Класс памяти

Любая переменная (объект) имеет *область видимости* и *время жизни*.
Область видимости - это та часть в программе, в которой переменная доступна. *Переменная считается доступной (видимой)* в блоке или в исходном модуле, если в этом блоке или модуле известны имя и тип переменной.
Время жизни - в течении какого времени переменная хранит свое значение.
Переменные в языке С и С++ могут быть ***локальными*** или ***глобальными***.
Переменная, определенная вне любого блока и вне любой функции является ***глобальной***.
Для глобальной переменной область видимости и время жизни в течение всей программы в том проекте, в котором она определена, кроме того блока или функции, где данная переменная переопределена.
Переменная определенная *внутри блока* или *функции* является ***локальной***.
Для ***локальной переменной*** *область видимости* и *время жизни* в той функции или в том блоке, где данная переменная определена.
Спецификатор класса памяти при определении переменной может быть **auto, register, static** или **extern**.
Если **класс памяти** *не указан явно*, то компилятор определяет его исходя из контекста определения переменной.
Переменные классов **auto** и **register** имеют локальное время жизни.
Спецификаторы **static** и **extern** определяют переменные с глобальным временем жизни.
При определении **локальной переменной** может быть использован **любой из четырех спецификаторов класса памяти**, а если он не указан, то подразумевается класс памяти **auto**.
Автоматическая переменная (auto) – это всегда локальная переменная, но не наоборот.
1. К **переменным типа auto доступ из внешнего блока напрямую невозможен.**
2. Инициализируются при определении или путем присваивания. Если инициализация отсутствует, то начальное значение не определено.
3. Память для переменных типа auto выделяется в стеке автоматически при входе в блок и освобождается тоже автоматически при выходе из блока. Причем, при повторном входе в блок этой переменной может быть выделен другой участок памяти.
Регистровая переменная (register) – могут быть **локальные переменные типа int.**
Ключевое слово register – предписывает компилятору распределить память для объекта в регистре, если это представляется возможным.
Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной**.**
Класс памяти static может применяться к *переменным* (локальным, глобальным) *и функциям*.
Для **статических переменных** память **выделяется в оперативной памяти**, и поэтому их значение сохраняется при выходе из блока.
Статическую переменную можно ***проинициализировать только константным выражением*.**
Инициализация для них выполняется **один раз перед началом программы**.
Если явная инициализация отсутствует, то переменная **инициализируется нулевым значением**.
Если ***глобальной переменной или функции*** задать класс **памяти static,** то это означает, что их ***область видимости только внутри исходного модуля*** в котором они определены**, а *время жизни в течении всей программы*.**
Если *локальной переменной* задается класс памяти **static,** то это означает, что **область видимости** данной переменной только внутри функции в которой она определена, а **время жизни** в течении всей программы.
При каждом последующем вызове функции переменная типа **static** сохраняет последнее значение которое было в данной переменной при предыдущем вызове.
Класс памяти extern автоматически применяется к ***именам функций*** и ***глобальным (переменным) объектам***.
Объект или функция имеющие класс памяти **extern**, имеет область видимости от точки определения до конца исходного модуля.
Можно сделать их видимыми и в других исходных модулях, для чего в этих модулях следует их объявить.

Определение и объявление функции

Функция – это логически самостоятельная именованная часть программы, которой могут передаваться параметры и которая может возвращать некоторое значение.Функция, во-первых, является одним из производных типов С и С++, а, во-вторых, минимальным исполняемым модулем программы.

Функция имеет следующий вид:

ТипФункции **Имя**Функции([СписокПараметров])
{ тело функции }

Тип Функции – это тип результата, возвращаемого из функции может быть любой тип данных, кроме массива или функции, но может быть указатель на массив или указатель на функцию.

Имя функции – это любой правильно написанный идентификатор.

Список параметров – это список разделенных запятыми объявлений тех параметров, которая получает функция при вызове. Для каждого параметра, передаваемого в функцию, указывается его тип и имя.

Тело функции – это блок или составной оператор. В теле функции может быть оператор **return**, который возвращает полученное значение функции в точку вызова. Он имеет следующую форму:

return [выражение];

Любая *функция* должна быть **объявлена** (прототип функции), **вызвана** и **определена** (описание функции).

