**Основные характеристики языка Си**

***Конструирование ПО (часть 2)***

*Общая характеристика языка*

Чувствителен к регистру, т.е. abcd и AbCd – два разных идентификатора.

Разбиение программы на строки, как правило, не имеет значения. (исключения есть)

*Основные типы данных*

Тип void

Базовые типы:

* Знаковые и беззнаковые целочисленные
* Вещественные

Производные типы:

* Массивы
* Указатели
* Тип «функция»
* Структуры
* Объединения

*Тип void*

Множество значений – пустое

* Нет значений
* Нет операций
* Нет свойств значений

Используется для обозначения отсутствия информации о типе

*Целочисленные типы данных*

Точные размеры и диапазоны для целочисленных типов стандартом не определены.

* Заданы только минимальные

Есть без/знаковые

Модификаторы:

* Unsigned – беззнаковый
* Signed – знаковый (по умолчанию) (но это неточно)

int/signed int/signed [-32767; 32767]

unsigned int/ unsigned [0; 65535]

short/short int/ signed short/ signed short int -32767; 32767

unsigned short int/ unsigned short 0; 65535

long/long int/signed long/signed long int –(2^31-1)..(2^31-1)

unsigned long/unsigned long int 0..(2^32-1)

long long/long long int/signed long long/signed long long int –(2^63-1)..(2^63-1)

unsigned long long/unsigned long long int 0..(2^64-1)

char (как signed так и unsigned, в настройках)

signed char -127..127

unsigned char 0.255

Требования стандарта к байту:

* Байт – адресуемый элемент, достаточный для хранения одного символа базового набора символов
* Каждый байт любого значения можно адресовать независимо.
* Байт состоит из непрерывной последовательности бит, количество которых определяется реализацией

Количество бит в байте задается константой CHAR\_BIT (определена в модуле limits.h)

По стандарту минимальное значение CHAR\_BIT равно 8

Microsoft – (unsigned) \_\_int[8/16/32/64]

Определяются разработчиками компилятора в модуле stdint.h –

intN\_t/uintN\_t – знаковый/беззнаковый, ровно N

int\_leastN\_t/uint\_leastN\_t – знаковый/беззнаковый, не менее N

*Целочисленные литералы*

Десятичная система счисления:

* 42, 5321, 214

8-чная система счисления:

* 037, 0463, 02314

16-чная система счисления:

* 0хА2, 0ХС021, …

Постфиксы:

* 0x15CLU, 0xF48Du, 0x42ll

Символьные литералы:

* Записываются в апострофах
* Может использоваться префикс L
* Задают число, равное коду символа

‘a’, ‘b’, ‘#’

‘\0’, ‘\n’, ‘\r’

‘\xCD’

‘\123’

L’Q’, L’ы’

Типы целочисленных литералов:

* Для обычных литералов:
  + В зависимости от значения и постфиксов (если есть)
  + Но не менее int
* Для символьных литералов:
  + С префиксом L – wchar\_t
  + Без префикса – int

*Вещественные типы*

Точные размеры и диапазоны для вещественных типов стандартом не определены

* Заданы только минимальные

Минимальное основание системы счисления для вещественных типов = 2

|  |  |  |
| --- | --- | --- |
| Название типа | размер, бит | Название по IEEE 754 |
| Float | 32 | Single |
| Double | 64 | Double |
| Long double | 80 | Double-Extended |

Компиляторы не обязаны использовать IEEE-совместимые форматы

В компиляторах Microsoft long double эквивалентен double

Примеры:

1.42, -0.2

5.2e-8, -3.5E+6

268E16, 14e-6

0xDB.84A1p2, 0x4F5.45P4

Типы вещественных литералов:

* Без постфиксов – double
* Постфикс F – float
* Постфикс L – long double

**Структура программы и основы синтаксиса**

Программа представляет собой последовательность объявлений переменных и функций

* Кроме того, могут встречаться специальные директивы

int a = 8, b

signed long int x;

unsigned char mychar

\_\_int16 somevar

*Функция main*

Функция с именем main является точкой входа в программу

* Т.е. выполнение начинается с нее

В зависимости от компилятора и типа проекта она может называться иначе

Параметры:

* Количество аргументов, переданных в командной строке при запуске
* Сами аргументы

