Главная

Сравнение синтаксиса Python и C++

Введение

Язык C++ является *компилируемым*, то есть трансляция кода с языка высокого уровня на инструкции машинного кода происходит не в момент выполнения, а заранее – в процессе изготовления так называемого *исполняемого файла* (в ОС Windows такие файлы имеют расширение `.exe`, а в ОС GNU/Linux чаще всего не имеют расширения).

Сходные синтаксические конструкции Python и C++ (кратко)

Любая инструкция в C++ в отличие от Python, должна завершаться точкой с запятой (;).

Объявление переменных

Код на Python

```python
a = 10.1
b = True
c = 10
```

Код на C++

```cpp
double a = 10.1;
bool b = true;
int c = 10;
```

Поскольку на языке C++ необходимо самому указывать тип переменной при ее создании, стоит запомнить несколько базовых типов:

- int - целочисленный тип со знаком
- double - числа с плавающей точкой
- bool - логический тип (обратите внимание, что на языке C++ значения "true" и "false" пишутся со строчной буквы)

Записать данные другого типа в объявленную ранее переменную невозможно.

Условные конструкции if, else

Код на Python

```python
if condition1:  # Comment
    command 1
    command 2
elif condition2:
    command 3
else:
    command 4
```

Код на C++

```cpp
if (condition1) {  // Comment
    command 1;
    command 2;
} else if (condition2) {
    command 3;
} else {
    command 4;
}
```

Циклы

В Python блок кода, соответствующий циклу или условной конструкции обозначается двоеточием (:), а затем обозначается отступом от остального кода, причем отступ является необходимым элементом синтаксиса языка. В (C/C++) - для обособления кода, используются фигурные скобки ({), (}) - открывающая, соотвественно закрывающая. В случае если блок не содержит всего одну строчку скобки не обязательны. Отступ вложенного блока кода ненеобходим в C++, но предпочтителен для лучшей читабельности кода.

Цикл while

Код на Python

```python
while condition:
    command 1
    command 2
```

Код на C++

```cpp
while (condition) {
    command 1;
    command 1;
}
```

Цикл for

Сравнение синтаксиса цикла, пробегающего от (*min*, *max*) в Python и C++.

Код на Python

```python
for i in range(min, max):
    command 1
    command 2
```

Код на C++

```cpp
for (int i = min; i < max; i++) {
    command 1;
    command 2;
}
```

Начинания с C++11, в C++ доступен для многих обьектов, содержащих некоторое множество элементов range-based for цикл, пробегающий по всем элементам контейнера.

Код на Python

```python
v = [1, 2, 3, 4]
for x in v:
        command
```

Код на C++

```cpp
vector<int> v = {1, 2, 3, 4}
for (int x : v) {
        command;
}
```

Как и в Python, в C++ есть ключевые слова *break;* (для преждевременного выхода из цикла), *continue;* (для перехода к следующей итерации, минуя оставшееся тело цикла).

Подключение библиотек

Для подключения к исполняемой программе новых функций в C применяются так называемые "заголовочные файлы", имеющие расширение *.h* или *.hpp*. В Python для подключения функционала из другого файла или библиотеки используется команда *import*. В C++ подключение библиотеки производится при помощи директивы *#include*. В рассматриваемом примере <iostream> - заголовочный файл, содержащий базовую функциональность для работы с потоками ввода и вывода.

Код на Python

```python
import some_file.py
import numpy
```

Код на C++

```cpp
#include <some_library.h>
#include <iostream>
```

Кроме того, нередко некоторые методы, классы и функций для разрешения конфликтов имен в C++ дополнительно имеют префикс, соответсвующий пространству имен (namespace), в частности : функционал стандартной библиотеки (std::). В случае, если нет риска возникновения конфликтов имен, при написании следующей строчки

```
using namespace (some_namespace);
```

данный префикс будет автоматически добавляться.

Базовый синтаксис C++ с упражнениями

hello.cpp

Пример простой программы на C++, которая печатает "Привет, Мир!":

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Для вывода здесь используется стандартная библиотека `iostream`, поток вывода `cout`.