Передача параметров в функцию

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм передачи параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

При передаче по значению выполняются следующие действия:

1. вычисляются значения выражений, стоящие на месте фактических параметров;
2. в стеке выделяется память под формальные параметры функции;
3. каждому формальному параметру присваивается значение фактического параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

Пример

```
#include<stdio.h>
void kvadrat(int);//функция с параметром
void main(){
int x;// локальная переменная, для функции main(), //поэтому она не доступна в функции kvadrat()
      printf("x=");
      scanf("%d",&x);
      kvadrat(x);
}

void kvadrat(int a){
      printf("%d^2=%d\n", a, a*a);
}
```

Функция напрямую, т.е. с помощью оператора перехода **return**, может вернуть только одно значение. Вернуть несколько значений из функции невозможно, но можно возвращать не значения, а записать эти значение сразу в ячейки локальной памяти. Для этого необходимо передавать параметры в функцию не по значению, а **по адресу**. Передать **адрес переменной** можно с помощью **указателя** или по средствам **ссылки**.

Указатели и ссылки

Понятие указателя

Указатель – это переменная, в которой хранится адрес области памяти. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом. Указатели делятся на две категории: *указатели на переменные* и *указатели на функции*.

Определение указателей на переменные имеет вид:

ТипДанных *имя;

ТипДанных – это тип, на который будет указывать указатель.

***** - определяет тип указателя;

имя – это переменная, которой хранится адрес переменной (ячейки), на которую будет указывать указатель (всегда целого типа);

***имя** – это содержимое адреса, на который указывает указатель (т.е. переменная, которая соответствует типу на который указывает указатель).

char* pc;
int* pi;
double* pd;

Некоторые способы определения указателей:

int *i; //i хранит адрес ячейки с целым числом
int *x, y; // x хранит адрес, а y – целое число!
double *f, *ff; // два указателя на ячейки double
char *c; // указатель на ячейку, хранящую символ
int* pi; // указатель на целую переменную.
const int* pci; // указатель на целую константу.
int* const cpi;// указатель-константа на переменную целого типа.
const int* const cpc; // указатель-константа на целую константу.
int a;** //который отличается от простого указателя тем, что хранит адрес ячейки, которая сама хранит адрес некоего объекта в памяти.

*Основные операции, связанные с применением указателей***:**

1. инициализация;
2. присваивание;
3. чтения и записи по адресу;
4. арифметические операции;
5. операция сравнения.

Инициализации указателя

1. С помощью операции получения адреса

```
int a = 5;
int* p = &a;
```

Символ '&', стоящий перед именем переменной, означает операцию получения адреса этой переменной.

2. С помощью проинициализированного указателя

```
int a = 5;
int* p = &a;
int* r = p;
```

3. Присваивание пустого значения:

```
int* N=NULL;
int* R=0;
```

Операция присваивания

Переменные различных типов можно присвоить друг другу.

Операция чтения и записи по адресу

Операция ‘*’ -это унарная операция снятия ссылки.

Если операция * **имя** находится в

- левой части операции присваивания, то выполняется запись по данному адресу;
- правой части операции присваивания, то выполняется операция чтения по адресу.

```
double x=67,*px,  y;
        px=&x;
        printf("\n%.2lf\n",*px);
        *px=*px+3;
        printf("%.2lf\n",x);
```

Арифметические операции

Арифметические операции применимы только к указателям одного типа**:**

1. Инкремент *увеличивает значение указателя на величину размера типа.*

```
        char* pc;
        int* pi;
        double* pd;
        .....
        pc++;           //значение увеличится на 1
        pi++;           //значение увеличится на 4 или 2
        pd++;           //значение увеличится на 8
```

2. Декремент уменьшает значение указателя на величину размера типа.

3. Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

4. Суммирование двух указателей не допускается.

5. Можно суммировать указатель со значение целого типа.

Операции сравнения

Операции сравнения применимы только к указателям одного типа. Результат операции сравнения целое число. Если это число 1 – это **true**, в остальных случаях **false**.

Результат операции “==” будет **true** если указатели указывают на одну и туже область памяти.

Результат операции “>=” будет **true** если переменная на которую указывает первый указатель расположен в памяти под более старшим адресом, чем переменная на которую указывает второй указатель и т.д.

