Начнём с нуля?

“Как запустить программу”

Этапы компиляции

- Препроцессинг
- Лексический анализ
- Создание объектных файлов
  - ассемблирование
  - трансляция в бинарный код
  - создание объектного кода со ссылками на функции
- Линковка

Типы файлов в C/C++

- Исполняемые файлы “*.c”, “*.cpp”
  Содержат, непосредственно, код
- Заголовочные(хедер) файлы “*.h”, “*.hpp”
  Содержат объявление функций, классов, методов, etc.

Этапы компиляции. Директивы препроцессора

- Набор команд для компилятора, облегчающие процесс сборки проекта
- Подключение библиотек
  - Подключение внутренних файлов #include <iostream>
  - Подключение внешних файлов #include “stdio.h”
- Прямая подстановка значений в код
  - #define N=50
  - #define true=false
- Использование ОС-зависимых функций:
  - #pragma once
- http://www.cplusplus.com/doc/tutorial/preprocessor/

Этапы компиляции. Лексический анализ

- Выделение из кода лексем
  - Типы данных, имена переменных и функций, константные значения
- Выделение операторов и выражений
  - a = b + c => a = (operator+)(b,c) => (operator=)(a,(operator+)(b,c))
  - a++ => (operator++)(a)
  - ++a => a(operator++)()
- Синтаксическая проверка конструкций языка

Этапы компиляции. Объектные файлы

- Исполняемые файлы на ОС-ориентированном языке
- Предназначены для исполнения ОС
- Для каждого файла проекта создается свой(а иногда и не один)

Этапы компиляции. Линковка

- Подключение библиотечных файлов
- Компоновка хедеров с объектными файлами
- Сборка объектных файлов в один исполняемый файл

Сборка. Пример

- Сборка=Компиляция+Линковка
- Одиночный файл

  g++ main.cpp -o MyProg

  Запуск:

  ./MyProg
- Несколько файлов

  g++ foo.cpp -o foo.o

  g++ bar.cpp -o bar.o

  g++ main.cpp -o main.o

  g++ foo.o bar.o main.o -o myProg

  Запуск

  ./MyProg
- И так каждый раз? А хедеры куда? А библиотеки где?...

Сборка. Альтернативы

- IDE? IDE?! VS?!! o_O
- makefile!
- Утилита make обеспечивающая сборку по правилам записанном в makefile’e
- Как вызывать?
  make
  make -f Makefile1
  Запуск:
  ./MyProg

  //Не забудьте установить утилиту make, если её нет

Сборка. Как устроен makefile

```
myProg: a.o b.o c.o                    //общая сборка
	g++ a.o b.o c.o -o myProg
a.o: a.cpp                             //собираем отдельные файлы
	g++ -c a.cpp
b.o: header.h b.cpp
	g++ -c b.cpp
c.o: header.h c.cpp
	g++ -c c.cpp
clean:                                 // чистим мусор
	rm -f myProg *.o
```

Можно автоматизировать процесс! Google в помощь!

Сборка. Статические Библиотеки

- Статические подключаемые библиотеки(SLL)
  - “*.a”, “*.lib”
  - Полное включение в код
  - gcc -c file.c
    gcc file.o -o file
    gcc file.o mylib.a -o file -static

Сборка. DLL

- Динамические подключаемые библиотеки(DLL)
  - “*.dll” , shared objects
  - Подключают нужную часть кода по необходимости
- Компилируем файл с кодом

  gcc -c mydll.c

  Собираем библиотеку

  gcc -shared -o mydll.dll mydll.o

  Подключаем

  gcc -o Myprog myprog.c -L./ -l mydll

“Виды памяти в C/C++”

Указатели

- Память - адресованная => по адресу можно получать данные
- Специальный тип, являющийся производным от других типов

```
int i = 10;          //объявление переменной с инициализацией
                     //ее значением 10
int * p = & i;       //объявление указателя на переменную i
```

- Указательная арифметика
- Функции тоже имеют адреса...

Указатели. Зачем?

