

Лекция 6

Классы памяти.

Динамическое распределение памяти

Классы памяти и область действия. Декларация на внутреннем и внешнем уровнях. Модификаторы *auto*, *extern*, *static* и *register*

Нам уже известно, что все объекты программы на C/C++ перед их использованием должны быть декларированы. Операционные объекты (в частности, переменные) при этом, кроме атрибута «*тип*», имеют необязательный атрибут «*класс памяти*», существенно влияющий на область и время их действия.

Класс памяти программного объекта определяет время ее существования (время жизни) и область видимости (действия) и может принимать одно из значений: *auto*, *extern*, *static* и *register*.

Класс памяти и область действия объектов по умолчанию зависят от места их размещения в коде программы.

Область действия объекта – это часть кода программы, в которой его можно использовать для доступа к связанному с ним участку памяти. В зависимости от области действия переменная может быть локальной (внутренней) или глобальной (внешней).

Имеется три основных участка программы, где можно декларировать переменные:

- внутри функции (блока);
- в заголовке функции при определении параметров;
- вне функции.

Эти переменные соответственно называются **локальными (внутренними)** переменными, **формальными параметрами** и **глобальными (внешними)** переменными.

Область действия локальных данных – от точки декларации до конца функции (блока), в которой произведена их декларация, включая все вложенные блоки.

Областью действия глобальных данных считается файл, в котором они определены, от точки описания до его окончания.

Если класс памяти у переменной не указан явным образом, он определяется компилятором исходя из контекста ее декларации.

Время жизни может быть постоянным – в течение выполнения программы, и временным – в течение выполнения функции (блока) программы.

Автоматические переменные

Переменные, декларированные внутри функций, являются внутренними и называются локальными переменными. Никакая другая функция не имеет прямого доступа к ним. Такие объекты существуют временно на этапе активности функции.

Каждая локальная переменная существует только в блоке кода, в котором она объявлена, и уничтожается при выходе из него. Эти переменные называются **автоматическими** и располагаются в стековой области памяти.

При необходимости такая переменная инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из функции (блока), в которой декларирована переменная, т.е. время ее жизни – с момента описания до конца блока.

По умолчанию локальные объекты, объявленные в коде функции, имеют атрибут класса памяти **auto**.

Принадлежность к этому классу можно также подчеркнуть явно, например:

```
void main(void) {  
    auto int max, lin;  
    ...  
}
```

так поступают, если хотят показать, что определение переменной не нужно искать вне функции.

Для глобальных переменных этот атрибут не используется.

Регистровая память (регистровые переменные)

Регистровая память – атрибут **register**. Объекты целого типа и символы рекомендуется размещать не в ОП, а в регистрах общего назначения (процессора), а при нехватке регистров – в стековой памяти (размер объекта не должен превышать разрядности регистра), для других типов компилятор может использовать другие способы размещения или просто проигнорировать данную рекомендацию.

Регистровая память позволяет увеличить быстродействие программы, но к размещаемым в ней объектам в языке Си (но не C++) не применима операция получения адреса «&».

Статические и внешние переменные

Объекты, размещаемые в статической памяти, декларируются с атрибутом **static** и могут иметь любой атрибут области действия. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Время жизни – постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной.

Глобальные объекты всегда являются статическими.

Атрибут **static**, использованный при описании глобального объекта, предписывает ограничение его области применения только в пределах остатка текущего файла, а значения локальных статических объектов сохраняются до повторного вызова функции, т.е. в языке C/C++ ключевое слово **static** имеет разный смысл для локальных и глобальных объектов.

Итак, переменная, описанная вне функции, является внешней (глобальной) переменной.

Так как внешние переменные доступны всюду, их можно использовать вместо списка аргументов для передачи значений между функциями.

Внешние переменные существуют постоянно. Они сохраняют свои значения и после того, как функции, присвоившие им эти значения, завершат свою работу.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление. Автоматические и регистровые переменные имеют неопределенные начальные значения («мусор»).

Внешняя переменная должна быть определена вне всех функций. При этом ей выделяется фактическое место в памяти. Такая переменная должна быть описана в той функции, которая собирается ее использовать. Это можно сделать либо явным описанием *extern*, либо по контексту.