Возвращаемое значение – код результата

* 0 – успешно
* Другое значение – признак ошибки

Int main() | void main() | int main(int argc) …

**Модификаторы const и volatile**

Const

* Переменная не должна изменяться

Volatile

* Обращения к переменной не должны оптимизироваться

Const int maxCount = 32767

Const int step – 70

Const double pi = 3.1415

Volatile unsigned int timer

Volatile unsigned int event

**Функция консольного ввода-вывода**

*Функция printf*

Printf(“формат”, парам1, парам2, …);

printf(“Hello world!”);

printf(“Ответ: %d, answer);

printf(“Символ: %c”, symbol);

printf(“получено %f МБ (%f%%)”, done, done/total\*100);

|  |  |
| --- | --- |
| Символ | Способ вывода |
| %d, %i | Целое 10-чное число со знаком |
| %o | Целое 8-чное число без знака |
| %u | Целое 10-чное число без знака |
| %x | Целое 16-чное число без знака (символы: 0123456789abcdef) |
| %X | Целое 16-чное число без знака (символы: 0123456789ABCDEF) |
| %f | Вещественное со знаком вида  [-]dddd.dddd |
| %c | Одиночный символ (параметр задает код символа) |
| %s | Строка (параметр задает адрес строки) |
| %% | Символ процента |
| %p | Указатель (формат вывода зависит от модели памяти) |

Разобрать какие есть еще в своем компиляторе

*Функция puts*

Puts(“строковые данные”);

Puts(“Выполнено 0%);

Puts(mystr);

*Функция putc*

Putc(‘<символ>’);

Putc(‘a’);

*Функция scanf*

Scanf(“формат”, &парам1, &парам2, …);

Scanf(“%d”, x); //Ошибка!

Scanf(“%d”, &y);

Scanf(“%c”, &symbol);

*Функции gets и getchar*

Gets(str);

Char symbol = getchar();

Printf(“%c”, symbol);

Symbol = getchar()

Putchar(symbol)

**Операции и выражения**

Выражение – это последовательность:

* Операндов
* Знаков операций
* Символов - разделителей

Операции:

* Один или несколько операндов
* Результат вычисления
* Побочные эффекты

|  |  |  |
| --- | --- | --- |
| Символ | Операция | Аналог в Pascal |
| + | Сложение сохранение знака | + |
| - | Вычитание изменение знака | - |
| \* | Умножение | \* |
| / | Деление | div  / |
| % | Взятие по модулю (остаток) | mod |

int x = 10, y = 3

printf(“%d”, x / y); // 3

printf(“%d”, x % y); // 1

float x = 10, y = 3

printf(“%f”, x / y) ; // 3.333333

*Неявное преобразование типов*

Производится, если в выражении участвуют операнды разных типов

В общем случае операнды приводятся к «более старшему» типу

char 🡪 short 🡪 int 🡪 long 🡪 long long 🡪 float 🡪 double 🡪 long double

По стандарту:

* Для вещественных типов выбирается «более старший»
* Целочисленным типам назначается ранг (integer conversion rank):
  + Если оба типа знаковые/беззнаковые, выбирается «более старший»
  + Иначе возможны преобразования «знаковый ⬄ беззнаковый»

*Операция преобразования типов*

(тип) выражение

int x = 5;

float f;

f = x / 2

printf(“%f”, f); // 2

f = (float)x / 2

printf(“%f”, f); // 2.5

*Операция присваивания*

Имеет вид:

* <lvalue> = <выражение>

Поведение:

* Результат вычисления – значения правого операнда
* Побочный эффект – левому операнду присваивается значение

x = y = z = 0

*Инкремент и декремент*

Обозначения:

* ++ инкремент (увеличить на 1)
* -- декремент (уменьшить на 1)

Две формы:

* ++x префиксная
* x++ постфиксная

В префиксной форме (++х, --х):

* Увеличить/уменьшить
* Значение выражения равно новому значению переменной

В постфиксной форме (х++, х--):

* Значение выражения равно старому значению переменной
* Увеличить/уменьшить

x = 10 x = 10

y = ++x y = x++

// x = 11, y = 11 // x = 11, y = 10

|  |  |  |
| --- | --- | --- |
| Символ | Операция | Аналог в Pascal |
| & | Побитовое И | and |
| | | Побитовое ИЛИ | or |
| ^ | Побитовое исключающее ИЛИ | xor |
| ~ | Побитовая инверсия | not |
| >> | Сдвиг вправо | shr |
| << | Сдвиг влево | shl |