Исполняемые операторы в программах на C++ не могут быть сами по себе – они должны быть обязательно заключены в *функции*.

Функция `main()` – это *главная функция*, выполнение программы начинается с её вызова и заканчивается выходом из неё. Возвращаемое значение `main()` в случае успешных вычислений должно быть равно 0, что значит "ошибка номер ноль", то есть "нет ошибки". В противном процесс, вызвавший программу, может посчитать её выполнившейся с ошибкой.

Чтобы выполнить программу, нужно её сохранить в текстовом файле `hello.cpp` и скомпилировать следующей командой:

```
$ g++ -o hello hello.cpp
```

Опция `-o` сообщает компилятору, что итоговый исполняемый файл должен называться `hello`. `g++` – это компилятор языка C++, входящий в состав проекта GCC (GNU Compiler Collection). `g++` не является единственным компиляторм языка C++. Помимо него в ходе курса мы будет использовать компилятор `clang`, поскольку он обладает рядом преимуществ, из которых нас больше всего интересует одно – этот компилятор выдаёт более понятные сообщения об ошибках по сравнению с `g++`.

Упражнение №1

Скомпилируйте и выполните данную программу.

Ввод и вывод на языке C++

В Python и в C ввод и вывод синтаксически оформлены как вызов функции, а в C++ – это *операция* над объектом специального типа – *потоком*.

Потоки определяются в библиотеке `iostream`, где определены операции ввода и вывода для каждого встроенного типа.

Вывод

Все идентификаторы стандартной библиотеки определены в пространстве имен `std`, что означает необходимость обращения к ним через квалификатор `std::`.

```
std::cout << "mipt";
std::cout << 2016;
std::cout << '.';
```

```
std::cout << true;
std::cout << std::endl;
```

Заметим, что в C++ мы не прописываем типы выводимых значений, компилятор неким (пока непонятным) способом разбирается в типе выводимого значения и выводит его соответствующим образом.

Вывод в один и тот же поток можно писать в одну строчку:

```
std::cout << "mipt" << 2016 << '.' << true << std::endl;
```

Для вывода в поток ошибок определён поток `cerr`.

Ввод

Поток ввода с клавиатуры называется `cin`, а считывание из потока производится другой операцией – `>>` :

```
std::cin >> x;
```

Тип считываемого значения определяется автоматически по типу переменной `x`.

Для всех типов, кроме `char`, считывание будет производиться с пропуском символов-разделителей и до следующего символа-разделителя. При этом пробел и табуляция так же, как и символ перевода каретки, являются корректными разделителями. Считывание в char происходит посимвольно независимо от типа символа.

Например для введенной строки "Иван Иванович Иванов",

```
std::string name;
std::cin >> name;
```

считает в name только первое слово "Иван".

Считать всю строку целиком можно с помощью функции `getline()`:

```
std::string name;
std::getline(std::cin, name);
```

Считывать несколько значений можно и в одну строку:

```
std::cin >> x >> y >> z;
```