Описание *extern* может быть опущено, если определение внешней переменной находится в том же файле, но до ее использования в некоторой конкретной функции.

Ключевое слово *extern* позволяет функции использовать внешнюю переменную, даже в том случае, если она определяется позже в этом или другом файле.

Важно различать описание внешней переменной и ее определение. Описание указывает свойство переменной, ее размер, тип и т. д.; определение же вызывает еще и отведение ей участка оперативной памяти. Например, если вне какой-либо функции появляются инструкции

```
int sp;  
double val[20];
```

то они определяют внешние переменные *sp* и *val*, вызывают отведение памяти для них и служат в качестве описания для остальной части этого исходного файла. В то же время строки:

```
extern int sp;  
extern double val [ ];
```

описывают в остальной части этого исходного файла переменную *sp* как *int*, а *val* как массив типа *double*, но не создают переменных и не отводят им места в памяти.

Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной. Другие файлы могут содержать описание *extern* для доступа к ней.

Любая инициализация внешней переменной проводится только в декларации. В декларации должны указываться размеры массивов, а в описании *extern* этого можно не делать.

Например, в основном файле проекта:

```
int sp = 50;  
double val [20];
```

```
void main() {
    ...
```

а в дополнительном файле этого проекта:

```
extern int sp;
extern double val [ ];
    ...
```

В C/C++ есть возможность с помощью директивы компилятору *#include* использовать во всей программе только одну копию описаний *extern* и присоединять ее к каждому файлу во время его препроцессорной обработки.

Если переменная с таким же идентификатором, как внешняя, декларирована в функции без указания *extern*, то тем самым она становится внутренней (локальной) для данной функции.

Не стоит злоупотреблять внешними переменными, так как такой стиль программирования приводит к программам, связи данных внутри которых не вполне очевидны. Переменные при этом могут изменяться неожиданным образом. Модификация таких программ вызывает затруднения.

Пример, иллюстрирующий использование внешних данных:

Основной файл проекта	Дополнительный файл
<pre>... int x, y; char str[] = "Rezult = "; void fun1(void); void fun2(void); void fun3(void); void main(void) { fun1(); fun2(); fun3(); } void fun1(void) { y = 15; printf("\n %s %d\n", str, y); }</pre>	<pre>... extern int x, y; extern char str[]; int r = 4; void fun2(void) { x = y / 5 + r; printf(" %s %d\n", str, x); } void fun3(void) { int z= x + y; printf(" %s %d\n", str, z); }</pre>

В результате выполнения этого проекта, состоящего из двух различных файлов, будет получено следующее:

```
Rezult = 15
Rezult = 7
Rezult = 22
```

Область действия переменных

В языке C/C++ нет ключевого слова, указывающего область действия (видимости) объекта. Область действия определяется местоположением декларации объекта в файле исходного текста программы.

Вспомним общую структуру исходного текста программы:

```
<директивы препроцессора>
<описание глобальных объектов>
<заголовок функции>
{
  <описание локальных объектов>
  <список инструкций>
}
```

Файл исходного текста может включать любое количество определений функций и/или глобальных данных.

Параметры функции являются локальными объектами и должны отличаться по идентификаторам от используемых в коде функции глобальных объектов. Локальные объекты, описанные в коде функции, имеют приоритет перед объектами, описанными вне функции, например:

```
#include<stdio.h>
int f1(int);
int f2(int);
int f3(int);
int n;                                // Глобальная n
void main (void)
{
    int i=2;                          // Локальная i=2
    n=3;                              // Глобальная n=3
    i = f1(i);                        // Обе переменные i – локальные
    printf(" 1: i=%d , n=%d\n",i,n);  // i=7, n=3
    n = f1(i);                        // n – глобальная, i – локальная
    printf(" 2: i=%d , n=%d\n",i,n);  // i=7, n=12
    i = f2(n);                        // i и n – локальные
    printf(" 3: i=%d , n=%d\n",i,n);  // i=15, n=12
    i = f3(i);                        // Обе переменные i – локальные
    printf(" 4: i=%d , n=%d\n",i,n);  // i=29, n=14
}
// -----
int f1(int i) {                       // Параметр функции i – локальная
    int n = 5;                        // n – локальная
    n+=i;
    return n;
}
```

```
//-----  
int f2(int n) {           // Параметр функции n – локальная  
    n+=3;  
    return n;  
}  
//-----  
int f3(int i) {  
    n+=2;                // n – глобальная  
    return i+n;  
}
```

Существуют следующие области действия:

- блок,
- файл,
- функция,
- прототип функции,
- область структурированных типов данных.

