1 Занятие №8

1.1 Квалификатор `const`

При определении переменной её тип может дополняться так называемыми квалификаторами. Язык Си определяет два квалификатора **`const`** и **`volatile`**. Квалификатор **`volatile`** используется, в основном, для программирования на низком уровне, поэтому мы его рассматривать не будем. Квалификаторы могут стоять на любом месте в спецификации типа: до имени типа, после имени типа, и даже в середине имён типов, состоящих из нескольких ключевых слов, например:

```
const unsigned long
unsigned long const
unsigned const long
```

— все правильные спецификации типа и описывают один и тот же тип.

Квалификатор **`const`** означает, что определяемый объект не может быть модифицирован, потому что например, находится в ПЗУ.

```
const int nproc = 30;
```

определяет переменную `nproc` типа **`int`**, которая не может быть модифицирована в данной единице компиляции. Переменная, описанная с квалификатором **`const`**, не может быть использована в константных выражениях, которые, в частности, задают количество элементов массива. Следующий фрагмент не является правильным в языке Си:

```
const int N = 10;
double arr[N];
```

Если квалификатор **`const`** используется для определения переменной-указателя, его семантика меняется в зависимости от того, где он расположен: до символа `*` или после него. Определение

```
char *ptr;
```

вводит указатель `ptr`, который и сам может изменяться, и память по адресу, на который указывает данный указатель, также может быть изменена.

```
char const *ptr;
```

определяет указатель `ptr` на неизменяемую область памяти. Сам указатель может изменяться.

```
char * const ptr;
```

определяет неизменяемый указатель на изменяемую область памяти.

```
char const * const ptr;
```

определяет неизменяемый указатель на неизменяемую область памяти.

Из всех вышеперечисленных комбинаций чаще всего используется **`const char *`**. Такой тип, например, имеют параметры многих стандартных функций. Например, функция strcmp определена следующим образом:

```
int strcmp(const char *s1, const char *s2);
```

потому что она принимает в качестве параметров две строки, которые не модифицирует. Обратите внимание, что не имеет смысла писать **const char * const**, потому что параметры указательных типов в функции передаются по значению. *Рекомендуется пользоваться квалификатором* ***const*** *в ваших программах.*

1.2 Чтение сложных деклараторов

Ранее мы уже рассмотрели различные конструкции, модифицирующие тип, такие как указатели, массивы, функции. Все эти конструкции могут комбинироваться в синтаксической конструкции, называемой «декларатором». Таким образом, полное определение переменной выглядит следующим образом:

```
<базовый тип> <декларатор> [ = <инициализатор> ];
```

Декларатор содержит имя определяемого объекта, но в некоторых местах может быть «анонимным», то есть не содержащим имя определяемого объекта. Анонимные деклараторы допускаются в операции приведения типа и при описании формальных параметров в прототипах функций.

Пример декларатора:

```
char (*(*x[3])())[5];
```

Анонимный декларатор может выглядеть следующим образом:

```
char (*(*[3])())[5];
```

Такая (на первый взгляд «странная») форма определения производных типов на самом деле введена по аналогии с выражениями. Декларатор можно рассматривать как некоторое выражение над типом. В таком выражении есть три операции:

`[]`	постфиксная — массив из заданного количества элементов
`()`	постфиксная — функция с заданными параметрами
`*`	префиксная — указатель
`()`	группировка членов в выражении

Постфиксные операции имеют самый высокий приоритет и читаются слева направо от определяемого имени. Префиксная операция имеет более низкий приоритет и читается справа налево. Скобки могут использоваться для изменения порядка чтения.

Таким образом, декларатор читается, начиная от имени определяемого объекта следуя правилам приоритетов операций. Имя определяемого объекта — это первое имя после базового типа.

Примеры:

`int a[3][4];`	массив из 3 элементов типа массива из 4 элементов типа **int** (матрица 3×4 целых)
`char **b;`	указатель на указатель на **char**
`char *c[];`	массив из неопределённого количества элементов типа указатель на тип **char**
`int *d[10];`	массив из 10 элементов типа указатель на тип **int**
`int (*e)[10];`	указатель на массив из 10 элементов типа **int**
`int *f();`	функция, возвращающая указатель на **int**
`int (*g)();`	указатель на функцию, возвращающую **int**
`int *(*g)();`	указатель на функцию, возвращающую указатель на **int**

1.3 Класс декларации typedef

Чтобы не нагромождать деклараторы и облегчить их чтение, введено специальное ключевое слово **typedef**. Оно записывается перед именем базового типа в декларации, например

```
typedef int *pint;
```

В этом случае имя `pint` определяется как синоним для типа `int *`, то есть далее в определениях переменных это имя можно использовать наравне с именем базового типа, например

```
pint a[10], f(), *p;
```

Конструкция **typedef** не вводит новый тип, а задаёт ещё одно имя для типа, которое может использоваться наравне со старым. Поэтому переменная `pint a;` и переменная `int *b;` имеют один и тот же тип `int *`.

