Работа №1

«Основы разработки программного обеспечения»

Цель работы

Освоение принципов разработки программного обеспечения.

После выполнения работы студенты могут

- Создавать с нуля проект, который может быть собран в исполняемое приложение.
- Использовать в проекте внешние библиотеки.
- Создавать сборку приложения, пригодную для переноса на другой ПК.
- Использовать отладчик для устранения ошибок в программном обеспечении.

Теоретическая часть

Разработка современных радиоэлектронных устройств требует разработки различного программного обеспечения. Такая необходимость возникает практически на всех этапах жизненного цикла устройства от его начального моделирования до внедрения. Современные алгоритмы обработки являются ключевой частью устройства, зачастую составляют ноу хау разработки и реализуются в виде программы, исполняемой цифровой частью устройства.

Конечной формой записи программного обеспечения является **исполняемый файл**, содержащего последовательность команд, выполняемых устройством. В зависимости от архитектуры устройства, такой файл может быть как обычным текстовым документом, так и двоичным файлом, содержащим машинные команды.

Исполняемый файл, выполняемый непосредственно процессором, выглядят как набор чисел, кодирующих выполняемые команды. При этом числа, кодирующие одни и те же команды, могут быть разными на разных процессорах. А в некоторых случаях архитектурные ограничения могу быть настолько сильными,

что одна и та же работа выполняется разными процессорами с применением совершенно разных команд.

Существует способ записи команд процессора в виде понятных человеку мнемонических обозначений — язык ассемблера. Тем не менее, элементарные команды процессора зачастую очень просты, поэтому разработка типового прикладного программного обеспечения требует использования сотен тысяч таких элементарных команд в одном исполняемом файле. Вследствие этого, ручной набор таких команд применяется крайне ограниченно и только для узкого круга специализированных задач.

Для того, чтобы избежать такой рутины при решении прикладных задач, были разработаны языки программирования более высокого уровня, а также специальное программное обеспечение (**компилятор**), позволяющее преобразовать исходный код на языке высокого уровня в команды процессора. Зачастую для каждого языка высокого уровня существует версия компилятора под ряд различных программных и аппаратных платформ, что снижает зависимость от какой то конкретной платформы или версии процессора.

На сегодняшний день существует большое многообразие различных языков программирования. В данной работе будет производиться знакомство с наиболее распространенным языком высокого уровня — языком C (Си). Данный язык широко используется при написании широкого круга программного обеспечения от системного, такого как компоненты операционной системы и драйверы, до прикладного. Язык C также используется при разработке встроенного программного обеспечения на современных микроконтроллерах, а синтаксис данного языка лег в основу большого количества других языков программирования, таких как Java, PHP и JavaScript. Данный язык выгодно отличается простотой реализации компилятора, поэтому доступен практически на любом существующем устройстве.

Синтаксис языка C

В языке C выделяют отдельно синтаксические конструкции и директивы препроцессора. Первые используются непосредственно для формирования исполняемого кода, а вторые — для управления процессом сборки исходных кодов

в исполняемый файл. Рассмотрим основные синтаксические конструкции языка. Рассмотрим основные понятия, вводимые любым языком программирования, в том числе и С:

Идентификатор представляет собой последовательность из символов латинского алфавита и цифр, обозначающую какую-либо программную сущность. Идентификатор является способом дать какой-либо программной сущности понятное человеку название. Существует также ограниченное количество зарезервированных идентификаторов (ключевых слов), используемых в языке в качестве служебных. Наиболее типовое применение идентификаторов — это переменные.

Программа, написанная на языке Си, разбирается компилятором последовательно, по строкам, поэтому в языке Си не допускается обращение к какому-то идентификатору, если он не был объявлен ранее.

Переменная представляет собой область памяти, хранящую данные строго определенного типа, которой в соответствие поставлен определенный идентификатор. Для объявления переменной необходимо указать ее тип и имя, при помощи которого можно будет обратиться к переменной. К примеру выражение

```
int a;
```

Объявит переменную целого типа под названием «a». В любой строке, расположенной после данной, можно будет сослаться на эту переменную, используя ее идентификатор.

Существуют ограничения на название переменной. Название должно состоять только из букв латинского алфавита, цифр или подчеркивания, не должно начинаться с цифры, а также не допускается называть две разные переменные одним именем, так как имена переменных — единственный для компилятора способ их различить.

Каких либо ограничений на количество объявляемых переменных нет.

Язык Си является языком с жесткой типизацией. То есть, каждая переменная имеет строго определенный тип, изменение которого после объявления переменной невозможно. Такой подход является прямым отражением машинных команд, в которые впоследствии будет преобразован исходный код. Тем не менее,

несмотря на жесткую типизацию, язык позволяет присваивать значения между переменными разных типов.

При записи типа переменной используются следующие ключевые слова:

Тип данных	**Описание**
`char`	Целочисленный тип данных размером в 1 байт. Используется для посимвольного хранения строк.
`int`	Целочисленный тип данных. Размер данных может меняться в зависимости от платформы и компилятора, но, как правило, составляет 4 байта.
`float`	Вещественный тип данных одинарной точности длиной 4 байта.
`double`	Вещественный тип данных двойной точности длиной 8 байт.
`void`	Пустой тип данных. Не может использоваться напрямую. При объявлении функции обозначает отсутствие возвращаемого значения.