Блок. Идентификаторы, описанные внутри блока, являются локальными. Область действия идентификатора начинается в точке определения и заканчивается в конце блока, видимость – в пределах блока и внутренних блоков, время жизни – до выхода из блока. После выхода из блока память освобождается.

Файл. Идентификаторы, описанные вне любого блока, функции, класса или пространства имен, имеют глобальную видимость и постоянное время жизни и могут использоваться с момента их определения.

Функция. Единственными идентификаторами, имеющими такую область действия, являются метки операторов. В одной функции все метки должны различаться, но могут совпадать с метками других функций.

Прототип функции. Идентификаторы, указанные в списке параметров прототипа (декларации) функции, имеют областью действия только прототип функции.

Структурированный тип данных. Элементы структур и объединений являются видимыми лишь в их пределах. Они образуются при создании переменной указанного типа и разрушаются при ее уничтожении.

Допускается в разных блоках программы использовать один и тот же идентификатор для разных объектов. Декларация такого идентификатора внутри блока скрывает доступ к ранее объявленному, например:

```
void main(void)  
{  
    int a = 3;  
    printf("\n Block 1: %d ", a); {
```

```
double a = 2.5;
printf( "\n Block 2: %lf ", a); {
    char a = 'A';
    printf( "\n Block 3: %c ", a);
}
printf( "\n New Block 2: %lf", a+1.25);
}
printf( "\n New Block 1: %d", ++a);
}
```

Результат программы:

Block 1: 3

Block 2: 2.5

Block 3: A

New Block 2: 3.75

New Block 1: 4

Выделение памяти для динамических объектов. Доступ к динамическим объектам. Освобождение памяти. Изменение памяти.

C/C++ поддерживает три основных типа **выделения (распределения) памяти**, с двумя из которых мы уже знакомы:

Статическое выделение памяти выполняется для **статических и глобальных переменных**. Память выделяется один раз, при запуске программы, и сохраняется на протяжении работы всей программы.

Автоматическое выделение памяти выполняется для **параметров функции и локальных переменных**. Память выделяется при входе в блок, в котором находятся эти переменные, и освобождается при выходе из него.

Как статическое, так и автоматическое распределение памяти имеют две общие черты:

- Размер переменной/массива должен быть известен во время компиляции.
- Выделение и освобождение памяти происходит автоматически (когда переменная создается или уничтожается).

В большинстве случаев с этим всё ОК. Однако, когда дело доходит до работы с внешним вводом, то эти ограничения могут привести к проблемам.

Например, при использовании **строки** для хранения имени мы не знаем, насколько длинным оно будет, пока пользователь его не введет. Или, когда нам нужно записать количество записей с диска в переменную, но заранее неизвестно, сколько таких записей есть. Или можно создать игру с непостоянным количеством монстров (во время игры одни монстры умирают, другие рождаются), пытаясь, таким образом, убить игрока.

Если нужно объявить размер всех переменных во время компиляции, то самое лучшее, что можно сделать — это попытаться угадать их максимальный размер, надеясь, что этого будет достаточно:

```
char name[30]; // будем надеяться, что пользователь введет имя менее 30 символов!  
Record record[400]; // будем надеяться, что количество записей будет не больше 400!  
Monster monster[30]; // 30 монстров максимум
```

Это плохое решение, по крайней мере, по трем причинам:

Во-первых, теряется память, если переменные фактически не используются или используются, но не на полную. Например, если мы выделим 30 символов для каждого имени, но имена в среднем будут занимать по 15 символов, то потребление памяти получится в два раза больше, чем нужно на самом деле.

