Область видимости, время жизни, связывание

Область видимости

Область видимости (scope) имени – это часть текста программы, в пределах которой имя может быть использовано.

В языке Си выделяют следующие области видимости

- блок;
- файл;
- функция;
- прототип функции.

Область видимости: блок

В языке Си *блоком* считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки.

Существуют два вида блоков:

- составной оператор;
- определение функции.

Блоки могут включать в себя составные операторы, но не определения функций.

Область видимости: блок

- Переменная, определенная внутри блока, имеет область видимости в пределах блока.
- Формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.

```
double f(double a)  // начало области видимости переменной a
{
    double b;       // начало области видимости переменной b
    ...
    return b;
}                   // конец области видимости переменных a и b
```

Область видимости: файл

- Область видимости в пределах файла имеют имена, описанные за пределами какой бы то ни было функции.
- Переменная с областью видимости в пределах файла видна на протяжении от точки ее описания и до конца файла, содержащего это определение.
- Имя функции всегда имеет файловую область видимости.

Область видимости: файл

```
#include <stdio.h>

int max;

void f(void)
{
    printf("%d\n", max);
}

int main(void)
{
    max = 5;
    f();
    ...
}
```

Область видимости: функция

- Метки - это единственные идентификаторы, область действия которых - функция.
- Метки видны из любого места функции, в которой они описаны.
- В пределах функции имена меток должны быть уникальными.

Область видимости: прототип функции

Область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.

```
int f(int i, double d);
```

Область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа.

```
int f(int, double);        // ок
int f(int i, double i);    // ошибка компиляции
```

Область видимости: прототип функции

Один из тех случаев, когда имена имеют значение – это параметры типа массивов переменной длины:

```
int print_matrix(int n, int m, double a[n][m]);
```

Правила перекрытия областей видимости

Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный.

```
{
    int a = 1;
    ...
    {
        int b = 2;
        ...
        printf("%d %d\n", a, b); // ок
    }

    printf("%d %d\n", a, b); // ошибка компиляции
}
```

Правила перекрытия областей видимости

Возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных.

```
{
    int a = 1;

    {
        int a = 2;

        printf(“%d\n”, a);     // 2
    }

    printf(“%d\n”, a);         // 1
}
```

Время жизни

Время жизни (storage duration) – это интервал времени выполнения программы, в течение которого «программный объект» существует.

В языке Си время жизни «программного объекта» делится на три категории

- глобальное (по стандарту - статическое (англ. static));
- локальное (по стандарту - автоматическое (англ. automatic));
- динамическое (по стандарту - выделенное (англ. allocated)).

Глобальное время жизни

Если «программный объект» имеет глобальное время жизни, он существует на протяжении выполнения всей программы.

Примерами таких «программных объектов» могут быть функции и переменные, определенные вне каких либо функций.

Локальное время жизни

Локальным временем жизни обладают «программные объекты», область видимости которых ограничена блоком.

Такие объекты создаются при каждом входе в блок, где они определяются. Они уничтожаются при выходе из этого «родительского» блока.

Примерами таких переменных являются локальные переменные и параметры функции.

Динамическое время жизни

Время жизни «выделенных» объектов длится с момента выделения памяти и заканчивается в момент ее освобождения.

В Си нет переменных, обладающих динамическим временем жизни.

Динамическое выделение выполняется программистом «вручную» с помощью соответствующих функций. Единственные способ «добраться» до выделенной динамической памяти – использование указателей.

Связывание

Связывание (linkadge) определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы.

Стандарт языка Си определяет три формы связывания:

- внешнее (external);
- внутреннее (internal);
- никакое (none).

Связывание

Имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

Имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.

Имена без связывания принадлежат одной функции и не могут разделяться вообще.

Время жизни, область видимости и связывание переменной зависят от места ее определения. По умолчанию

```
int i;       // глобальная переменная
```

- Глобальное время жизни
- Файловая область видимости
- Внешнее связывание

```
{
    int i;  // локальная переменная
    ...
```

- Локальное время жизни
- Видимость в блоке
- Отсутствие связывания

Классы памяти

Управлять временем жизни, областью видимости и связыванием переменной (до определенной степени) можно с помощью так называемых классов памяти.

В языке Си существует четыре класса памяти

- auto;
- static;
- extern;
- register.

Класс памяти auto

Применим только к переменным, определенным в блоке.

```
int main(void)
{
    auto int i;
```

Переменная, принадлежащая к классу auto, имеет *локальное время жизни, видимость в пределах блока, не имеет связывания*.

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

Класс памяти static

Класс памяти static может использоваться с любыми переменными независимо от места их расположения.