X = x + 10 x += 10

\*=, /=, <<=, >>=

Int x = 42, y = 98;

x ^= y ^= x ^= y (меняет местами значения x и y, часто где можно увидеть)

КОД СОДЕРЖИТ ГРУБУЮ ОШИБКУ

(но скорее всего будет работать)

*Операция Sizeof*

Вычисляет размер переменной или типа в байтах

float f;

printf(“%d”, sizeof f);

printf(“%d”, sizeof (int));

printf(“%d”, sizeof (f));

*Работа с логическими значениями*

В отличие от Pascal в языке C нет полноценного логического типа.

При проверке на истинность:

* Ненулевое значение – True
* Нулевое значение – False

При вычислении результата:

* 1, если результат – True
* 0, если результат – False

== – равно (=)

!= – не равно (<>)

&& – логическое И (and)

|| – логическое ИЛИ (or)

! – логическое отрицание (not)

*Приоритеты операций*

1. Постфиксные операции:

* Вызов функции
* Обращение к элементу массива
* Обращение к полю структуры (записи)
* Постфиксные инкремент/декремент

1. Унарные операции:

!х, ~x, +х, -х, префиксные инкремент/декремент,

Взятие адреса, обращение по указателю, определение размера sizeof x

1. Явное приведение типа (type)x
2. Мультипликативные операции:
   1. Умножение
   2. Деление
   3. Взятие остатка
3. Аддитивные операции
   1. Сложение
   2. Вычитание
4. Операции побитового сдвига
5. Операции сравнения (по убыванию):
   1. > < >= <=
   2. == !=
6. Побитовые операции (по убыванию):
   1. Побитовое И
   2. Побитовое исключающее ИЛИ
   3. Побитовое ИЛИ
7. Логические операции (по убыванию)
   1. Логическое И
   2. Логическое ИЛИ
8. Операция условия ?:
9. Операции присваивания
   1. В том числе составные
10. Операция запятая ,

**Операторы управления вычислительным процессом**

***Операторы ветвления***

*Оператор if*

if (выражение)

оператор\_1;

else

оператор\_2;

if (выражение)

{

Оператор\_1;

Оператор\_2;

}

else

оператор\_3

Pascal/Delphi:

* Точка с запятой – разделитель   
  begin оператор\_1; оператор\_2 end

C:

* Точка с запятой – признак конца оператора
* Закрывающая фигурная скобка } – признак конца составного оператора  
  {оператор\_1; оператор\_2;}

Побитовые операции:

* Вычисляются всегда
* Компилятор может изменять порядок вычисления операндов

Логические операции:

* Порядок вычисления операндов строго «слева направо»
* Сокращенное вычисление

Эквивалент оператора if для использования в выражениях

Синтаксис:

* операнд\_1 ? операнд\_2 : операнд\_3

Принцип работы:

* Вычислить операнд\_1
* Если операнд\_1 – истина, результат – операнд\_2
* Иначе результат – операнд\_3

*Оператор goto*

Позволяет выполнять безусловный переход к метке

Объявлять метку не требуется

int x = 5;

goto MyLabel;

x++;

MyLabel:

printf(“%d”, x);

Переход возможен только в пределах функции

*Оператор switch*

Switch (целое\_выражение)

{

case константа\_1: операторы\_1; break;

case константа\_2: операторы\_2; break;

…

case константа\_n: операторы\_n; break;

default: операторы\_n+1;

}

Только для целочисленных типов.

Значения в case – константы

Может быть указана ветвь Default, переход к которой выполняется, если нет подходящего case

Если не указан break, выполнение переходит в следующую ветвь

Switch(x)

{

Case 1:

Case 2:

Printf(“Good”);

Break;

Default;

PrintF(“Bad”);

}

*Операция запятая*

Символ «запятая» в языке С может быть:

* Разделителем
* Операцией

Операция запятая:

* Два операнда
* Результат – значение 2-го операнда

int x;

int y = (x = 3, 3 \* x);

// x = 3, y = 9

**Операторы циклов**

*Оператор for*

В заголовке – три предложения:

* Первое – инициализация цикла
* Второе – условие входа в цикл
* Третье – вычисления в конце итерации

Предложения могут быть:

* Выражениями – все три
* Объявлением переменных – первое

for (int i = 0; i < 5; i++)

printf(“%d”, i);

*Оператор while*

while (выражение)

оператор;

int k = 2;  
while (k <= 100)  
 k = k \* 2  
printf(“%d/n”, k);

*Оператор do…while*

Do  
 оператор;  
while (выражение);

Do  
{  
 Оператор;  
}  
while (выражение);

В отличие от Pascal/Delphi задается условие продолжения, а не выхода из цикла

*Оператор break*

Может использоваться для досрочного выхода из цикла (аналогично Pascal/Delphi).