Во-вторых, память для большинства обычных переменных (включая фиксированные массивы) выделяется из специального резервуара памяти — **стека**. Объем памяти стека в программе, как правило, невелик. Если превысить это число, то произойдет *переполнение стека*, и операционная система автоматически завершит выполнение вашей программы.

В-третьих, и самое главное, это может привести к искусственным ограничениям и/или переполнению массива. Что произойдет, если пользователь попытается прочесть 500 записей с диска, но памяти выделено максимум для 400? Либо мы выведем пользователю ошибку, что максимальное количество записей — 400, либо (в худшем случае) выполнится переполнение массива и затем что-то очень нехорошее.

К счастью, эти проблемы легко устраняются с помощью динамического выделения памяти. **Динамическое выделение памяти** — это способ запроса памяти из операционной системы запущенными программами при необходимости. Эта память не выделяется из ограниченной памяти стека программы, а из гораздо большего хранилища, управляемого операционной системой — **heap (кучи)**. На современных компьютерах размер кучи может составлять гигабайты памяти.

Динамическое выделение переменных

Для динамического выделения памяти для одной переменной используется оператор **new**:

```
new int;
```

Здесь мы запрашиваем выделение памяти для целочисленной переменной из операционной системы. Оператор **new** возвращает **указатель**, содержащий адрес выделенной памяти.

Для доступа к выделенной памяти создается указатель:

```
int *ptr = new int;
```

//динамически выделяем целочисленную переменную и присваиваем её адрес ptr, чтобы потом иметь возможность доступа к ней

```
*ptr = 8;
```

// присваиваем значение 8 только что выделенной памяти

Это один из случаев, когда указатели полезны. Без указателя с адресом на только что выделенную память, у нас бы не было способа получить доступ к ней.

Как работает динамическое выделение памяти?

На компьютере имеется память (возможно, большая ее часть), которая доступна для использования приложениями. При запуске приложения операционная система загружает это приложение в некоторую часть этой памяти. И эта память, используемая приложением, разделена на несколько частей, каждая из которых выполняет определенную задачу. Одна часть содержит код, другая используется для выполнения обычных операций (отслеживание вызываемых функций, создание и уничтожение глобальных и локальных переменных и т.д.). Тем не менее, большая часть доступной памяти просто находится там, ожидая запросов на выделение от программ.

Когда вы динамически выделяете память, вы просите операционную систему зарезервировать часть этой памяти для использования вашей программой. Если ОС может выполнить этот запрос, то возвращается адрес этой памяти обратно, в приложение. С этого момента и в дальнейшем ваше приложение может использовать эту память, как только пожелает. Когда вы уже выполнили всё, что было необходимо, с этой памятью, то её нужно вернуть обратно в операционную систему, для распределения между другими запросами.

В отличие от статического или автоматического выделения памяти, сама программа отвечает за запрос и обратный возврат динамически выделенной памяти.

Инициализация динамически выделенных переменных

Когда вы динамически выделяете переменную, то вы также можете её инициализировать посредством **прямой инициализации**

```
int *ptr1 = new int (7);
```

// используем прямую инициализацию

Удаление переменных

Когда уже всё, что нужно было, выполнено с динамически выделенной переменной – нужно явно указать, что нужно освободить эту память. Для отдельных переменных это выполняется с помощью оператора **delete**:

```
// предположим, что ptr ранее уже был выделен с помощью оператора new
delete ptr;
```

//возвращаем память, на которую указывал ptr, обратно в операционную систему

```
ptr = 0;
```

//делаем ptr нулевым указателем

Оператор delete на самом деле ничего не удаляет. Он просто возвращает память, которая была выделена ранее, обратно в операционную систему. Затем операционная система может переназначить эту память другому приложению (или этому же снова).

Хотя может показаться, что мы удаляем *переменную*, но это не так! Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, и ей можно присвоить новое значение, как и любой другой переменной.

