выполнения последнего оператора в теле функции.

Организация программы на C++

- **Пример простейшей программы, состоящей только из одной функции, например, может быть таким:**

```
#include<stdio.h>
void main()
{int a,b,c;
a=5;b=7;
c=a+b;
printf("sum=%d\n",c);
}
```

Разберем эту программу по порядку.

Первой строкой нашей программы является

#include<stdio.h>

Строка #include называется командой препроцессора (или просто макрокомандой). В результате этой команды содержимое файла stdio.h будет помещено в наш исходный файл. К препроцессору мы еще вернемся .

Файл stdio.h называется заголовочным файлом (header-файлом). В нем содержится информация о printf, которая необходима нашей программе.

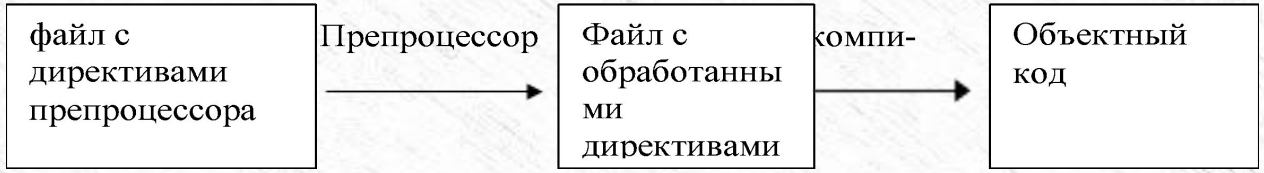
stdio.h - стандартный заголовочный файл, который позволяет нашей программе пользоваться средствами библиотеки ввода-вывода. Каждая реализация C++ содержит, кроме транслятора, набор стандартных библиотек. Такая библиотека - это набор заранее оттранслированных функций, которые программист может использовать точно так же, как и функции, написанные им самим.

Стандартные заголовочные файлы содержат предварительные объявления функций или других средств из этих библиотек. Для того, чтобы пользоваться такой функцией, необходимо подключить в программу файл с ее объявлением. После этого можно вызывать функцию из стандартной библиотеки таким же образом, как и нашу функцию print(). Кроме функций, можно использовать определенные в библиотеке переменные, типы и др.

Заголовочные файлы обычно имеют расширение .h или .hpp. В документации содержатся сведения об стандартных заголовочных файлах для данной реализации и о функциях, описанных в них. Некоторая минимальная совокупность заголовочных файлов с набором функций входит в стандарт языка C++. Программист тоже может создавать и подключать свои заголовочные файлы с объявлениями своих функций.

Начальные сведения о препроцессировании

- Как вы уже видели, макрокоманда #include делает заголовочный файл частью программы пользователя. Макрокоманды начинаются с символа #, перед которым и после которого могут находиться пробелы (но это не принято).
- Макрокоманды обрабатываются до запуска собственно компилятора языка. Обрабатывающая их программа называется препроцессором, а сами макрокоманды еще называются директивами препроцессора.
- Общая схема обработки программы на C++ такова :



Директивы препроцессора

- Директивы препроцессора представляют собой инструкции, записанные в тексте программы на СИ, и выполняемые до трансляции программы. Эти директивы обычно помещаются в начале исходного текста, но синтаксис C++ не запрещает их появлению в любой отметке программы. Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т.п. Любая команда начинаются со знака # принимается как директива предварительной обработки, если она не находится внутри символьного литерала, в символьной константе, или вложена в комментарий. После директив препроцессора точка с запятой не ставятся. Препроцессор Borland C ++ поддерживает следующие директивы препроцессора:
- # (null directive) #ifdef
- #define #ifndef
- #elif #include
- #else #line
- #endif #pragma
- #error #undef #if

Директива #define

- Директива #define служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы.
- Директива #define имеет две синтаксические формы:
-
- **#define идентификатор текст**
- **#define идентификатор (список параметров) текст**
-
- Эта директива заменяет последующие вхождения этого идентификатора на текст. Такой процесс называется макроподстановкой. Текст может представлять собой любой фрагмент программы на СИ, а также может и отсутствовать. В последнем случае все экземпляры идентификатора удаляются из программы.

Директива #define

- **Примеры:**
- # define ttext "good "
- # define yes 1
- .....
- puts(ttext); /*расширяется в puts("good") но не внутри,   комментария */
- .....
- if (i = = yes) /* расширяется в теле if как if (i = =1) */
- Пример директивы с использованием макро с параметрами
- общий вид
- # define идентификатор_макро тело_макро
- В теле Макро формальные параметры должны записываться в скобках
- # define cub (x) ((x)*(x)*(x))
-
- int i =4,j;
- j =cub(i); /* расширяется в j =((i)*(i)*(i))*/
- j = cub(i+1); /* расширяется в j =((i+1)*(i+1)*(i+1))*/
- Но если бы ошибочно # define cub (x) (x*x*x)
- то имели бы   j =cub (i+1); /* j =x+1*x+1*x+1*/