Применяется в составе оператора switch для пропуска последующих ветвей

*Оператор continue*

for (int i = -5; i <= 5; i++)  
{  
 if (0 == i)  
 continue;  
 printf(“%f”, 25.0 / i);  
}

**Массивы и указатели**

*Массивы в языке С*

float arr[10];

int users[100], items[50];

Индексация элементов всегда начинается с нуля

Имя массива – константный указатель на его первый элемент (с индексом 0)\*

(\* в плане не совсем правда, но пока что пойдет)

int x[5][8];

int arr[5] = {3, 5, 8, 13, 21};\

int buf[256] = {}; (256 элементов ‘0’)

int x[] = {1, 3, 5};

int y[4][3] = {

{1, 3, 5},

{2, 4, 6},

{3, 5, 7}

};

==

1 3 5  
2 4 6  
3 5 7  
0 0 0

или

int y[4][3] = {

1, 3, 5, 2, 4, 6, 3, 5, 7

};

*Доступ к данным в памяти*

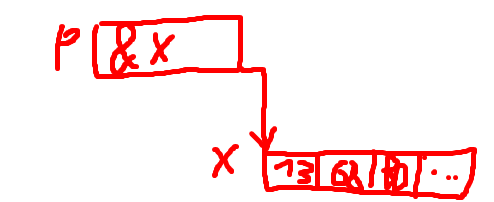
Доступ к данным в памяти:

* Прямой (по имени переменной)
* Косвенный (по указателю)

Указатель – переменная, значением которой является адрес другой переменной.

int x;

int \*p = &x;



int \*p ⬄ int\* p

НО

int\* p1, p2;  
p1 имеет тип «указатель на int»  
p2 имеет тип «int»

void \*p; 😉 (просто pointer, не типизированный)

*Нулевой указатель*

Обозначается NULL

Совместим с указателями любого типа

Фактическое значение зависит от платформы

(void \*)0 == NULL

0 == NULL  
Но:

int x = 0;  
if (NULL == (void \*)x)  
 printf(“Не всегда!”);

*Операции над указателями*

Взятие адреса

…

*Взятие адреса*

var

int \*p; p: ^Integer;

int x; x: Integer;

p = &x; p := @x;

&(a + 1)  
Выражение не имеет адреса

&5  
Константа не имеет адреса

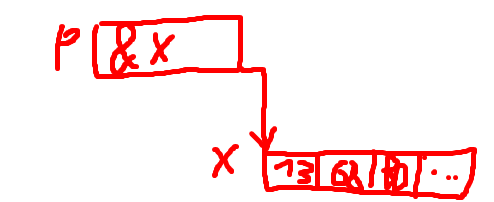
// int c[40];  
&c  
Имя массива – указатель на его первый элемент.  
Может компилироваться (в этом случае игнорируется)  
(на самом деле это не так, но пока схаваем такое, сойдет)

int x = 10;

int \*p;

p = &x;

\*p = 5;

 (ток не 13 68… а 05 00 00 00)

*Увеличение/уменьшение указателя на целое число*

int \*p;  
p = &a[0];  
p = p + 5;

// Значение p увеличилось на  
// 5 \* sizeof(int) = 10  
// т.е. p указывает на a[5]

*Разность указателей*

int \*p1, \*p2;  
int x;

p1 = &a[0];  
p2 = &a[5];  
x = p2 – p1;

// x будет присвоено значение 5

Сравнение указателей

p1 = &a[0];  
p2 = &a[3];

if (p1 > p2)

printf(“Случилось невозможное!”);

*Присваивание указателей*

p1 = p;

Указатели должны иметь один и тот же тип

Для присваивания указателей разных типов использовать приведение типов

Исключение: тип void\*

*void и void \**

Ключевое слово void обозначает отсутствие информации о типе.

void \*p;

Указателю void \* можно присваивать указатели любых других типов.  
Аналог void \* в Pascal/Delphi – тип Pointer.