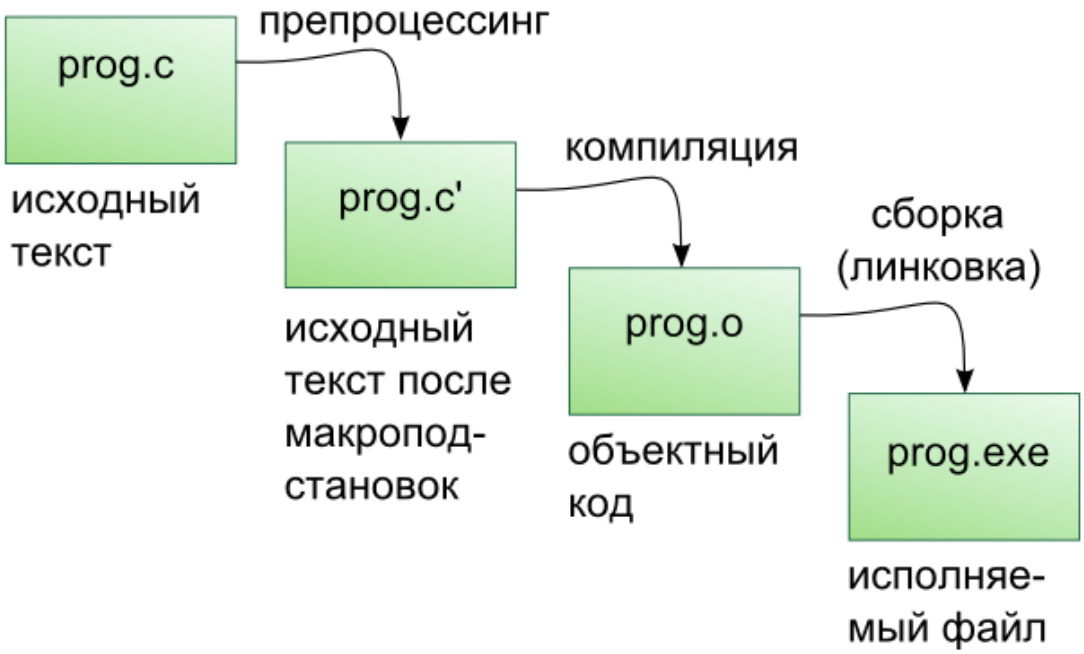
Упражнение №2

Напишите программу, которая считает гипотенузу прямоугольного треугольника по двум катетам. Ввод и вывод стандартные.

Ввод	Вывод
3 4	5

Этапы сборки: препроцессинг, компиляция, компоновка

Компиляция исходных текстов на Си в исполняемый файл происходит в три этапа.



Препроцессинг

Эту операцию осуществляет текстовый препроцессор.

Исходный текст частично обрабатывается – производятся:

1. Замена комментариев пустыми строками
2. Текстовое включение файлов – `#include`
3. Макроподстановки – `#define`
4. Обработка директив условной компиляции – `#if`, `#ifdef`, `#elif`, `#else`, `#endif`

Компиляция

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем.
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ.** Дерево разбора обрабатывается с целью установления его семантики (смысла) – например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
4. **Оптимизация.** Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Генерация кода.** Из промежуточного представления порождается объектный код.

Результатом компиляции является **объектный код.**

Объектный код – это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

При отладочной сборке возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов).

Компоновка

Компоновка также называется *связывание* или *линковка*. На этом этапе отдельные объектные файлы проекта соединяются в единый *исполняемый файл*.

На этом этапе возможны так называемые ошибки связывания: если функция была объявлена, но не определена, ошибка обнаружится только на этом этапе.

Упражнение №3

Выполните в консоли для ранее созданного файла hello.cpp последовательно операции препроцессинга, компиляции и компоновки:

1. Препроцессинг:

```
$ g++ -E -o hello1.cpp hello.cpp
```

1. Компиляция:

```
$ g++ -c -o hello.o hello1.cpp
```

1. Компоновка:

```
$ g++ -o hello hello.o
```

Принцип раздельной компиляции

Компиляция – алгоритмически сложный процесс, для больших программных проектов требующий существенного времени и вычислительных возможностей ЭВМ. Благодаря наличию в процессе сборки программы этапа компоновки (связывания) возникает возможность *раздельной компиляции*.

В модульном подходе программный код разбивается на несколько файлов `.cpp`, каждый из которых компилируется отдельно от остальных.

Это позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов. Также это даёт возможность замены отдельных компонентов конечного программного продукта, без необходимости пересборки всего проекта.

Пример модульной программы с раздельной компиляцией на C++

Рассмотрим пример: есть желание вынести часть кода в отдельный файл – пользовательскую библиотеку.

program.cpp

```
#include "mylib.hpp"
const int MAX_DIVISORS_NUMBER = 10000;

int main()
{
        int number = read_number();

        int Divisor[MAX_DIVISORS_NUMBER];
        int Divisor_top = 0;
        factorize(number, Divisor, &Divisor_top);

        print_array(Divisor, Divisor_top);
        return 0;
}
```

Подключение пользовательской библиотеки в C++ на самом деле не так просто, как кажется.

