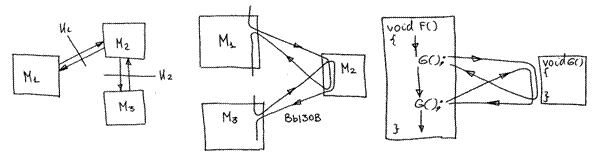
***3.4. Модульное программирование***

 «Divide et impera (Разделяй и властвуй)»

Модульное проектирование в технологии программирования очевидно по сути. Тем более, что любая промышленная технология производства рано или поздно приходит к сборке сложных изделий из набора совместимых и взаимозаменяемых деталей. Никому не надо объяснять термин «интерфейс». Тем не менее, наиболее сложно соблюдать эту заповедь: разрабатывать модульные программы. Отчасти это происходит потому, что взаимодействие модулей в программе несколько отличается от взаимодействия между модулями в другой технической системе.

Особенности функции как модуля

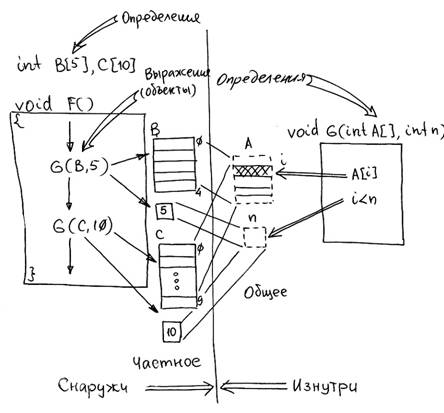
В чем разница между модулями (функциями) в программе и модулями в другой технической системе, например автомобиле. Там и здесь речь идет о завершенных изделиях, имеющих стандартные интерфейсы соединения модулей (например, шланг подачи бензина или провод подключения аккумулятора в силовом агрегате автомобиля). Но в конкретной технической системе соединение модулей производится раз и навсегда (**статически**), а в интерфейсах протекают непрерывные процессы: по бензопроводу подается горючее, а от аккумулятора – напряжение. Все модули работают непрерывно и параллельно. В программных модулях в каждый момент времени выполняется одна функция (**F**). Если в теле функции **F** в выражении встречается **вызов –** имя другой функции (**G**), то между ними устанавливается временная (**динамическая**) связь: выполнение первой функции прекращается до тех пор, пока не выполниться вторая. Этот принцип выполнения называется **вложенностью вызовов функций** и может быть повторен многократно.



**Рис. 34-1. Статическое связывание модулей и динамическое связывание вызовов функций**

Итак, первое, в чем нельзя ошибаться: функции синтаксически записываются как **независимые**модули, установление связей между ними через вложенные вызовы производится в процессе выполнения, то есть динамически.

Далее необходимо установить различие между формальными и фактическими параметрами. Прежде всего, это два разных взгляда на программный интерфейс функции. Формальные параметры – это описание этого интерфейса изнутри. Оно дается в виде **определения переменных,**то есть описания свойств объекта, который может быть передан на вход. Имя фактического параметра – это обобщенное (абстрактное) обозначение некоторой переменной, видимой в процессе работы функции **изнутри.** Например, функция обрабатывает абстрактный массив с именем **A** и размерностью **n.** При вызове функции в списке присутствуют фактические параметры, имеющие синтаксис **выражений**, то есть уже определенных переменных или промежуточных результатов, которые в данном вызове **ставятся в соответствие**формальным параметрам. Таким образом, они представляют взгляд на тот же самый интерфейс, но уже со стороны вызывающей функции.



**Рис. 34-2. Интерфейс передачи параметров в функцию**

Итак, формальные и фактические параметры имеют принципиально разный синтаксис: описания переменных (определения) и использования их (выражения). Связь между ними также устанавливается в момент вызова динамически.

Главное, к чему необходимо привыкнуть: функция пишется для обработки данных **вообще,** то есть представляет собой обобщенное описание алгоритма для некоторых произвольных данных, имена которых представляют собой их «будущие обозначения» в процессе работы функции. Что же касается транслятора, то для него формальные параметры представляют собой «ожидаемые на входе значения», своего рода «заглушки», поэтому функция и транслируется применительно к имеющимся определениям (именам и типам).

Вызов функции, наоборот, представляет собой выполнение **частный** случай выполнения алгоритма для конкретных данных.