С указателем void \* нельзя применять операцию разыменования указателя.

Unsigned short a = 0x1234;  
Void \*p = &a;  
Char b;  
Unsigned short c;

B = \*(char \*)p;  
C = \*(unsigned short \*)p;

Printf(“%02X\n”, b);  
Printf(“%04X\n”, c);

**Связь массивов и указателей**

int a[5];  
int \*p;

// Присваивание указателю  
// адреса первого элемента

p = a;  
p = &a[0];

Доступ к элементам массива:

* a[3]
* \*(p + 3)

// Имя массива – указатель  
int x1 = \*(a + 3);  
int x2 = a[3];

// Указатель можно использовать как имя массива  
int x3 = \*(p + 3);  
int x4 = p[3]

НО

p += 3; // Работает  
a += 3; // Ошибка

int a[3][3];

a[0] 🡪 a[0][0] a[0][1] a[0][2]  
a[1] 🡪 a[1][0] a[1][1] a[1][2]  
a[2] 🡪 a[2][0] a[2][1] a[2][2]

a[i] + j == &a[i][j]  
\*(a[i] + j) == a[i][j]  
\*(\*(a + i) + j) == a[i][j]

**First-class citizens**

Объект первого класса – сущность, все элементы которой:

* Могут быть переданы как параметр процедуры/функции
* Могут быть возвращены функцией
* Могут участвовать в присваивании
* Могут проверяться на равенство

Массивы в языке C не являются объектами первого класса.

* При передаче в процедуры/функции информацию о размере приходится передавать самостоятельно
* Не могут быть возвращены функцией
* Не могут присваиваться
* Не могут проверяться на равенство

*Динамическое распределение памяти*

Функции динамического распределения памяти доступны при подключении stdlib.h

// Прототипы функций

void \*malloc(size\_t size); // Выделить память  
void \*realloc(void \*ptr, size\_t size); // Изменение размера ранее выделенного блока (возвращаемое знач. – адрес области)  
void \*calloc(size\_t nmemb, size\_t size); // Выделение памяти, заполняет нулями, умножение двух чисел, nmemb на size  
void free(void \*ptr); // Освобождение памяти

Все функции выделения памяти в случае ошибки возвращают NULL

realloc:

* Если параметр-указатель равен NULL, работает аналогично malloc
* Если новый размер меньше старого, часть данных теряется
* Если новый размер больше старого новые байты заполнятся

**Функции**

Объявление функции состоит их:

* Прототипа функции
* Тела функции

Прототип – «заголовок» функции

Прототип включает:

* Имя функции
* Тип возвращаемого значения
* Список и типы параметров

int myFunction() – Параметров нет, возвращаемое значение: int

void myFunction() – ( крч как процедура, нет возвращаемого значения (void) )

int myFunc (int x, float y)  
{  
…  
}

int getGCD(int a, int b)  
{  
 while (0 != a && 0 != b)  
 {  
 if (a > b)  
 a = a % b;  
 else  
 b = b % a;  
 }  
 return a+b;  
}

Область видимости переменной – блок, в котором она объявлена.

* Блок – последовательность операторов между {}
* Если объявление вне блоков, область видимости – файл.

Область видимости метки – функция.

*Передача параметров*

В языке C параметры всегда передаются **по значению**

* Все остальные способы передачи организуют, используя те или иные возможности языка

В прототипе можно не указывать имена параметров:

int myFunc(int a, float b);

int myFunc(int, float);

При описании самой функции:

* Имена параметров должны быть указаны
* Их типы должны соответствовать указанным в прототипе

Если функция без параметров, можно указывать пустые скобки:

int myFunc();

В новых версиях стандарта это называется old-style и считается устаревшим

Ключевое слово void вместо списка параметров означает, что функция – без параметров

int someFunc(void);

Объявление переменной и опережающее описание функции имеют оидн и тот же синтаксис

int f(void);

int \*fip(float x);

int f(void), \*fip(float x);

Указатели на функции

Объявление переменной-указателя на функцию:

// Указатель на функцию  
// long someFunc(int);  
long (\*pFunc)(int);

**void printNumber(int n = 42)  
{  
 Printf(“%d\n”, n);  
}**

**printNumber(15); // 15  
printNUmber(); // 42**

**ЭТО C++ А НЕ C!!!1!11!!**

**Строки**

В отличии от Pascal в языке C нет полноценного строкового типа.