Помимо типа переменной, можно дополнительно указать перед типом переменной один из следующих модификаторов:

Модификатор	**Описание**
`short`	Используется вместе с `int` для указания, что переменная должна занимать меньше байт. Допускается вместо `short int` писать просто `short`. Соответствует целому числу, занимающему 2 байта.
`long`	Используется вместе с `int` и `double` для указания, что переменная должна занимать больше байт. Тип `long double` соответствует вещественному числу расширенной точности. Вместо `long int` допускается писать просто `long`. Размер `long int` зависит от платформы и может быть как 4, так и 8 байт.

Модификатор	Описание
`long long`	Используется вместе с `int` для объявления целочисленной переменной в 8 байт длиной.
`unsigned`	Используется вместе с `int` или `char` для указания, что переменная должна быть беззнаковой. К примеру, `char` имеет диапазон значений от -128 до 127, а `unsigned char` — от 0 до 255. Можно комбинировать `unsigned` с другими модификаторами: `unsigned long int`. В случае с `int` допускается писать просто `unsigned`.
`signed`	Используется для указания, что переменная должна быть знаковой. По умолчанию любая целая переменная является знаковой, поэтому этот модификатор не используется на практике.

Существуют более сложные типы данных. Одним из них является **перечислимый тип**. Такой тип данных позволяет задать человекопонятные имена набору идущих подряд констант, что в дальнейшем позволяет избежать появления в коде «магических чисел», смысл которых теряется. Для определения перечислимого типа используется ключевое слово `enum`:

```
enum ReturnType
{
    RT_OK,
    RT_ERROR
};
```

Константы, объявленные в `enum`, являются целыми числами идущими по порядку, начиная с нуля. В вышеприведенном примере `RT_OK == 0`, `RT_ERROR == 1`. При изменении `enum` компилятор автоматически переназначает новые номера константам.

Массивы, указатели, структуры

Более сложные типы данных формируются на основе наборов из простых переменных.

При необходимости создать набор однотипных значений следует использовать **массив**. Массив представляет собой набор однотипных переменных, объединенных общим идентификатором и различающихся внутри идентификатора по целочисленным индексам. Массив объявляется так же, как и обычная переменная, но после имени переменной в квадратных скобках записывается его размер.

Типовое применение массива — представление строки в виде последовательности кодов символов:

```
char c[128];
```

Для обращения к элементам массива также используются квадратные скобки. Нумерация в массиве всегда начинается с нуля:

```
c[0]   = 1;
c[127] = 19;
```

При операциях с массивами часто оперируют с указателями. Указатель представляет собой целое число, хранящее адрес в памяти. Тип указателя определяет тип данных, лежащих по хранимому адресу. Указатель при объявлении отличается от обычной переменной наличием символа «*» между типом и названием переменной. Например, выражение

```
int *a;
```

объявляет указатель на переменную целого типа. Звездочку допускается писать как вплотную к типу, так и вплотную к имени переменной:

```
int* b;
```

Однако при объявлении нескольких переменных одного типа через запятую звездочка также может применяться, и тогда компилятор смотрит на наличие звездочки перед именем переменной. На практике лучше не смешивать в одной строке объявления переменных и указателей, чтобы избежать путаницы В приведенном примере переменные `a` и `d` будут обычными, `b` и `c` – указателями:

```
int a, *b, *c, d;
```

Прямое присвоение указателю произведет изменение хранимого в нем адреса. К примеру, можно присвоить указателю адрес другой переменной, полученный при помощи операции взятия адреса («&»):

```
int a = 0;
int* b = &a;
```

Для доступа к данным по указателю следует произвести разыменование, поставив перед названием переменной символ «*». В приведенном ниже примере производится запись числа 10 по адресу, хранимому в переменной `b`. Так как `b` хранит адрес `a`, то значение `a` будет изменено.

```
*b = 10;
```

При работе со строками используют указатель на массив char'ов — тип `char*`. Доступ к отдельным элементам такого массива такой же, как и у обычных массивов:

```
char* str = "hello";
s[0] = 'H';
```

В случае, если необходимо объединить несколько разнородных переменных в одну большую, можно использовать **структуру**. Для объявления структуры используется ключевое слово `struct`:

```
struct
{
    int a;
    double b;
} SomeVars;
```

Отдельные переменные структуры называются **поля**, а доступ к ним осуществляется через оператор «`.`»:

```
SomeVars.a = 10;
double c = SomeVars.b + 1;
```

В случае, если переменная является указателем на структуру, для доступа к полям необходимо сначала разыменовать указатель:

```
SomeVars* s;
(*s).a = 10;
```

Более простой способ записи — использование оператора «->»:

```
SomeVars* s;
s->a = 10;
```

Наконец, любому составному типу данных можно присвоить новый идентификатор и таким образом зарегистрировать новый тип. Для этого используется ключевое слово **`typedef`**. Приведем пример нового типа данных, хранящего одновременно указатель и количество элементов в массиве, на который он указывает:

```
typedef struct
{
    float* ptr;
    int size;
} blob;
```

После такого объявления можно использовать следующую форму записи:

```
float ar[10];
blob b;
b.ptr = &a;
b.size = 10;
```

Через `typedef` вводятся новые типы данных, в том числе предоставляемые стандартной библиотекой. К примеру, так определяется тип `size_t`, хранящий размер произвольных данных:

```
typedef unsigned int size_t;
```

Литералы

Литерал является способом записи константы в теле программного кода. Такая константа представляет собой данные, вводимые на этапе компиляции программы, которые физически будут храниться в исполняемом файле.