Рассмотренная модель может быть применима сама к себе: реальная программа представляет собой иерархию вызовов функции, а формальные параметры функции верхнего уровня могут быть фактическими параметрами в следующем (вложенном) вызове.

Итак, главное необходимое условие модульного программирования – научиться абстрагироваться от конкретных обрабатываемых данных и выносить их «за пределы» проектируемого алгоритма.

По отношению к результату функции можно сформулировать те же самые принципы: результат – это обобщенное значение, которое возвращается после вызова функции в конкретное выражение, где расположен вызов.

**Модульность формальная и истинная.** Формально соблюдаемая модульность – синтаксическая: программа состоит из множества вызывающих друг друга функций (модулей), размер модуля ограничен определенным числом строк текста программы. Но не любая программа, разбитая на функции, будет модульной. Соблюдение духа, но не буквы модульного программирования, требует соблюдения следующих принципов:

        **логическая завершенность.**Функция (модуль) должна реализовывать логически законченный, целостный алгоритм;

        **ограниченность.**Функция (модуль) должна быть ограничена в размерах, в противном случае ее необходимо разбить на логически завершенные части - модули, вызывающие друг друга;

        **замкнутость.**Функция (модуль) не должна использовать глобальные данные, иметь «связь с внешним миром» помимо программного интерфейса, не должна содержать ввода и вывода результатов во внешние потоки - результаты должны быть размещены в структурах данных;

        **универсальность.**Функция (модуль) должна быть универсальна, параметры процесса обработки и сами данные должны передаваться извне, а не должны подразумеваться или устанавливаться постоянными;

        **принцип «черного ящика».**Функция (модуль) должна иметь продуманный «программный интерфейс» - набор фактических параметров и результат функции, через который она «подключается» к другим частям программы (вызывается).

**Естественное появление модулей при разработке программы.**В процессе проектирования модули возникают естественным образом в процессе разбиения решаемой задачи на подзадачи, либо при выделении фрагментов некоторой задачи в отдельную подзадачу. Кроме того, функции (модули) могут иметь и «технологическое» происхождение. Если некоторая задача в процессе проектирования порождает значительное число программных контекстов и имеет значительный уровень вложенности управляющих конструкций, то в ней можно выделить подзадачу, оформив ее в виде функции с однократным вызовом.

Второй случай появления модулей связан с эффектом «де жа вю»: если в процессе разработки алгоритма возникает непреодолимое желание повторить уже выполненную последовательность действий, возможны следующие варианты:

        выполнить goto к имеющемуся фрагменту (категорически не рекомендуется);

        повторить текст фрагмента в новом месте (не эффективно);

        оформить повторяющийся фрагмент в виде модуля с вызовом в двух точках программы (лучше всего).

 Примеры модульного проектирования

**Супер-простое число.** Число 1997 обладает замечательным свойством: само оно простое, простыми также являются любые разбиения его цифр на 2 части, то есть 1-997, 19-97, 199-7. Требуется найти все такие числа для заданного количества значащих цифр.

Поскольку проверка, является ли число простым, будет применяться многократно по отношению как к самому числу, так и к его частям, функцию проверки, является ли заданное число простым, оформим в виде функции, применив тем самым принцип модульного программирования.

//------------------------------------------------------34-01.cpp

//----- Функция проверки, является ли число простым

 int PR(int a){

 if (a==0) return 0;                      // 0 это не простое число

 for ( int n=2; n<a; n++)

      { if (a%n==0) return 0 ; }       // Если делится, можно выйти сразу

 return 1;}                                  // Дошли до конца - простое

Дополнительная проверка «крайностей»: 1 является простым числом, но для нее цикл не выполнится ни разу и будет возвращена «истина». 0, вообще говоря, простым числом не является, поэтому должен быть «отсечен».