Вместо строкового типа используются массивы символов…

… А полноценных массивов в C тоже нет!

*Способы представления строк*

С упреждающей длиной  
С терминальным элементом (С’ишная тема) (на конце обычно “\0”)

*Unicode*

Для хранения символа может отводиться больше 1 байта

*Поддержка Unicode в C/C++*

Вводятся дополнительные типы:

* char
* wchar (wchar\_t)
* TCHAR

Для Unicode-строк написаны отдельные реализации функций.

* Для TCHAR тоже

char \*str = “Hello, world!”;  
char str[] = “Hello, world!”;  
char str[] = {‘H’, ‘e’, ‘l’, …};  
char str[] = {72, 101, 108, …};

char \*str = “Hello, world!”; (ссылка на)  
char str[] = “Hello, world!”; (сам массив)

char \*s = “Hello!”;

char \*const s = “Hello!”;

const char\*s = “Hello!”;

const char \*const s = “Hello!”;

**Атака переполнением буфера**

*Функции для работы со строками*

Первоначальные реализации стандартной библиотеки при неправильном использовании делали программы уязвимыми к атакам переполнением буфера.

* А правильное было возможно далеко не всегда!

EICAR test virus

ASLR

**Строковые функции**

В современных реализациях стандартной библиотеки C:

* «Классические» функции:
  + Принимают просто указатель на строку
* Safe-функции:
  + Имеют дополнительные параметры, задающие размеры строк
  + Имеют постфикс \_s в имени

*Функция strcpy()* копирует в строку из строки

char \*strcpy(   
 char \*dest,  
 const char \*src  
);

// Пример реализации

char \*user\_strcpy(char \*dest, const char \*src)  
{  
 char \*result = dest;  
 do  
 {  
 \*dest++ = \*src;  
 }  
 while (\*src++);  
 return result;  
}

*Функция strlen()*

size\_t strlen(  
 const char \*src  
);

*Конкатенация строк*

char \*str1 = “Студент”;  
char \*str2 = “ БГУИР”;  
char \*str3;

str3 = str1 + str2; ФУУУ, нельзя так, это же 2 указателя, 2pizza

*Функция strcat()*

Char \*strcat(  
 char \*dest,  
 const char \*src  
);

*Функция strchr()* (ищет)

char \*strchr(  
 char \*s,  
 int c  
);

*Функция strdup()* (создает копии строки)

char \*strdup(  
 const char \*s  
);

// Пример реализации  
char \*user\_strdup(const char \*s)  
{  
char \*result = malloc(strlen(s) + 1);  
return result ? strcpy(result, s) : 0;  
}

*Функция strcmp ()* (сравнение строк)

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

*Функция sprintf()* (вывод по формату)

int sprintf(  
char \*s,  
const char \*format,  
…  
);

*Функция sscanf()* (вывод по формату)

int sscanf(  
const char \*s,  
const char \*format,  
…  
);

ANSI UNICODE TCHAR

strcopy wcscpy \_tcscpy

\_strdup \_wcsdup \_tcsdup

sprintf swprintf \_stprintf

sscanf swscanf \_stscanf

char \*strcpy(…)  
char \*strcpy\_s(…, size\_t nElem, …);

**Классы хранения и видимость переменных**

Классы хранения

* Автоматический (auto)
* Регистровый (register)
* Статический (static)
* Внешний (extern)

Блок – последовательность операторов, заключенная в фигурные скобки

Переменная автоматического класса хранения:

* Создается при входе в блок
* Уничтожается при выходе из блока
* До инициализации имеет неопределенное значение

Используется по умолчанию для локальных переменных

int f1(void)  
{  
auto int x = 10; // Но и без auto будет auto, cmon это ж авто  
}

В C++11 ключевое слово auto изменило семантику.

* Используется для указания, что тип переменной определяется по инициализатору

*Регистровый класс хранения*

Время жизни и область видимости аналогичны auto-переменным

По возможности переменной ставится в соответствие регистр общего назначения

void main()  
{  
register long sum = 0;  
for (register int i = 1; i <= 100; i++)  
 sum = sum + i;

printf(“\nsum[100] = %d”, sum);  
}

*Статический класс хранения*

Переменные статического класса хранения размещаются в статической памяти.