Литералы подразделяются на строковые и числовые.

Строковый литерал представляет собой последовательность символов, заключенную в двойные кавычки. Такая последовательность соответствует типу `char*` (указатель на строку):

```
char* a = "Hello";
```

Отдельно выделяют **символьный литерал**. Он соответствует одному символу (тип `char`). Записывается как буква, цифра или обозначение спецсимвола в одинарных кавычках:

```
char c = 'a';
```

При необходимости записать непечатный символ в строковый литерал используется специальная форма записи с обратной косой чертой («\»). Если при обработке литерала компилятор находит данный символ, то фактически вставляемый в исполняемый файл байт будет зависеть от того, какой символ записан после косой черты. Такая форма записи позволяет например вставить в строку символы, являющиеся служебными в языке программирования:

Форма записи	**Вставляемый символ**	**Пояснение**
`\r`	<Возврат каретки>	Непечатный символ, используемый для разбиения текстовых документов на строки.
`\n`	<Перевод строки>	Непечатный символ, используемый для разбиения текстовых документов на строки.
`\t`	<Табуляция>	Символ «широкого пробела»
`\\`	`\`	
`\"`	`"`	
`\'`	`'`	

Числовой литерал представляет собой последовательность цифр, может содержать символ «`.`» для разделения целой и дробной части, а также символы «`e+`» или «`e-`» для задания экспоненциальной части. Например, запись вида `2.0e10` трактуется как $2 \cdot 10^{10}$, а `-3e-5` — как $-3 \cdot 10^{-5}$. При наличии символа «`.`»

литерал соответствует типу `double`, в остальных случаях — типу `int`.

Целые числа можно записывать в различных системах счисления:

- Простая форма записи подразумевает десятичную систему. Например, `221`.
- В случае, если первая цифра равна 0, считается, что число записано в восьмеричной системе счисления. Например, `0212` соответствует числу 138.
- Если литерал начинается на `0x`, это соответствует шестнадцатеричному числу. Например, `0x411` соответствует $411_{16}=1041_{10}$.
- Для записи двоичных чисел не существует универсального стандарта языка. Тем не менее, некоторые компиляторы, в частности gcc и его версия для windows, mingw, позволяют записывать побитово двоичные числа с префиксом 0b. Например, `0b1100` соответствует $1100_2=12_{10}$.

Операции

Операция представляет собой некоторую функцию, выполняемую над операндами (к примеру, переменными). Если операндом является переменная, то будет взято ее значение на момент выполнения операции. Если операндом является результат вызова функции, то сначала будет вызвана данная функция, будет вычислено ее возвращаемое значение, и лишь затем оно будет использовано в операции.

Операции разделяются на унарные, использующие один входной операнд, и бинарные, использующие два операнда. К унарным операциям относятся:

- «`-`» — Унарный минус
- «`+`» — Унарный плюс
- «`~`» — Побитовая инверсия
- «`!`» — Логическое отрицание
- «`&`» — Взятие адреса
- «`*`» — Разыменование указателя

- «`sizeof`» — Вычисление занимаемой памяти
- Инкремент «`++`» и декремент «`--`». Особенность этих операций в том, что порядок их выполнения завсист от того, с какой стороны от переменной они записаны. При написании «`a++`» значение переменной будет увеличено на единицу после выполнения всех остальных операций, а при написании «`++a`» — в самом начале.

К бинарным операциям относятся следующие:

- Операция присовения — «`=`». Выражение «`a=100`» присваивает переменной значение 100 и возвращает присвоенное значение. Присвоение всегда производится **справа налево**. Это делает возможным проверку присвоенного значения, а также цепные присвоения:

  ```
  b=c=a=100;
  ```

  Также при помощи присвоения можно поменять значение переменной:

  ```
  v = v + 1
  ```

  При выполнении данного кода сначала будет вычислено значение `v`, затем оно будет увеличено на 1, а затем полученный результат будет записан обратно в `v`.
- Простые арифметические операции: «`+`» — сложение, «`-`» — вычитание, «`*`» — умножение.
- Операция деления: «`/`». В случае, если первый операнд деления является целым числом, деление будет производиться в целых числах, с отбрасыванием дробной части. В противном случае будет производиться вещественное деление.
- «`%`» — остаток от деления. К примеру, результатом операции `10%3` является `1`.
- Побитовые операции: «`&`» — поразрядное И, «`|`» - поразрядное ИЛИ, «`^`» — поразрядное исключающее ИЛИ.
- Сдвиговые операции: «`<<`» — сдвиг влево, «`>>`» — сдвиг вправо. К примеру, в результате операции `9 << 4` получится число 144 (0x90), а в результате `0xFF00 >> 8` получится 255 (0xFF).
- Логические операции: «`&&`» — логическое И, «`||`» — логическое ИЛИ.

- Операции сравнения: «>» — больше, «>=» — больше или равно, «<» — меньше, «<=» — меньше или равно, «==» — равно, «!=» — не равно.