Пример передачи параметра в функцию по адресу

```
void fff(int, int, int*, int*, int*, double*);

void main(){
      int x = 5 , y = 2, sum, pr, raz;
      double del;
      fff( x, y, &sum, &pr, &raz, &del);
      printf("%d + %d = %d\n", x, y, sum);
      printf("%d - %d = %d\n", x, y, raz);
      printf("%d * %d = %d\n", x, y, pr);
      printf("%d / %d = %.2lf\n", x, y, del);
}
void fff(int a, int b, int*p1, int*p2, int*p3,
                                          double*p4){
      *p1=a+b;
      *p2=a*b;
      *p3=a-b;
      *p4 = (double)a/b;
}
```

Ссылка

Ссылка это и есть, в принципе, указатель, только скрытый и является еще одним именем для переменной. Ссылка, по сути, является просто вторым именем переменной (объекта).

Тип & имя_ссылки = имя_переменной;

int i = 10;
int &p = i;
const char & CR='\n';

Правила работы со ссылками:
1. Ссылка должна быть явно проинициализирована при ее описании, если она не является параметром функции, не описана как extern.
2. После инициализации ссылке не может быть присвоено другое значение.
3. Операция над ссылкой приводит к изменению величины, на которую она ссылается.
4. Не существует указателей на ссылки, массивов ссылок (т.к. под массив выделяется память при компиляции) и ссылок на ссылки.
5. Функции не могут быть перегружены, если описание их параметров отличается только наличием ссылки.

```
void fff(int, int, int&, int&, int&, double&);
void main(){
      int x = 5 , y = 2, sum, pr, raz;
      double del;
      fff( x, y, sum, pr, raz, del);
      printf("%d + %d=%d\n", x, y, sum);
      printf("%d -%d=%d\n", x, y, raz);
      printf("%d * %d=%d\n", x, y, pr);
      printf("%d / %d=%.2lf\n", x, y, del);
}
void fff(int a, int b, int&p1, int&p2, int&p3,
                                          double&p4){
      p1=a+b;
      p2=a*b;
      p3=a-b;
      p4=(double)a/b;
}
```

Рекурсивные функции

Рекурсивная функция – это функция, которая вызывает сама себя – это прямая рекурсия, либо вызывает себя косвенно с помощью другой функции.

При вызове рекурсивной функции:
1. в стеке выделяется память под формальные параметры функции;
2. управление передается первому исполняемому оператору функции;
3. при повторном вызове данный процесс повторяется;
4. при завершении вызова функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

Рекурсивные функции применяются:
1. для компактной реализации алгоритмов;
2. для работы со структурами данных.

Достоинством рекурсии является компактность записи.
Недостатками рекурсии – расход времени и памяти на повторные вызовы, а главная опасность переполнение стека.

Посчитать факториал неотрицательного целого числа.
n! = n*(n-1)*(n-2)*..*2*1, причем 1! = 1 и 0! = 1.

```
int fact(int n){
      if(n == 1 || n == 0) return 1;// базовая задача
      return n * fact(n-1); // шаг рекурсии
```

```
}
```

Данную задачу можно записать использую оператор “? :”

```
int fact (int n){
      return (n>1) ? n * fact(n-1) : 1;
}
```

Ряд Фибоначчи. x_n- ый элемент ряда Фибоначчи равняется сумме двух предыдущих элементов ряда, т.е. $x_n = x_{n-1} + x_{n-2}$.

```
int fibonachi(int n){
// базовая задача
      if( n ==1 || n == 0 ) return n;
// шаг рекурсии
      else  return fibonachi(n-1)+fibonachi(n-2);
}
```

Генерация случайных чисел

Для работы с **генерацией случайных чисел** необходимо подключить следующие библиотеки: **stdlib.h** и **time.h**

В библиотеке **stdlib.h** находятся прототипы функции:
int rand(void); возвращает числа в диапазоне от 0 до 32 767.

void srand(unsingned int); получает в качестве аргумента число в пределах от 0 до 65535, если **int** занимает 2 байта и от 0 до 4 294 967 295 если 4 байта. Функция **srand** устанавливает начальное псевдослучайное число.
Функцию **srand –** рекомендуется использовать следующим образом:

srand(time(NULL));

где в качестве параметра используется вызов функции **time**, которая находится в библиотеке **time.h** и возвращает текущую дату в виде целого числа.

Получение случайного числа из интервала [a , b]
int n;
n = a + **rand**() % (b – a + 1);