* Т.е. в секции/сегменте данных

Могут быть:

* Локальные;

Глобальные.

Область видимости глобальной переменной с классом хранения static – файл.

* Но есть нюансы

static – класс хранения по умолчанию для глобальных переменных.

Переменные с классом хранения static по умолчанию инициализируются нулевыми значениями.

*Внешний класс хранения*

Используется, чтобы:

* Сделать доступной в блоке переменную, определенную глобально
* Использовать глобальные переменные, объявленные в разных файлах

…  
void f1()  
{  
extern double i;  
…  
}  
double i = 9.36;

// A.c // B.c  
… …  
int x = 42; extern int x;  
void f1() void f2()  
{…} {…  
… x += 5;  
 …  
 }

Ограничение:

* Нельзя задавать начальное значение (инициализировать)

*Классы хранения*

Только для локальных переменных:

* auto
* register

Для локальных и глобальных переменных:

* static
* extern

|  |  |  |  |
| --- | --- | --- | --- |
| **Класс хранения** | **Место хранения** | **Время жизни** | **Область видимости** |
| auto | стек | блок | блок |
| register | регистры | блок | блок |
| static (локальная) | сегмент/секция данных | программа | блок |
| static (глобальная) | сегмент/секция данных | программа | файл |
| extern | сегмент/секция данных | программа | блок/программа |

**Структуры, объединения и перечисления**

*Структура*

Структура – совокупность переменных (возможно, различных типов), сгруппированных под одним именем

struct TItem  
{  
 int n;  
 char name[20];  
 float cost;  
 int count;  
 char note[100];  
};  
  
struct TItem curr;

Объявление структурного типа имеет ту же область видимости, что и объявление переменных

* Т. е. блок (если объявлен в блоке)  
  или файл (если объявлен вне блока)

// Объявление структурных переменных (по-другому)

struct item  
{  
 int n;  
 char name[20];  
 float cost;  
 int count;  
 char note[100];  
} m1, m2, m3;

// Тут item уже объявлен  
struct item m1 = {1, “генератор”, 100.5, 100, “частота 460 Гц”};

struct item  
{  
 int n;  
 char name[20];  
 float cost;  
 int count;  
 char note[100];  
} m1 = {1, “генератор”, 1.2, 100, “сдан в ремонт”};

m1.n = 1;  
m1.name = “генератор”; // НЕЛЬЗЯ, name – массив  
strcpy(m1.name, “генератор”);

struct item \*p1 = &m1;

(\*p1).cost = 1.5;

p1->cost = 1.5;

*Присваивание структур (в отличии от массивов – можно)*

struct item m1, m2, m3;  
m1 = m2;  
m1 = m2 = m3;

struct item \*p;

p = (struct item \*)malloc(sizeof(struct item));  
…  
free(p);

++p->cost ++(p->cost) (++p)->cost

Передача функциям структурных переменных

mult(m1);  
…  
float mult(struct item m1)  
{  
 return (m1.cost \* m1.count);  
}

mult(&m1);  
…  
float mult(struct item \*pItem)  
{  
 return (pItem->cost \* pItem->count);  
}

*~~Оператор~~ Класс хранения typedef*

typedef тип новое\_имя\_типа;

typedef int integer;  
typedef int \*PINTEGER;

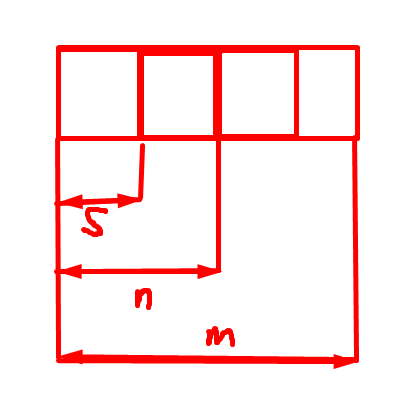
typedef относится к классам хранения! ОН НЕ ОПЕРАТОР O\_o

typedef struct \_item (можно и без \_item, т.е.: typedef struct)  
{  
 int n;  
 char name[20];  
 float cost;  
 int count;  
 char note[100];  
} item;

item m1, m2, m3;

**Объединение**

union cell  
{  
 short n;  
 float m;  
 char s;  
} var;



var.m = 1;

void f1(union cell \*p)  
{  
…  
p->m = 2;  
…  
}

**Битовые поля**

struct   
{  
 unsigned int y:1;  
 unsigned m:1;

int k:2;  
} stat;

stat.y = 0;