Практически для всех бинарных операций допускается сокращенная форма записи. К примеру, два следующих выражения, увеличивающих значение переменной на 2.5, полностью идентичны:

```
a = a + 2.5;
a += 2.5;
```

Операции выполняются в соответствии со встроенным приоритетом. К примеру, при написании:

```
a = 10 + 20*3
```

сначала будет выполнено умножение, затем сложение и затем присвоение. При использовании круглых скобок можно изменить порядок следования операций. В приведенном выше примере, можно выполнить сложение до умножения, если записать операцию следующим образом:

```
a = (10 + 20) * 3
```

Структурирование кода

Инструкция — это некое элементарное действие. Инструкция может содержать обращение к переменным или вызов функции. В языке С инструкции разделяются оператором «;». Хорошим тоном при написании кода считается такое оформление, при котором одна строка содержит только одну инструкцию:

```
int a=1;
a += 4;
```

Вычислительный блок (иногда также называется «операторный блок») - это участок кода, ограниченный фигурными скобками и содержащий некоторое количество инструкций. Такой блок используется для структурирования кода, выделения отдельных участков и управления выполнением программы. Также любая переменная, объявленная внутри вычислительного блока, будет доступна только внутри него.

Для управления программой используются **операторы**. В языке С существуют следующие операторы:

- Оператор безусловного перехода `goto`. Осуществляет переход к заданной метке, объявленной где-либо в программе. Например, следующий участок кода будет до бесконечности увеличивать значение переменной на 1:

```
int a=0;
loop: a+=1;
goto loop;
```

  На практике использование оператора `goto` является дурным тоном, так как сильно снижает читаемость кода и провоцирует ошибки.

- Оператор условного перехода `if`. В общем виде условный переход записывается следующим образом:

```
if( УСЛОВИЕ )
{
}
else
{
}
```

  Оператор `if` вычисляет выражение, подставленное в качестве условия. Если результатом данного условия является ненулевое значение, то выполняется операторный блок, идущий сразу после `if`. Второй операторный блок, обозначенный ключевым словом `else`, не является обязательным и может отсутствовать. Он будет выполнен в том случае, если условие не выполнено, то есть не выполнился основной операторный блок. После обработки условного перехода выполнение программы продолжается в обычном режиме, то есть выполняются инструкции, записанные далее.

  Если вычислительный блок, стоящий после if или else, содержит только одну инструкцию, допускается не писать фигурные скобки:

```
if( a==128)
    a = 1;
```

- Операторы цикла. **Цикл** — это повторяемый несколько раз один и тот же участок кода. Одно выполнение цикла носит название **итерация**. Инструкции,

выполняемые в цикле, также заключаются в фигурные скобки, однако если цикл содержит всего одну инструкцию, фигурные скобки можно не писать. Язык С поддерживает несколько способов задания циклов:

- Цикл с предусловием `while` проверяет заданное условие перед началом следующей итерации и в случае, если условие выполняется, запускает итерацию, иначе прерывает цикл:

```
while (УСЛОВИЕ )
{
}
```

  Вычисление условия в цикле `while` аналогично оператору `if`. После того, как цикл завершился, выполнение продолжается в обычном режиме с первой строчки, записанной после закрытой фигурной скобки цикла.

- Цикл с постусловием работает аналогично `while`, но проверяет условие после выполнения итерации:

```
do{
}while( УСЛОВИЕ );
```

- Цикл общего назначения. Такой цикл записывается следующим образом:

```
for ( ИНИЦИАЛИЗАЦИЯ ; УСЛОВИЕ ; ОПЕРАТОР )
{
}
```

  В поле ИНИЦИАЛИЗАЦИЯ записывается инструкция, выполняемая при старте цикла, в поле УСЛОВИЕ — условие начала новой итерации, а в поле ОПЕРАТОР — инструкция, выполняемая при переходе к новой итерации. Например, в приведенном участке кода:

```
char ar[128];
for (a=0; a<128; a++)
{
    ar[a] = 0;
}
```

  цикл используется для того, чтобы обнулить все 128 элементов массива `ar`.

Такой же цикл можно записать через `while`:

```
char ar[128];
a=0;
while(a<128)
{
    ar[a] = 0;
    a++;
}
```

- Для задания бесконечного цикла достаточно задать цикл с условием, которое всегда будет истинным. Например, цикл `while(1)`. Также допускается запись вида `for(;;)`.
- Также язык С содержит операторы управления циклом. Для того, чтобы мгновенно выйти из цикла, следует использовать оператор `break`. Для мгновенного перехода к началу следующей итерации цикла существует оператор `continue`.

Создание функций

Дальнейшее структурирование программного кода производится путем его разделения на отдельные исполняемые участки — функции. **Функция** представляет собой оформленный участок кода, обозначаемый идентификатором, которому при запуске передается набор параметров, а результатом его работы является некоторое возвращаемое значение. Язык С выделяет два понятия — **объявление** функции и **определение** функции.

Объявление функции позволяет указать наличие функции в коде. В литературе также встречается понятие «**Прототип функции**», то есть объявление способа ее вызова без конкретизации ее содержимого. Типовая практика — привести объявления функций в начале файла в качестве оглавления. Объявление выглядит следующим образом:

```
ТИП ИМЯ(АРГУМЕНТЫ);
```

`ТИП` определяет тип данных, возвращаемых функцией. В случае, если функция не должна ничего возвращать, вместо типа данных указывается `void`.