1. Сам алгоритм представляет собой полный перебор **n-**значных чисел. Прежде всего, необходимо получить сам диапазон. Для этого 1 умножается в цикле **n** раз на 10. Верхняя граница – в 10 раз больше нижней. Полученные числа сохранять не будем, будем просто выводить.

void super(int n){

long v,a; int i;

for (v=1,i=0; i<n; i++) v\*=10;                    // Определение нижней границы

for (a=v; a<10\*v; a++){

            // Проверить число на супер-простоту

            if (… суперпростое…) cout << a << endl;

}}

2. Фрагмент проверки является скорее «технологической заглушкой», обозначающей общую логику процесса. Саму проверку удобнее произвести по принципу просеивания: если очередное условие не соблюдается, выполняется переход к следующему шагу цикла оператором **continue.**Первое условие, что само число является простым, проверяется вызовом функции. Сложнее проверить его части. Для получения частей необходимо рассмотреть частные и остатки от деления этого числа на 10,100 и т.д. до **v -**нижней границы диапазона. Если хотя бы одно частное или остаток из них не является простым, то все число также не является супер-простым. Грубо процесс проверки можно представить так.

If (PR(a)==0) continue;

for (long ll=10; ll<v; ll\*=10){                      // ll пробегает значения 10,100, 1000 < v

            … PR(a/ll)…                              // Проверка старшей части

            … PR(a%ll)…                            // Проверка младшей части

            }

if (…все части простые…) cout << a << endl;

3. В предыдущем варианте мы отступили от принципа нисходящего проектирования, поскольку сначала требовалось сформулировать условие: проверить, являются ли все части числа простыми, из чего следует, что написанный нами вчерне процесс проверяет условие всеобщности. Для его реализации будем использовать стандартный контекст с **break.**Если условие не соблюдается (то есть выполняется обратное), то происходит досрочный выход, тогда «естественное» достижение конца цикла по условию, стоящему в заголовке, говорит о справедливости свойства всеобщности.

//------------------------------------------------------34-02.cpp

//------- Супер-простое число с n значащими цифрами

 void super(int n){

 long v,a; int i;

 for (v=1,i=0; i<n; i++) v\*=10;              // Определение нижней границы

 for (a=v/10; a<v; a++){

if (PR(a)==0) continue;

            for (long ll=10; ll<v; ll\*=10){     // ll пробегает значения 10,100, 1000 < v

if (PR(a/ll)==0)          // Проверка старшей части

                        break;                      // Не простое досрочный выход

                        if (PR(a%ll)==0)        // Проверка младшей части

                        break;                      // Не простое досрочный выход

                        }

            if (ll==v)                                     // Достигли конца все простые

printf("super=%ld\n",a);

}}

**Сортировка слов в строке.**Требуется отсортировать слова в строке: получить выходную строку со словами, упорядоченными по различным критериям: по длине или по алфавиту.

Возможны различные варианты решения этой задачи, в частности, можно попытаться по максимуму свести эту задачу к стандартным. Например, запомнить в массиве пары характеристик каждого слова: начало и длина. Затем отсортировать массив по длинам **любым**известным способом. После чего переписать слова в выходную строку, зная их начала в исходной.

Мы рассмотрим более компактный вариант, использующий естественное представление и преобразование данных. Если найти в строке слово с минимальной характеристикой (минимальной длины или первое по алфавиту), то после переписывания в выходную строку его можно удалить, заменив пробелами. Последовательное применение этого действия к исходной строке даст нам сортировку **выбором.** Программу в такой постановке можно разбить на три модуля:

        поиск слова с минимальной характеристикой;

        основная функция – выбор и переписывание слов;

        функция сравнения двух слов в одной строке по алфавиту.

Для начала

При помощи функции поиска можно выполнить упорядочение слов по длине. Данный пример является хорошей иллюстрацией сущности сортировки выбором, приведенной в 2.6 для обычных массивов: из входного множества объектов (последовательности) выбирается минимальный (максимальный) и переносится в выходное. Наглядность программы состоит в том, что найденное слово удаляется из входной строки путем его «забивания» пробелами.

Выделение отдельного модуля и его проектирование начинается с определения интерфейсов – заголовков функций. Если функция поиска слова с минимальной характеристикой возвращает индекс его в строке (или -1 при отсутствии слов), то основная функция сортировки записывается в виде двойного цикла.