- Передача данных в функции
- Создание массивов
- Связанные друг с другом данные
- Практически неограниченная память для выделения
- Приведение указателей
- Передача данных в обобщенном виде (void*)

Ссылки

- Реализация модели синонимов в C++
- Передача внешних неглобальных переменных в функции
- Безопасность
  - По ссылке мы получаем сам объект, а не адрес
  - Ссылка инициализирована

Глобальная Память

- Выделяется до начала работы программы
- Переменные определенные вне функций:
  - создание в .cpp-файле
    ```
    int maxind = 1000;
    ```
  - объявление в .h файле
    ```
    extern int maxind;
    ```

Локальная память - стек

- Выделяется/освобождается во время работы программы автоматического
- Области видимости переменных

```
{ int a=10;}
printf(“%s”,a);   // :’(
```

- Области видимости устроены по принципу стека

Статическая память. Заявленная память

- Статическая локальная переменная
- Статическая глобальная переменная

- extern - заявленная переменная

Динамическая память - куча

- Выделяется/освобождается во время работы программы вручную
- ВЫДЕЛИЛ ПАМЯТЬ - ОСВОБОДИ!
- Как пользоваться:
  - malloc/calloc/realloc/free

```
int * values =(int*) malloc(n*sizeof(int);    // Захватить участок памяти
                                              // размером в n байт

p= realloc(p, m*sizeof(int))
void free(void *p);                           // Освободить участок
                                              // памяти с адресом p
```

  - new/delete
  - new[]/delete[]

“Ввод/вывод в C/C++”

Файлы в Linux

- В **Linux** любое устройство считается файлом.

```
/dev/null        //пустое устройство, по факту удаляющее поступающие данные
/dev/cdrom       //файл, соответствующий CD-ROM
/dev/hwrandom    //файл с потоком случайных чисел
```

- Имя - ссылка на файл, следовательно, имен у файла может быть несколько.
- Права доступа принадлежат файлу, а не его имени (именам).
- При запуске С-программы по-умолчанию открываются файлы:

```
stdout      //файл вывода
stdin       //файл ввода
stderr      //файл, куда выводятся сообщения об ошибках
```

- **`stdout, stdin, stderr`** - глобальные переменные типа **FILE**.

Стандартный ввод/вывод в C

- Для ввода/вывода в стиле C используются

```
printf //функция пишет в stdout
scanf  //функция читает из stdin
puts   //функция пишет в stdout строку
gets   //функция читает из stdin строку
getch  //функция читает из stdin один символ

Для работы с ними нужно подключать:
#include <cstdio>
```

- Более подробно о содержимом **<cstdio>** : **http://cppstudio.com/cat/309/323/**

Printf

- `int printf(const char * Format, ...);`
- Функция **`printf()`** записывает в поток **`stdout`** свои аргументы в соответствии с форматной строкой **`Format`**.
- Строка **`Format`** может содержать элементы двух видов:
  - символы, подлежащие выводу на экран;
  - спецификаторы формата, определяющие способ представления аргументов на экране.
- Количество аргументов должно точно совпадать с количеством спецификаторов формата, и порядок их следования должен быть одинаковым.
  - Если аргументов будет больше, чем спецификаторов, то лишние аргументы проигнорируются.
  - Если аргументов будет меньше, чем спецификаторов, то недостающие аргументы функция извлекает из стека.
- Функция **`printf()`** возвращает количество фактически напечатанных символов. Отрицательное число означает ошибку.
- Погуглите про спецификаторы формата и модификаторы кода формата

Scanf

- `int scanf(const char* Format, ...);`
- Функция **`scanf()`** представляет собой универсальную процедуру, которая считывает данные из потока **`stdin`** и записывает их в переменные, указанные в списке аргументов. Она может считывать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат.
- Функцию **`scanf()`** может считывать поле, не присваивая его ни одной переменной. Для этого перед кодом соответствующего формата следует поставить символ **`*`**. Например:

```
scanf("%d%*c%d", &x, &y);      //При вводе строки "10/20" в переменные x и y
                               //запишутся числа 10 и 20, а знак "/" проигнорируется.
```

- Функция **`scanf()`** возвращает количество успешно введенных полей. В это число не входят считанные, но не присвоенные каким-либо переменным, поля. Если до первого присваивания произошла ошибка, то функция возвращает константу **`EOF`**.
- Погуглите про спецификаторы формата и модификаторы кода формата

Файловый ввод/вывод C. Открытие/закрытие

- Указатель файла - указатель на структуру типа **`FILE`**. В этой структуре хранится информация о файле, в частности его имя, статус и текущее положение курсора. Объявление указателя файла:

```
#include <cstdio>
FILE* fp;
```

- **`fopen()`** – Открывает поток и связывает его с файлом, возвращает указатель на этот файл. Если во время открытия файла происходит ошибка, функция **`fopen()`** вернет нулевой указатель.

```
FILE* fopen(const char* fileName, const char* mode);
```