ИМЯ представляет собой идентификатор, являющийся именем функции. Функции различаются компилятором только по идентификатору, поэтому нельзя давать одинаковые идентификаторы разным функциям.

АРГУМЕНТЫ — это список локальных переменных внутри функции, передаваемых в нее в качестве начальных значений. Аргументы разделяются запятой. В общем случае функция может не принимать никаких аргументов, тогда вместо них записывается ключевое слово `void` либо скобки остаются пустыми.

Пример объявления функции, принимающей в качестве аргументов вещественное и целое число и возвращающей вещественное число:

```
double power(double a, int p);
```

Определение функции записывается так же, как и объявление, однако определение содержит вычислительный блок и набор инструкций, составляющих содержимое функции. При определении функции используется оператор `return`, который мгновенно выходит из функции. В случае, если функция возвращает какое-либо значение, оно указывается после `return`. Например, определение вышеприведенной функции выглядит следующим образом:

```
double power(double a, int p)
{
    double res = a;
    for(int a=0; a<p; a++) res *= a;
    return res;
}
```

Категорически **запрещается** определять одну функцию внутри другой. Тело функции должно быть обязательно заключено в фигурные скобки, которые не должны быть включены ни в какие другие фигурные скобки.

Для того, чтобы выполнить код, записанный в функции, необходимо ее вызвать. Для вызова функции необходимо написать ее идентификатор и в скобках указать передаваемые аргументы в том же порядке, в котором они указаны при объявлении функции. В случае, если функция не принимает никаких аргументов, необходимо просто указать пару пустых скобок. Возвращаемое значение можно присвоить какой-либо переменной, либо использовать напрямую:

```
double a=power(2, 10);
if( a > power(10, 3) )
{
    a=0;
}
```

Как правило, объявления функций выносят в отдельный файл, называемый заголовочным. Он имеет расширение «.h» и используется как оглавление для файла, содержащего непосредственно код приведенных функций. Обычно объявления функций снабжены комментариями, позволяющими описать назначение всех функций и приведенных констант. В случае, если исходный код функций уже откомпилирован в статическую или динамическую библиотеку, заголовочный файл необходим для того, чтобы компилятор знал, какие функции присутствуют в библиотеке.

В языке Си отсутствуют какие-либо специальные функции с зарезервированными именами. Тем не менее, существует функция, с которой начинается выполнение программы операционной системой — функция `main`. Она объявляется следующим образом:

```
int main(int argc, char** argv)
```

Аргументы функции — это параметры командной строки, переданные при запуске. Если отсутствует необходимость обрабатывать параметры командной строки, можно сократить объявление `main`. Простейший вариант функции `main` выглядит следующим образом:

```
int main()
{
    return 0;
}
```

Директивы препроцессора

Помимо синтаксических конструкций, язык Си содержит **директивы препроцессора**. Они используются для управления процессом сборки на первом этапе. Рассмотрим основные из них.

К директивам препроцессора можно отнести **комментарии**. Они используются для добавления в коде пояснений, облегчающих понимание кода, либо для временного отключения ненужных участков. Классический язык Си поддерживает только многострочные комментарии, которые начинаются с символов «/*» и заканчиваются символами «*/». Все что находится между ними будет проигнорировано на этапе сборки. Более новые стандарты языка вводят поддержку однострочных комментариев, заимствованных из языка C++. При написании символов «//» все, что находится после этих символов до конца текущей строки, будет проигнорировано:

```
/* Многострочный
комментарий */
int a=0; // Однострочный комментарий
```

Более сложные директивы препроцессора начинаются с символа «#», расположенного в начале строки. Рассмотрим две основные используемые директивы.

Директива `#include` позволяет включить в проект другой файл. Фактически, содержимое такого файла будет подставлено вместо данной директивы. После директивы имя файла указывается в угловых скобках, либо в двойных кавычках. В первом случае файл будет искаться компилятором по системным адресам, а во втором — по системным адресам и в папке с проектом:

```
#include <stdio.h> // Включение библиотеки стандартного ввода-вывода
#include "customlib.h" // Включение сторонней библиотеки
```

Директива `#define` используется для макроподстановки. Она позволяет определить правило замены символов перед компиляцией. Таким способом можно объявлять константы. Например:

```
#define C_SPEED 3.0e8
double lambda = 250.0e6 / C_SPEED;
```

После прохода препроцессора во всех местах исходного кода вместо `C_SPEED` будет подставлено `3.0e8`:

```
double lambda = 250e6 / 3.0e8;
```

Макроподстановка определяется от заданного места до конца файла, однако можно ограничить область макроподстановки директивой `#undef`:

```
#undef C_SPEED
```

Сборка исполняемого файла

Рассмотрим процесс сборки проекта из исходных кодов. Для примера, возьмем проект, состоящий из нескольких исходных файлов:

- functions.c — файл с исходным кодом вспомогательных функций.
- functions.h — заголовочный файл, содержащий объявления функций из functions.c.
- main.c — файл, содержащий функцию main, с которой начнется выполнение программы. В начале файла main.c стоит несколько директив #include, в том числе #include "functions.h"

Также в проекте необходимо использовать динамическую библиотеку lib.dll. Она предоставляется вместе с заголовочным файлом lib.h.