Сама библиотека должна состоять из двух файлов: `mylib.hpp` и `mylib.cpp`:

mylib.hpp

```
#ifndef MY_LIBRARY_H_INCLUDED
#define MY_LIBRARY_H_INCLUDED
```

```cpp
#include <cstdlib>

//считываем число
int read_number();

//получаем простые делители числа
// сохраняем их в массив, чей адрес нам передан
void factorize(int number, int *Divisor, int *Divisor_top);

//выводим число
void print_number(int number);

//распечатывает массив размера A_size в одной строке через TAB
void print_array(int A[], size_t A_size);

#endif // MY_LIBRARY_H_INCLUDED
```

mylib.cpp

```cpp
#include <iostream>

#include "mylib.hpp"

//считываем число
int read_number()
{
        int number;
        std::cin >> number;
        return number;
}

//получаем простые делители числа
// сохраняем их в массив, чей адрес нам передан
void factorize(int x, int *Divisor, int *Divisor_top)
{
        for (int d = 2; d <= x; d++) {
                while (x%d == 0) {
                        Divisor[(*Divisor_top)++] = d;
                        x /= d;
                }
        }
}

//выводим число
void print_number(int number)
{
        std::cout << number << std::endl;
}

//распечатывает массив размера A_size в одной строке через TAB
void print_array(int A[], size_t A_size)
{
        for(int i = A_size-1; i >= 0; i--)
        {
                std::cout << A[i] << '\t';
        }
        std::cout << std::endl;
}
```

Препроцессор C++, встречая `#include "mylib.hpp"`, полностью копирует содержимое указанного файла (как текст) вместо вызова директивы. Благодаря этому на этапе компиляции не возникает ошибок типа Unknown identifier при использовании функций из библиотеки.

Файл `mylib.cpp` компилируется отдельно.

А на этапе компоновки полученный файл `mylib.o` должен быть включен в исполняемый файл `program`.

Среда разработки обычно скрывает весь этот процесс от программиста, но для корректного анализа ошибок сборки важно представлять себе, как это делается.

Упражнение №4

Давайте сделаем это руками:

```
$ g++ -c mylib.cpp                    # 1
$ g++ -c program.cpp                  # 2
$ g++ -o program mylib.o program.o    # 3
```

Теперь, если изменения коснутся только `mylib.cpp`, то достаточно выполнить только команды 1 и 3. Если только program.cpp, то только команды 2 и 3. И только в случае, когда изменения коснутся интерфейса библиотеки, т.е. заголовочного файла `mylib.hpp`, придётся перекомпилировать оба объектных файла.

Утилита make и Makefile

Утилита `make` предназначена для автоматизации преобразования файлов из одной формы в другую. По отметкам времени каждого из имеющихся объектных файлов (при их наличии) она может определить, требуется ли их пересборка.

Правила преобразования задаются в скрипте с именем `Makefile`, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил, которые в свою очередь описываются:

1. целями (то, что данное правило делает);

2. реквизитами (то, что необходимо для выполнения правила и получения целей);

3. командами (выполняющими данные преобразования).

В общем виде синтаксис Makefile можно представить так:

```
# Отступ (indent) делают только при помощи символов табуляции,
# каждой команде должен предшествовать отступ
<цели>: <реквизиты>
    <команда #1>
    ...
    <команда #n>
```

То есть, правило make это ответы на три вопроса:

{Из чего делаем? (реквизиты)} ---> [Как делаем? (команды)] ---> {Что делаем? (цели)}

Несложно заметить что процессы трансляции и компиляции очень красиво ложатся на эту схему:

{исходные файлы} ---> [трансляция] ---> {объектные файлы}

{объектные файлы} ---> [линковка] ---> {исполнимые файлы}

Простейший Makefile

Для компиляции `hello.cpp` достаточно очень простого мэйкфайла:

```
hello: hello.cpp
    gcc -o hello hello.cpp
```

Данный Makefile состоит из одного правила, которое в свою очередь состоит из цели – `hello`, реквизита – `hello.cpp`, и команды – `gcc -o hello hello.cpp`.