- Для переменной вне какого-либо блока, static изменяет связывание этой переменной на внутреннее.
- Для переменной в блоке, static изменяет время жизни с автоматического на глобальное.

Класс памяти static

Статическая переменная, определенная вне какого-либо блока, имеет *глобальное время жизни, область видимости в пределах файла и внутреннее связывание*.

```
static int i;
void f1(void)
{
    i = 1;
}
void f2(void)
{
    i = 5;
}
```

Этот класс памяти скрывает переменную в файле, в котором она определена.

Класс памяти static

Статическая переменная, определенная в блоке, имеет *глобальное время жизни, область видимости в пределах блока и отсутствие связывания*.

```
void f(void)
{
    static int j;
    ...
}
```

- Такая переменная сохраняет свое значение после выхода из блока.
- Инициализируется только один раз.
- Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

Класс памяти extern

Помогает разделить переменную между несколькими файлами.

```
// file_1.c
int number;

// file_2.c
int process(void)
{
    if (number > 5)
    ...
```

error: ‘number' undeclared

```
// file_1.c
int number;

// file_2.c
extern int number;

int process(void)
{
    if (number > 5)
    ...
```

OK

Класс памяти extern

Используется для переменных определенных как в блоке, так и вне блока.

```
extern int i;
```

Глобальное время жизни, файловая область видимости, связывание непонятное

```
{
    extern int i;
    ...
```

Глобальное время жизни, видимость в блоке, связывание непонятное

Связывание определяется по определению переменной.

Класс памяти extern

- Объявлений (extern int number;) может быть сколько угодно.
- Определение (int number;) должно быть только одно.
- Объявления и определение должны быть одинакового типа.

Замечание.

```
extern int number = 5; // определение
```

Класс памяти extern

6.2.2 #4 For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

6.2.2 #7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

`static int i;`
`extern int i;`
// ОК
	`extern int i;`
	`static int i;`
	// ОШИБКА

Класс памяти register

Использование класса памяти register – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора.

- Используется только для переменных, определенных в блоке.
- Задает *локальное время жизни, видимость в блоке и отсутствие связывания*.
- Обычно не используется.

К переменным с классом памяти register нельзя применять операцию получения адреса &.

Классы памяти и функции

К функциям могут применяться классы памяти static и extern.

```
extern int f(int i);
```

f имеет внешнее связывание. Может вызываться из других файлов.

```
static int g(int i);
```

g имеет внутренне связывание. Из других файлов вызываться не может.

```
int h(int i);
```

h имеет внешнее связывание (по умолчанию). Может вызываться из других файлов.

Классы памяти и функции

Использование класса памяти static для функций полезно потому что:

- Функции, определенные со static, невидны в других файлах и могут безболезненно изменяться (инкапсуляция).
- Так как функция имеет внутренне связывание, ее имя может использоваться в других файлах.

Аргументы функций по умолчанию имеют класс памяти *auto*. Единственный другой класс памяти, который может использоваться с параметрами функций, - *register*.

Недостатки использования функций, которые используют глобальные переменные:

- Если глобальная переменная получает неверное значение, трудно понять какая функция работает неправильно.
- Изменение глобальной переменной требует проверки правильности работы всех функций, которые ее используют.
- Функции, которые используют глобальные переменные, трудно использовать в других программах.

```
// log.c

#include <stdio.h>

FILE *flog;

int log_init(const char
                          *name)
{
    flog = fopen(name, "w");
    if(!flog)
        return 1;

    return 0;
}

void log_close(void)
{
    fclose(flog);
}
```

```
// log.h

#ifndef __LOG__H__

#define __LOG__H__

extern FILE *flog;

int log_init(const char
                        *name);

void log_close(void);

#endif  //  __LOG__H__
```

```
// main.c

#include <stdio.h>
#include "log.h"

void func_1(void)
{
    fprintf(flog, "func_1\n");
}

void func_2(int a)
{
    fprintf(flog, "func_2(%d)\n", a);
}

int main(void)
{
    if(log_init("test.log"))
```

```
    {
        printf("Could not create log file");
        return -1;
    }

    func_1();
    func_2(5);

    log_close();

    return 0;
}
```

```
// log.c

#include <stdio.h>

static FILE *flog;

int log_init(const char
                        *name)
{
    flog = fopen(name, "w");
    if(!flog)
        return 1;

    return 0;
}

FILE* log_get(void)
{
    return flog;
}

void log_close(void)
{
    fclose(flog);
}
```

```
// log.h

#ifndef __LOG__H__

#define __LOG__H__

#include <stdio.h>

int log_init(const char
                    *name);

FILE* log_get(void);

void log_close(void);

#endif  //  __LOG__H__
```