На первом этапе сборки исходные коды обрабатываются **препроцессором**. Он проходит по всем файлам с расширением «.c» и обрабатывает директивы `#include` и `#define`, а также пропускает комментарии. Полученные файлы являются промежуточными и как правило даже не сохраняются на диск, оставаясь в оперативной памяти.

На втором этапе в работу включается **компилятор**. Он обрабатывает все синтаксические конструкции языка, преобразовывая исходный код в набор команд. При этом каждый файл «.c», обработанный препроцессором, преобразуется в так называемый объектный файл с расширением «.o».

Завершающим этапом сборки является **компоновка**. На данном этапе компоновщик проверяет зависимости, записанные в объектных файлах. Вызов любой функции является зависимостью, поскольку в качестве адреса вызова функции нужно подставить адрес места в объектном файле, где эта функция описана. Если компоновщик успешно находит все зависимости, он объединяет полученные объектные файлы в один исполняемый файл и подставляет

правильные адреса функций. В конечном счете это приводит к формированию готового исполняемого файла.

Следует отметить, что компилятор различает статические и динамические библиотеки. Статические библиотеки компонуются вместе с исполняемым файлом и после сборки являются его частью. Динамические библиотеки не компонуются внутрь. Вместо этого в исполняемом файле устанавливается зависимость от внешней библиотеки, которую необходимо будет найти в процессе запуска приложения, и окончательная компоновка производится при запуске. В системе MS Windows динамические библиотеки могут находиться в одной папке с исполняемым файлом. На рисунке 1 приведена схема процесса сборки программного обеспечения из исходных кодов:

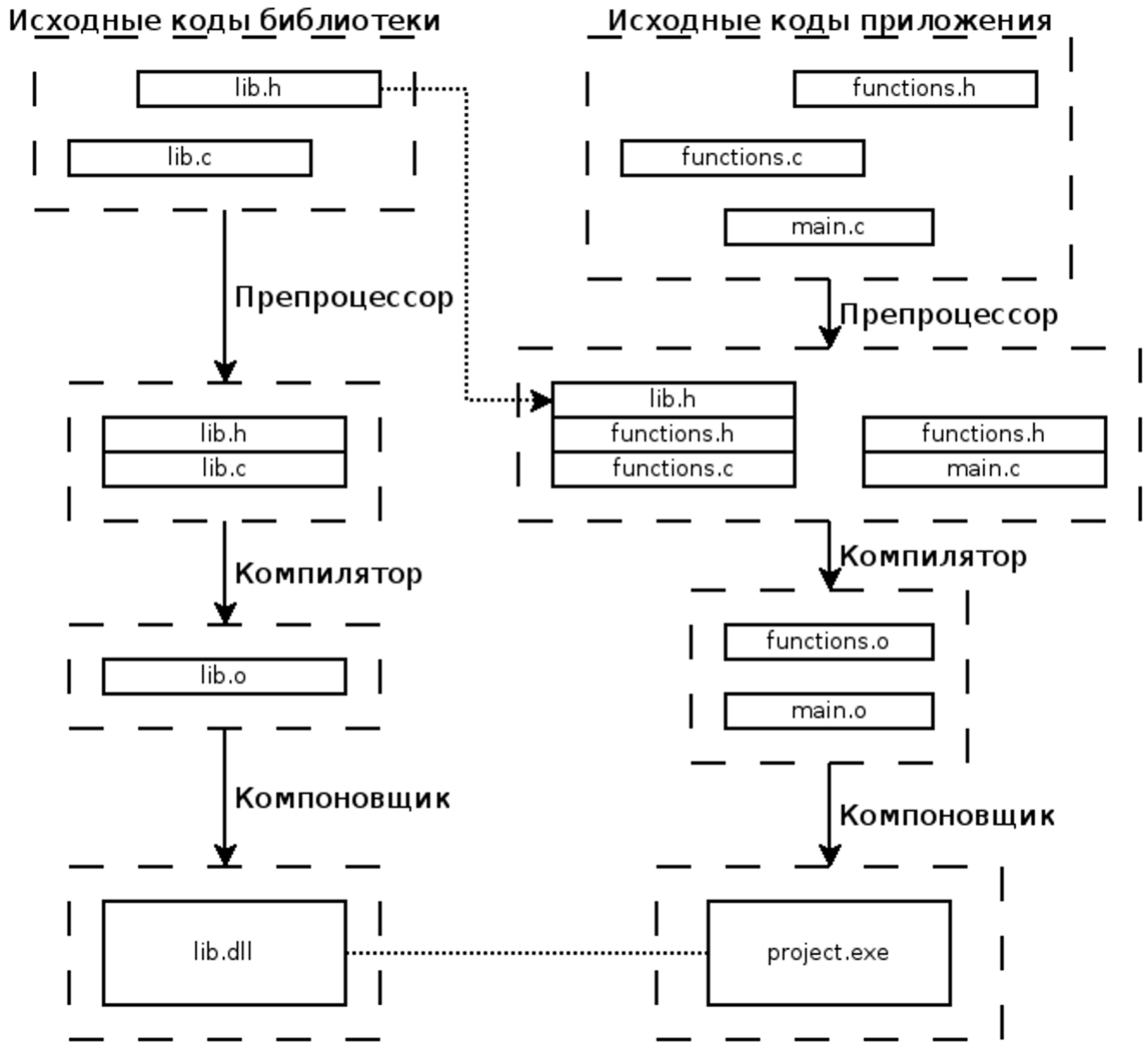

Рисунок 1 - Диаграмма сборки программного обеспечения.