//------------------------------------------------------34-03.cpp

///---- Сортировка слов в строке в порядке возрастания (выбором)

 void sort(char in[], char out[]){

 int i=0,k;

 while((**k=find1(in)**)!=-1) {                        // Получить индекс очередного слова

            for (; in[k]!=' ' && in[k]!=0; i++,k++) {

                        out[i]=in[k]; in[k]=' ';       // Переписать с затиранием

                        }

            out[i++]=' ';                                 // После слова добавить пробел

            }

 out[i]=0;}                                              // В конце – конец строки

Используется присваивание результата вызываемой функции «на лету» **k=find1(in**), т.к. он проверяется на -1, а затем используется в теле цикла. В теле цикла происходит простое копирование символов, во входной строке используется индекс **k –**возвращенный функцией поиска, в выходной строке – собственный линейно изменяющийся индекс записи – **i.**Одновременно после копирования символ заменяется на пробел.

Варианты функции поиска слова с минимальной характеристикой будем разрабатывать по технологии «грязного» программирования (см.**3.5**). Функция-заготовка просматривает строку по словам (см. **4.4**). Структурные компоненты строки – цепочка пробелов и цепочка символов слова имеют каждый свой цикл просмотра.  В программе можно выделить точку, где она находится в момент обнаружения начала очередного слова.

//------------------------------------------------------34-03.cpp

//---- Поиск слова с минимальной характеристикой

// "грязная программа" - пословная обработка

 void find0(char in[]){

 int i=0;

 while (in[i]!=0) {                         // Цикл пословного просмотра строки

             while (in[i]==' ') i++;       // Пропуск пробелов перед словом

             if (in[i]==0) return;         // После пробелов нет слова - выход

             //**Начало очередного слова - i**

             for (k=0;in[i]!=' ' && in[i]!=0; i++);

             }                                  // Цикл просмотра слова

 }

Для варианта с выбором слова минимальной длины можно просто доработать эту функцию: внести в цикл, движущийся по символам слова, счетчик длины и добавить контекст сравнения на минимум.

//---- Поиск слова максимальной длины пословная обработка

 int find1(char in[]){

 int i=0, k, m=0, b=-1;

 while (in[i]!=0) {                         // Цикл пословного просмотра строки

            while (in[i]==' ') i++;        // Пропуск пробелов перед словом

            if (in[i]==0) return b;

            for (k=0;in[i]!=' ' && in[i]!=0; i++,k++);      // Подсчет длины слова

            if (b==-1 || k<m){            // Контекст выбора минимума

                        m=k; b=i-k; }      // Одновременно запоминается

            }                                   // индекс начала

 return b; }

Для варианта сортировки по алфавиту удобнее вынести сравнение двух слов в строке в отдельную функцию, хотя она и будет вызываться один раз. Это нужно, чтобы изменения, вносимые в «грязную» программу, были минимальными и обозримыми. Сама функция представляет собой простой цикл попарного сравнения с обнаружением первого расхождения.

//------------------------------------------------------34-03.cpp

#define CMP(i) (c[i]==0 || c[i]==' ')            // Определение подстановки - проверка

 //---- Сравнение слов в строке              // символа на конец слова

 int my\_cmp(char c[], int i1, int i2){

 while(1){                                               // Вечный цикл с определением

            if (CMP(i1) && CMP(i2))              // первого расхождения

                        return 0;                        // Кончились одновременно - равны

            if (CMP(i1)) return -1;                  // Одно кончилось раньше другого

            if (CMP(i2)) return 1;

            if (c[i1]!=c[i2]) return c[i1]-c[i2];    // Обнаружено расхождение

            i1++; i2++;                                // Иначе – к следующей паре

            }

 }

В функцию поиска минимального слова по алфавиту в точке обнаружения начала слова нужно вставить стандартный контекст запоминания минимального, в котором вызывается функция сравнения для текущего слова (индекс **i**) и слова, которое считается минимальным (индекс **b**). Для первого сравнения устанавливается и проверяется «защелка» **b=-1.**

//------------------------------------------------------34-03.cpp

 //---- Поиск слова минимального по алфавиту

 int find2(char in[]){

 int i=0, k, m, b;

 b=-1; m=0;

 while (in[i]!=0) {                                     // Цикл пословного просмотра строки

            while (in[i]==' ') i++;                    // Пропуск пробелов перед словом

            if (in[i]==0) return b;

            if (b==-1 || my\_cmp(in,i,b)<0)      // Начало очередного слова

                        b=i;                               // i - очередной, b - оптимальное

            for (k=0;in[i]!=' ' && in[i]!=0; i++);

            }

 return b; }