- **`fclose()`** – Закрывает поток, открытый ранее функцией **`fopen()`**. Она записывает все оставшиеся в буфере данные в файл и закрывает его..

```
FILE* fopen(const char* fileName, const char* mode);
```

- Погуглите про режимы(mode) открытия/закрытия файлов

Файловый ввод/вывод

- **putc()**, **fputc()** - Запись символа **character** в файл, открытый с помощью функции **fopen()**. Если выполнена успешно, то возвращает символ, записанный в файл; в противном случае - константу **EOF**.

```
int putc(int character, FILE* fp);
int fputc(int character, FILE* fp);
```

- **getc()**, **fgetc()** - Считывает символ из файла.

```
int getc(FILE* fp);
int fgetc(FILE* fp);
```

- **fputs()** - записывает в заданный поток строку **str**, при возникновении ошибки возвращает константу **EOF**.

```
int fputs(const char* str, FILE* fp);
```

- **fgets()** - считывает строку из указанного потока, пока не обнаружит символ перехода или не прочитает **(length-1)** символов. Результирующая строка содержит символ перехода и заканчивается нулевым символом. В случае успеха функция возвращает указатель на введенную строку, в противном случае - нулевой указатель.

```
char* fgets(char* str, int length, FILE* fp);
```

Файловый ввод/вывод

- **fprintf, fscanf** - Аналогичны функциям **printf()** и **scanf()** за исключением того, что работают с файлами.

```
int fprintf(FILE* fp, const char* Format, ...);
int fscanf(FILE* fp, const char* Format, ...);
fprintf(stdout, ...);     // аналогично printf(...);
fscanf(stdin, ...);       // аналогично scanf(...);
fprintf(stderr, ...);     // запись в поток stderr
```

- **fflush** – Записывает содержимое буфера в файл, связанный с указателем **fp**. При вызове с нулевым аргументом, все данные из всех буферов будут записаны во все файлы, открытые для записи.
  В случае успешного выполнения возвращает 0, иначе - константу **EOF**.

```
int fflush(FILE* fp);
```

- **feof** – При обнаружении конца файла возвращает истинное значение, в противном случае - 0.
- `int feof(FILE* fp);`
- **ferror** – Возвращает истинное значение, если при выполнении операции произошла ошибка, в противном случае возвращает ложное значение.

```
int ferror(FILE* fp);
```

Ввод/вывод. C++

- `#include <iostream>`
- В начале выполнения программы на языке **C++** автоматически открываются следующие файлы:

  std::cin - Стандартный ввод - Устройство по-умолчанию: Клавиатура
  std::cout - Стандартный вывод - Устройство по-умолчанию: Экран
  std::cerr - Стандартный вывод ошибок - Устройство по-умолчанию: Экран
- Запись в файл производится с помощью оператора **<<**, а чтение - с помощью оператора **>>**.
- Стандартные потоки сами определяют типы аргументов, для вывода не нужна форматная строка.
- `using namespace std` избавляет от написания **std::** перед названиями потоков ввода/вывода. Директива **using** может быть указана в начале файла программы, в начале функции и пр.

Файловый ввод/вывод. C++

- ```
  #include <fstream>
  std::ifstream f( "input.txt" );     //Поток ввода
  std::ofstream h( "output.txt" );    //Поток вывода
  std::fstream io( "inout.txt" );     //Поток ввода-вывода
  f >> ...;
  h << ...;
  f.flush();        //Принудительная запись информации из буфера в файл
  f.close();        //Но при выходе из функции файл будет закрыт автоматически
  f.open(const char* filename, ios_base::openmode mode = ios_base::in | ios_base::out);
  ```
  ```

Linux направление потоков ввода/вывода

- Перенаправление ввода
  test_in.txt > ./MyProg
  ./MyProg < test_in.txt
- Перенаправление вывода
  ./MyProg > test_out.txt
- Перенаправление вывода в режиме добавления
  ./MyProg >> test_out.txt
- Перенаправление ввода/вывода
  ./MyProg < test_in.txt > test_out.txt

Дескриптор	Название	Описание
0	stdin	Стандартный ввод
1	stdout	Стандартный вывод
2	stderr	Стандартный вывод ошибок

- Можно пользоваться дескрипторами вместо полных названий:
  ./MyProg 2>&1 //Перенаправляется stderr на stdout.
  //Сообщения об ошибках передаются туда же, куда и стандартный вывод.
- Последовательности команд в Linux можно объединять в сценарии - bash-скрипты. Погуглите как

Полезности

1. https://stepik.org/73
2. https://stepik.org/762
3. https://stepik.org/7