Среда разработки

Стандарт языка C регламентирует только содержимое файлов исходных кодов. Вследствие этого, каждый компилятор организовывает процесс сборки по-своему. Как правило, для того, чтобы описать процесс сборки, создается специальный набор файлов — так называемый **проект** — в котором описывается, какие из файлов исходных кодов необходимо собрать, в каком порядке, и как в итоге должен называться исполняемый файл.

При разработке программного обеспечения в рамках курса рекомендуется использовать среду разработки Qt Creator. Она является средой с открытым исходным кодом и доступна для свободного скачивания на сайте http://qt.io/. Допускается использование сторонних сред разработки, позволяющих создавать исполняемые файлы, таких как Microsoft Visual Studio, Eclipse, CLion, CodeBlocks, Dev-C++ и т. д.

Окно среды разработки в процессе работы содержит в левой части дерево, отображающее файлы проекта (рисунок 2). Корневой элемент дерева — это название проекта. Вызов контекстного меню на названии проекта путем нажатия правой кнопки мыши позволяет собрать проект, добавить к нему файлы (как новые, так и существующие), либо закрыть проект.

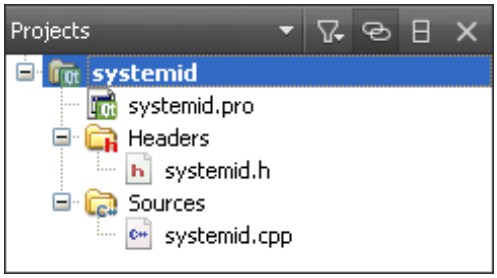

Рисунок 2 - Дерево проекта в Qt Creator

Файлы проекта группируются в папки «Headers» — для заголовочных файлов и «Sources» — для файлов исходных кодов. Отдельно стоит файл с расширением «pro», содержащий описание проекта. В файле в виде простого текста описывается, каким образом следует собрать проект, и что он в себя включает. Для описания проекта используются встроенные переменные, которым можно задавать значения при помощи знака равенства. Также, если переменная является списком, можно добавлять к списку значения при помощи «+=» и убирать при помощи «-=». Основные переменные проектного файла следующие:

`TEMPLATE` — описывает формат проекта. Возможные значения: «`app`» — прикладное приложение, «`lib`» — библиотека, «`subdirs`» — проект, объединяющий несколько проектов, расположенных во вложенных папках.

`CONFIG` — список, используемый для настройки проекта. В большинстве случаев задается однократно при создании проекта. Содержит в себе список

используемых модулей Qt.

SOURCES — список файлов исходных кодов, включенных в проект. Пути к файлам задаются относительно папки с файлом .pro и разделяются пробелом.

HEADERS — список заголовочных файлов, включенных в проект. Пути к файлам задаются относительно папки с файлом .pro и разделяются пробелом.

LIBS — набор инструкций для компоновщика, позволяющий указать подключаемые внешние библиотеки. Инструкции следует указывать через знак «+=». Для того, чтобы подключить внешнюю библиотеку, используются следующие инструкции:

- -L"<PATH>" — указание компоновщику дополнительного пути для поиска внешних библиотек. Вместо <PATH> следует подставить путь к папке с библиотеками.
- -l<NAME> — указание компоновщику названия подключаемой внешней библиотеки. В среде MS Windows компоновщик будет искать в известных ему папках файл <NAME>, <NAME>.dll или lib<NAME>.dll.

После внесения изменений в проектный файл следует сохранить его и дождаться, пока среда разработки перечитает проектный файл и примет сделанные изменения. Пример проектного файла для небольшого проекта выглядит следующим образом:

```
TEMPLATE = app
CONFIG += console
CONFIG -= qt
SOURCES += main.c
HEADERS += mylib.h
LIBS += -L"C:\_progs\mylibs"
LIBS += -lmylib
```

Помимо файла .pro, среда автоматически создает файл с расширением .pro.user, хранящий настройки конкретной сборки. Данные настройки привязаны к компилятору и не являются частью исходных кодов проекта, поэтому при переносе проекта на другой компьютер файл .pro.user переносить не следует.

Порядок проведения работы

В данной работе необходимо разработать приложение на языке Си, реализующее заданный алгортм работы. Алгоритм задается в виде блок-схемы, расположенной в персональном репозитории.

1. Создать проект-заготовку

   1.1. Создать пустой проект на языке C.

   1.2. Создать новый файл main.c и добавить его к проекту

   1.3. В файле main.c добавить включение заголовочного файла stdio.h

   1.4. Определить в main.c функцию `int main()`, возвращающую 0 в конце своего операторного блока.

   1.5. Используя функцию `puts`, вывести в консоль произвольную строку. В качестве аргумента функции следует передать строковый литерал, то есть строку, заключенную в двойные кавычки

   ```
   puts("Hello world");
   ```

   1.6. Собрать приложение, запустить его и убедиться в его работоспособности.

   1.7. Данный пункт является необязательным. Для вывода сообщений в консоли на русском языке необходимо задать кодировку файлов исходных кодов и настроить текстовый терминал. Для этого необходимо:

      1.7.1. Подключить заголовочный файл `windows.h`

      1.7.2. Выяснить текущую кодировку файла исходных кодов main.c. В среде Qt Creator следует вызвать пункт меню «Правка» → «Выбрать кодировку». Следует запомнить тот пункт списка, который будет выделен.