Битовые поля по возможности упаковываются в одну и ту же единицу хранения

Есть два способа заставить компилятор упаковывать битовые поля с новой единицы хранения

* Думать и читать стандарт 😊

**Перечисления**

enum Month {Jan, Feb, Mar,} // от 0 и погнали по +1

enum Month m1;  
m1 = Jan;

**Файлы**

Файл – именованная область на каком-либо носителе, используемая для хранения данных

*Режимы доступа к файлу*

Текстовый [31][38][35][30]

Бинарный [3A][07]

*Структура FILE*

Зависит от реализации

Полагаться на формат структуры **FILE** ***НЕЛЬЗЯ***

FILE \*fopen(  
 const char \*filename,  
 cont char \*mode  
);

FILE \*p;  
p = fopen(“1.doc”, “r+b”);  
…

*Режим открытия файла*

|  |  |  |  |
| --- | --- | --- | --- |
| Режим | Доступ | Создает? | Заменяет? |
| r | R | - | - |
| r+ | R/W | - | - |
| w | W | + | + |
| w+ | R/W | + | + |
| a | A | + | - |
| a+ | R/A | + | - |

|  |  |
| --- | --- |
| Режим | Описание |
| t | Текстовый |
| b | Бинарный |

*Функция fclose()*

int fclose( // int – код ошибки  
 FILE \*stream  
);

*Стандартные потоки*

stdin   
stdout  
stderr

*Функция fputs()*

int fputs(  
 const char \*str,  
 FILE \*stream  
);

Успех – неотрицательное значение  
Ошибка – EOF

*Функция fputc()*

int fputc(  
 int c,  
 FILE \*stream  
);

Успех – код символа  
Ошибка – EOF

*Функция fgets()*

int fgets(  
 char \*str,  
 int n,  
 FILE \*stream  
);

Успех – str  
Ошибка – NULL  
Конец файла – NULL

*Функция fgetc()*

int fgetc(  
 FILE \*stream  
);

Успех – прочитанный символ  
Ошибка – EOF  
Конец файла – EOF

*Функция fprintf()*

int fprintf(  
 FILE \*stream  
 const char \*format,  
 …  
);

Возвращает количество записанных символов

*Функция fscanf()*

int fscanf(  
 FILE \*stream  
 const char \*format,  
 …  
);

Возвращает количество прочитанных переменных (EOF – ошибка или конец файла)

*Доступ к файлу*

Последовательный

Произвольный

*Функция fseek()*

int fseek( // Переместиться   
 FILE \*stream,  
 long offset,  
 int origin  
);

Параметр origin:

* SEEK\_SET – начало файла
* SEEK\_CUR – текущее положение
* SEEK\_END – конец файла

Возвращаемое значение:

* 0 – успешно
* Не 0 – ошибка

*Функция ftell()*

long ftell( // В какой позиции файла я нахожусь?  
 FILE \*stream  
);

*Функция feof()*

int feof( // указывает конец ли  
 FILE \*stream  
);

*Функция ferror()*

int ferror( // вернет код последней ошибки  
 FILE \*stream  
);

*Признак конца файла*

Не хранится, возвращается стандартной библиотекой как значение EOF

char c;  
while (!feof(stdin))  
 c = fgetc(stdin);

Будет выполняться до нажатия Ctrl+Z (IBM PC), Ctrl+D (Mac) …

*Функция fwrite()*

size\_t fwrite(   
 const void \*buffer,  
 size\_t size,  
 size\_t count,  
 FILE \*stream  
);

Возвращает количество записанных элементов

*Функция fread()*

size\_t fread(   
 const void \*buffer,  
 size\_t size,  
 size\_t count,  
 FILE \*stream  
);

Возвращает количество прочитанных элементов

*Функция fflush()*

int fflush( // Смывает поток, очищает буфер  
 FILE \*stream  
);

0 – успешно  
EOF – ошибка

*Буферизация при работе с файлами*

Программа на C/C++ 🡨🡪 Стандартная библиотека C/C++ 🡨🡪 Драйвер файловой системы 🡨🡪 Диск

Оперативка

[ТУТ КОНЕЦ]

Умные мысли

Побочный эффект операции – любые действия, выполняемые операцией не связанные с вычислением результатов