Директива #define

- При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров.
- **Пример:**
- #define MAX(x,y) ((x)>(y))?(x):(y)
- Эта директива заменит фрагмент
- t=MAX(i,s[i]);
- на фрагмент
- t=((i)>(s[i])?(i):(s[i]);
- Как и в предыдущем примере, круглые скобки, в которые заключены формальные параметры макроопределения, позволяют избежать ошибок связанных с неправильным порядком выполнения операций, если фактические аргументы являются выражениями.
- Например, при наличии скобок фрагмент
- t=MAX(i&j,s[i]||j);
- будет заменен на фрагмент
- t=((i&j)>(s[i]||j)?(i&j):(s[i]||j);
- а при отсутствии скобок - на фрагмент
- t=(i&j>s[i]||j)?i&j:s[i]||j;
- в котором условное выражение вычисляется в совершенно другом порядке.

Условные директивы #if, #elif, #else, and #endif

- Условные директивы **#if, #elif, #else,** и **#endif** работают подобно обычным условным операторам C.
-
- #if константное-выражение-1
- <секция-1>
- <#elif константное-выражение -2 newline секция-2>
- …….
- <#elif константное-выражение -n newline секция-n>
- <#else <newline> final- секция>
- #endif

Условные директивы #if, #elif, #else, and #endif

- Если константное-выражение-1 отлично от нуля (истина), выполняются операторы препроцессора представленные -1 секцией и управление передаётся директиве #endif (который заканчивает эту условную последовательность) и выполнение продолжается со следующей секции. Иначе, если константное выражение-1 равно нулю (ложь), секция -1 игнорируется и управление передаётся к следующей директиве #elif (или другой) где константное-выражение -2 истинно. Если истинно, секция -2 выполняется, после чего управление передаётся директиве #endif. Иначе, если константное-выражение -2 ложно, управление передаётся следующему #elif, и так далее, пока не будет достигнута директива #else или #endif. Необязательная директива #else используется как альтернативное состояние, для которого все предыдущие испытания были ложными.
- Условную последовательность заканчивает директива #endif.
- Обработанная секция может содержать вложенные директивы произвольной глубины, а каждый #if должен быть согласован с #endif.

Директивы #ifdef и #ifndef

- Условные директивы #ifdef и #ifndef позволяют выполнить проверку, определен ли идентификатор в настоящее время или нет. Директива
- **#ifdef идентификатор**
- имеет значение 1(истина) если идентификатор в настоящее время определен, и 0(ложь) если идентификатор в настоящее время не определён.
- Директива
- **#ifndef идентификатор**
- имеет значение 0(ложь) если идентификатор в настоящее время определен, и 1(истина) если идентификатор в настоящее время не определён.

Директива #include

- Директива #include включает в текст программы содержимое указанного файла. Эта директива имеет две формы:
- #include "имя файла"
- #include <имя файла>
- Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы, задаваемых командой PATH.

Директива #undef

- Директива #undef используется для отмены действия директивы #define. Синтаксис этой директивы следующий
- **#undef идентификатор**
- Директива отменяет действие текущего определения #define для указанного идентификатора. Не является ошибкой использование директивы #undef для идентификатора, который не был определен директивой #define.
- Пример:
- #undef WIDTH
- #undef MAX
- Эти директивы отменяют определение именованной константы WIDTH и макроопределения MAX.
-

Директива #line

- Директива #line используется, чтобы снабдить номером операторы в программе для сообщения ошибки и перекрестной ссылки. Если программа состоит из секций, полученных из некоторого другого файла, часто полезно отметить такие секции номерами первоначальных операторов, полученных из сложной программы.

- **#line integer_constant <"filename">**
-
- Директива #line указывает, что следующая исходная команда первоначально прибыла от номера оператора filename. Как только filename был зарегистрирован, последующая команда #line, касающаяся этого файла может опустить явный filename аргумент.
-
-

Директива #error

- Синтаксис этой директивы

- **#error errmsg**
-
- Эта директива обычно вложена в инструкцию условного выражения препроцессора, которая захватывает некоторое нежелательное условие компиляции. В нормальном случае, это условие имеет значение «ложь». Если условие истинно, транслятор печатает сообщение об ошибке и компиляция останавливается.
-

Выводы

- Итак, мы сделали первые шаги. На следующей лекции мы начнем последовательное изучение языка. А пока прошу вас еще раз обратить внимание на основные моменты :
- 1) Программа на C++ рассматривается как совокупность файлов. Все файлы равноправны.
- 2) В любой программе на C++ должна быть функция main().
- 3) Все функции в C++ равноправны, отличие main() только в том, что с нее
- начинается выполнение программы.
- 4) в C++ имеется препроцессор, который обрабатывает текст программы
- перед компиляцией
- 5) для подключения функций из стандартных библиотек нужно использовать
- директиву препроцессора #include<имя> , где "имя" - имя заголовочного
- файла с объявлением этих функций.