      1.7.3. В начале функции main следует записать следующее:

      ```
      SetConsoleCP(CODEPAGE);
      SetConsoleOutputCP(CODEPAGE);
      ```

      Вместо `CODEPAGE` следует подставить номер кодовой страницы. Для кодировки UTF-8 он равен 65001, для кодировки Windows-1251 он равен 1251.

1.7.4. В настройках терминала Windows, вызываемого при запуске программы, следует выбрать шрифт Lucida Console. Настройки терминала доступны через меню, вызываемое при нажатии правой кнопкой мыши на заголовок окна.

2. Создать функцию, обрабатывающую ввод с клавиатуры. Функция должна задавать пользователю вопрос, на который можно ответить «Да» или «Нет» и возвращать ответ в виде числа, где «1» означает «Да», а «0» означает «Нет». В качестве аргумента в функцию передается строка с вопросом, выводимым на экран.

2.1. Определить функцию:

```
int getYesNo(char* question)
{
}
```

Функция должна быть определена раньше функции `main`.

2.2. В теле функции вывести переданную переменную `question`, используя следующую конструкцию:

```
printf("%s [Y/N] ", question);
```

2.3. Создать массив для хранения строки, введенной пользователем, и запросить пользовательский ввод:

```
char answer[128];
scanf("%s", answer);
```

2.4. В случае, если введенная строка начинается с символа 'y', следует вернуть из функции значение 1:

```
if(answer[0] == 'y')
    return 1;
```

2.5. В самом конце функции следует вернуть нулевое значение. Функция будет выполнена до этой строки в том случае, если введенная строка не начинается с символа `'y'`:

```
return 0;
```

2.6. Протестировать созданную функцию. Для этого следует вызвать ее в функции main и сохранить результат ее выполнения в целочисленной переменной. Переменную следует вывести на экран для проверки:

```
int a = getYesNo("Yes or no?");
printf("%d\n", a);
```

2.7. Протестировать приложение и убедиться, что функция работает корректно, то есть выводимое на экран число соответствует вводимой строке.

3. Обратиться к репозиторию и получить оттуда блок-схему алгоритма. Для этого необходимо, используя клиент git, склонировать себе свой репозиторий, используя предоставленные адрес, логин и пароль. В склонированную папку с репозиторием вполследствии будут помещены исходные коды проекта.

4. Реализовать алгоритм, представленный в виде блок-схемы. Для этого рекомендуется:

4.1. Представить узлы алгоритма в виде отдельных функций без возвращаемого значения (тип `void`).

4.2. Каждая функция должна содержать в себе вызов функции `getYesNo` и вызов другой аналогичной функции, соответствующей диаграмме, в зависимости от ответа пользователя.

4.3. Для того, чтобы можно было вызвать функции в произвольном порядке, следует в начале файла записать объявления функций. Объявление функции представляет собой заголовок функции без самого кода:

```
int getYesNo(char* question);
```

Разработанный алгоритм следует протестировать, проверив самостоятельно все преходы между блоками.

5. Отправить исходные коды программы в свой персональный репозиторий на сервере.

5.1. Создать в склонированной ранее папке новую папку с именем, соответствующим названию проекта.

5.2. Скопировать в созданную папку исходные файлы проекта. К исходным

файлам относятся файлы с расширением `.pro` и `.cpp`.

Внимание! Файл с расширением `.pro.user` не относится к файлам исходных кодов, а хранит настройки сборки проекта на конкретной машине, поэтому его не следует отправлять вместе с исходниками. В проводнике Windows расширение файла скрыто, однако файл `.pro.user` отличается большим размером, чем файл проекта (порядка десятков кБ). При редактировании в блокноте файл `.pro.user` начинается со строки «`<?xml version="1.0" encoding="UTF-8"?>`»

5.3. Используя клиент git, добавить новые файлы (команда `add`), а затем зафиксировать новую ревизию (команда `commit`).

5.4. Отправить ревизию на сервер (команда `push`).

5.5. Для проверки корректности выполнения пункта 5 необходимо склонировать свой репозиторий в другую папку и убедиться, что исходные коды проекта находятся в новом склонированном репозитории.

6. **ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ**. Составить программу, проверяющую, является ли введенное число простым. Для этого необходимо:

6.1. Запросить у пользователя ввод числа:

```
int v;
scanf("%d", &v);
```

6.2. Определить цикл по простым числам от 2 до v-1:

```
int a;
for(a=2; a<v; a++)
{
}
```

6.3. Прервать цикл, если введенное число делится нацело на переменную цикла:

```
if(v%a == 0)
    break;
```

6.4. Если по окончании цикла `a==v`, вывести сообщение о том, что число является простым.В противном случае вывести значение `a`, которое является

наименьшим делителем введенного числа.

6.5. Запустить приложение и протестировать его.

Вопросы для самоконтроля и подготовки к защите работы №1

1. Что такое переменная? Какие типы переменных вы знаете?
2. Какие операторы для управления работой программы вы знаете?
3. Что такое цикл? Каким образом в языке С можно задать цикл?
4. Чем отличается объявление и определение функции?
5. Из каких файлов состоит проект на языке С?