Теперь, для компиляции достаточно дать команду `make` в рабочем каталоге. По умолчанию `make` станет выполнять самое первое правило, если цель выполнения не была явно указана при вызове:

$ make <цель>

Makefile для модульной программы

```
program: program.o mylib.o
        g++ -o program program.o mylib.o

program.o: program.cpp mylib.hpp
        g++ -c program.cpp

mylib.o: mylib.cpp mylib.hpp
        g++ -c hylib.cpp
```

Попробуйте собрать этот проект командой `make` или `make hello`. Теперь измените любой из файлов `.cpp` и соберите проект снова. Обратите внимание на то, что во время повторной компиляции будет транслироваться только измененный файл.

После запуска `make` попытается сразу получить цель `program`, но для ее создания необходимы файлы `program.o` и `mylib.o`, которых пока еще нет. Поэтому выполнение правила будет отложено и `make` станет искать правила, описывающие получение недостающих реквизитов. Как только все реквизиты будут получены, `make`вернется к выполнению отложенной цели. Отсюда следует, что `make` выполняет правила рекурсивно.

Фиктивные цели

На самом деле в качестве make целей могут выступать не только реальные файлы. Все, кому приходилось собирать программы из исходных кодов, должны быть знакомы с двумя стандартными в мире UNIX командами:

```
$ make
$ make install
```

Командой make производят компиляцию программы, командой `make install` – установку. Такой подход весьма удобен, поскольку все необходимое для сборки и развертывания приложения в целевой системе включено в один файл (забудем о скрипте `configure`). Обратите внимание на то, что в первом случае мы не указываем цель, а во втором целью является вовсе не создание файла `install`, а процесс установки приложения в систему. Проделывать такие фокусы нам позволяют так называемые фиктивные (phony) цели. Вот краткий список стандартных целей:

all – является стандартной целью по умолчанию. При вызове make ее можно явно не указывать; clean – очистить каталог от всех файлов полученных в результате компиляции; install – произвести инсталляцию; uninstall – и деинсталляцию соответственно.

Для того чтобы make не искал файлы с такими именами, их следует определить в `Makefile`, при помощи директивы `.PHONY`. Далее показан пример `Makefile` с целями `all`, `clean`, `install` и `uninstall`:

```
        .PHONY: all clean install uninstall

        all: program

        clean:
                rm -rf mylib *.o
        program.o: program.cpp mylib.hpp
                gcc -c -o program.o program.cpp
        mylib.o: mylib.cpp mylib.hpp
                gcc -c -o mylib.o mylib.cpp
        program: program.o mylib.o
                gcc -o mylib program.o mylib.o
        install:
                install ./program /usr/local/bin
        uninstall:
                rm -rf /usr/local/bin/program
```

Теперь мы можем собрать нашу программу, произвести ее инсталляцию/деинсталляцию, а так же очистить рабочий каталог, используя для этого стандартные make цели.

Обратите внимание на то, что в цели `all` не указаны команды; все что ей нужно – получить реквизит `program`. Зная о рекурсивной природе make, не сложно предположить, как будет работать этот скрипт. Также следует обратить особое внимание на то, что если файл `program` уже имеется (остался после предыдущей компиляции) и его реквизиты не были изменены, то команда `make` ничего не станет пересобирать. Это классические грабли make. Так, например, изменив заголовочный файл, случайно не включенный в список реквизитов (а надо включать!), можно получить долгие часы головной боли. Поэтому, чтобы гарантированно полностью пересобрать проект, нужно предварительно очистить рабочий каталог:

```
$ make clean
$ make
```

P.S. Неплохая статья с описанием мейкфайлов.