Если есть **typedef**-имя и декларация, использующая это имя, то от **typedef**-имени можно избавиться, подставив декларируемое имя вместо **typedef**-имени в **typedef**-декларацию и добавив при необходимости скобки для того, чтобы порядок чтения не изменился. Например,

```
typedef void (*pfunc)(int);
pfunc signal(int, pfunc);
```

после преобразования получаем

```
void (*signal(int, void (*)(int)))(int);
```

1.4 Работа с бинарными файлами

Под *бинарным* файлом мы будем понимать файл, удовлетворяющий следующим условиям:

- файл рассматривается как последовательность байт, никакого деления файла на строки не подразумевается;
- данные в файле хранятся в двоичном, а не текстовом представлении.

Конечно, деление на текстовые и бинарные файлы достаточно условно. Текстовый файл иногда может оказаться удобно рассматривать как бинарный и работать с ним соответствующим образом.

Мы рассмотрим работу с бинарными файлами произвольного доступа, то есть с файлами, которые допускают произвольное позиционирование указателя текущего положения в файле, а не просто последовательное чтение от начала до конца.

1.4.1 Открытие бинарных файлов

В системах Unix формат текстовых и бинарных файлов совпадает, в других же системах эти два типа файлов могут обрабатываться по-разному. Поэтому при открытии бинарного файла с помощью функции `fopen` необходимо использовать специальный флаг открытия `"b"`. С учётом этого флага допустимые режимы открытия файла перечислены в таблице 1.

"rb"	Открыть для чтения. Если файл не существует, fopen завершается с ошибкой. Текущая позиция в файле устанавливается на начало файла.
"wb"	Открыть для записи. Если файл не существовал, он создаётся, если существовал — очищается. Текущая позиция в файле устанавливается на начало файла.
"ab"	Открыть для добавления. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла. Каждая операция записи будет перемещать указатель текущего положения в конец файла, затем записывать данные.
"r+b"	Открыть для чтения и записи. Если файл не существует, fopen завершается с ошибкой. Текущая позиция в файле устанавливается на начало файла.
"w+b"	Открыть для чтения и записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
"a+b"	Открыть для чтения и записи. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.

Таблица 1: Режимы открытия файлов функции fopen для бинарных файлов

1.4.2 Чтение — функция fread

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *f);
```

Функция fread считывает данные из бинарного файла. Параметр ptr — это адрес начала буфера, в который будут записаны считанные данные. size — это размер одного элемента данных, а nmemb — это количество элементов данных, которые необходимо прочитать. f — дескриптор потока, из которого ведётся чтение.

Общее количество байт, которые необходимо прочитать, определяется перемножением параметров size и nmemb

```
bytes_to_read = size * nmemb;
```

Далее делается попытка считать данное количество байт из дескриптора потока. Предположим, что было успешно считано read_bytes байт. Тогда возвращаемое значение функции fread вычисляется следующим образом:

```
retval = read_bytes / size;
```

Таким образом, возвращаемое значение — это количество элементов данных, которые были считаны целиком. В случае ошибки или наступления конца файла возвращается количество элементов, которые были полностью считаны до возникновения ошибки или конца файла.

Поскольку возвращаемое значение получается делением нацело на размер одного элемента данных, если размер одного элемента больше одного байта, и будет считано количество байт, не кратное размеру одного элемента, неполный элемент будет записан в память, но способа узнать о том, что он был записан, не существует. Поэтому размер одного элемента данных всегда нужно задавать равным 1, а возвращаемое значение сравнивать с запрошенным количеством байт. Здесь возможны следующие ситуации:

- Считано 0 байт. Это значит, что наступил конец файла, или возникла ошибка чтения.

Чтобы узнать, завершилась ли функция `fread` по ошибке или по концу файла, нужно использовать функции `feof` и `ferror`.

- Считано байт меньше, чем запрошено, но количество байт кратно размеру одного элемента данных. В этом случае нужно обработать считанное количество элементов.
- Считано байт меньше, чем запрошено, и количество байт не кратно размеру одного элемента данных. В этом случае можно выдать ошибку о нарушении формата входных данных.
- Считано байт ровно столько, сколько запрошено. Нужно обработать считанные элементы.

1.4.3 Запись — функция `fwrite`

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *f);
```

Функция `fwrite` записывает данные в дескриптор потока. Параметр `buf` — это адрес начала буфера, в котором хранятся записываемые данные, `size` — это размер одного элемента данных, а `nmemb` — это количество элементов данных, которые необходимо записать. `f` — дескриптор потока, в который ведётся запись.

Общее количество байт, которые необходимо записать, вычисляется аналогично функции `fread`. Возвращаемое значение также получается делением нацело действительно записанного количества байт на размер одного элемента данных.

1.4.4 Позиционирование в файле

```
#include <stdio.h>

#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

Функция `ftell` позволяет получить смещение указателя на текущее положение в файле в байтах относительно начала файла. При ошибке функция возвращает -1.

Функция `fseek` позволяет переместить указатель на текущее положение в файле. Параметр `stream` задаёт дескриптор потока, связанный с файлом, параметр `offset` задаёт смещение в байтах, а параметр `whence` — точку, от которой отсчитывается смещение. Этот параметр может принимать три значения, перечисленные ниже.

`SEEK_SET`	смещение отсчитывается от начала файла
`SEEK_CUR`	смещение отсчитывается от текущего положения в файле
`SEEK_END`	смещение отсчитывается от конца файла

Если новое положение в файле находится за текущим концом файла, и файл открыт на запись или чтение/запись, файл расширяется нулями до требуемого размера. Если новое положение в файле находится до начала файла, возвращается ошибка.

При успешном завершении функция `fseek` возвращает 0, а при ошибке — -1.

Обратите внимание, что функции позиционирования могут быть неприменимы к стандартным потокам, потому что стандартные потоки могут быть связаны с устройствами, которые не допускают произвольное позиционирование (например, терминалы).

1.4.5 Изменение размера файла

```
#include <stdio.h>
#include <unistd.h>

int truncate(const char *path, off_t length);
```

Функция `truncate` позволяет изменить (увеличить или уменьшить) размер существующего файла с путём `path`. Новый размер файла будет равен значению, заданному в параметре `length`. Если файл увеличивается в размере, новая часть инициализируется нулями. Если файл уменьшается, старый остаток файла теряется.

1.4.6 Вычисление размера файла

Приведённая ниже функция позволяет получить размер файла по его имени. В качестве параметра ей передаётся имя файла, она возвращает размер файла при успешном завершении и -1 при ошибке.

```
#include <stdio.h>

long fsize(char const *path)
{
  FILE *f = 0;
  long size;

  if (!(f = fopen(path, "r"))) return -1;
  if (fseek(f, 0, SEEK_END) < 0) goto error;
  if ((size = ftell(f)) != -1) goto error;
  fclose(f);
  return size;

 error:
  if (f) fclose(f);
  return -1;
}
```

1.4.7 Пример программы

Следующая программа принимает в качестве параметров командной строки список имён файлов. Предполагается, что эти файлы содержат целые числа в бинарном виде. Программа увеличивает все числа в файле на единицу.

```
#include <stdio.h>

int process_file(char const *path)
{
```

```
  FILE *f = 0;
  int   data;
  int   rb;

  if (!(f = fopen(path, "r+b"))) {
    fprintf(stderr, "cannot␣open␣file␣`%s'\n", path);
    return 1;
  }
  while ((rb = fread(&data, 1, sizeof(data), f)) == sizeof(data)) {
    if (fseek(f, -sizeof(data), SEEK_CUR) < 0) {
      fprintf(stderr, "seek␣error␣in␣`%s'\n", path);
      fclose(f);
      return 1;
    }
    data++;
    if (fwrite(&data, 1, sizeof(data), f) != sizeof(data)) {
      fprintf(stderr, "write␣error␣to␣`%s'\n", path);
    }
  }
  if (ferror(f)) {
    fprintf(stderr, "read␣error␣from␣`%s'\n", path);
    fclose(f);
    return 1;
  }
  if (rb > 0) {
    fprintf(stderr, "format␣error␣in␣`%s'\n", path);
    fclose(f);
    return 1;
  }
  if (fclose(f) < 0) {
    fprintf(stderr, "write␣error␣to␣`%s'\n", path);
    return 1;
  }
  return 0;
}

int main(int argc, char *argv[])
{
  int retval = 0;
  int i;
  for (i = 1; i < argc; i++) {
    retval |= process_file(argv[i]);
  }
  return retval;
}
```

1.4.8 Упражнения

1. Разберите следующие деклараторы

```
struct foo *f(int (*)(int));
```

```
const char * (*x)[2][2];
unsigned long * const (* const z[2])[4];
```

2. В аргументах командной строки задаётся имя бинарного файла, хранящего дерево поиска, и целое число - ключ поиска в файле. Необходимо напечатать значение, соответствующее заданному ключу, хранящееся вместе с ключём в бинарном дереве поиска в файле.

   Бинарное дерево поиска в файле устроено следующим образом. Каждая запись занимает 4 слова (по 32 бита каждое слово). Первое слово — это ключ, второе слово — это значение, третье слово — это номер записи, соответствующей вершине левого поддерева, и четвёртое слово — это номер записи, соответствующей вершине правого поддерева. Нулевая запись в файле не используется, и зарезервирована для обозначения «нулевого указателя». Корень всего дерева находится в первой записи.