Обратите внимание, *удаление указателя, не указывающего на динамически выделенную память, может привести к проблемам.*

Висячие указатели

Нет никаких гарантий относительно того, что произойдет с содержимым освобожденной памяти, или со значением удаляемого указателя. В большинстве случаев память, возвращаемая операционной системе, будет содержать те же значения, которые были у нее до *освобождения*, а указатель так и останется указывать на память, только уже освобожденную (удаленную).

Указатель, указывающий на освобожденную память, называется **висячим указателем**. Разыменование или удаление висячего указателя приведет к неожиданным результатам. Рассмотрим следующую программу:

```
#include <iostream>
int main()
{
    int *ptr = new int; // динамически выделяем целочисленную переменную
    *ptr = 8;           // помещаем значение в выделенную ячейку памяти

    delete ptr; //возвращаем память обратно в операционную систему. ptr
теперь уже висячий указатель

    std::cout << *ptr; //разыменование висячего указателя приведет к
неождаанным результатам
    delete ptr; // попытка освободить память снова приведет к неожиданным
результатам также

    return 0;
}
```

Здесь значение 8, которое ранее было присвоено выделенной памяти, после освобождения может и далее находиться там, а может и нет. Также возможно, что освобожденная память уже могла быть выделена другому приложению (или для собственного использования операционной системы), и попытка доступа к ней приведет к тому, что операционная система автоматически прекратит выполнение программы.

Процесс освобождения памяти может также привести к созданию *нескольких* висячих указателей. Рассмотрим следующий пример:

```
#include <iostream>
int main()
{
    int *ptr = new int;    // динамически выделяем целочисленную переменную
    int *otherPtr = ptr;  // otherPtr теперь указывает на ту же самую
                          // выделенную память, что и ptr

    delete ptr; //возвращаем память обратно в операционную систему. ptr и
otherPtr теперь висячие указатели
    ptr = 0; //ptr теперь уже нулевой указатель

    // однако otherPtr по-прежнему является висячим указателем!

    return 0;
}
```

Есть несколько рекомендаций, которые могут здесь помочь.

Во-первых, старайтесь избегать ситуаций, когда несколько указателей указывают на одну и ту же часть выделенной памяти. Если это невозможно, то проясните, какой указатель из всех «владеет» памятью (и отвечает за ее удаление), а какие указатели просто получают доступ к ней.

Во-вторых, когда вы удаляете указатель, и если он не выходит из области видимости сразу же после удаления, то его нужно сделать нулевым, т.е. задать значение 0. Под «выходом из области видимости сразу же после удаления» имеется в виду, что вы удаляете указатель в самом конце блока, в котором он объявлен.

Правило: Присваивайте удаленным указателям значение 0, если они не выходят из области видимости сразу же после удаления.

!!! В C++ 11 вместо 0 нужно использовать nullptr

Работа оператора new

При запросе памяти из операционной системы в редких случаях она может быть недоступной (т.е. её может и не быть в наличии).

По умолчанию, если new не сработал, память не выделилась, то генерируется исключение *bad_alloc*. Если это исключение будет неправильно обрабатываться (а именно так и будет, поскольку мы еще не рассмотрели исключения и их обработку), то программа просто прекратит своё выполнение (произойдет сбой) с необработанной ошибкой исключения.

Во многих случаях процесс генерации исключения оператором new (как и сбой программы) нежелателен, поэтому есть альтернативная форма new, которая возвращает нулевой указатель, если память не может быть выделена. Нужно просто добавить константу *std::nothrow* между ключевым словом new и типом выделения данных:

```
int *value = new (std::nothrow) int;
```

//указатель value станет нулевым, если динамическое выделение целочисленной переменной не выполнится

В примере выше, если new не возвратит указатель с динамически выделенной памятью, то возвратится нулевой указатель.

Разыменовывать его также не рекомендуется, так как это приведет к неожиданным результатам (скорее всего, к сбою в программе). Поэтому, наилучшей практикой является проверка всех запросов на выделение памяти, для обеспечения того, что эти запросы выполняться успешно и память будет выделена.

```
int *value = new (std::nothrow) int;
```

// запрос на выделение динамической памяти для целочисленного значения

if (!value) // обрабатываем случай, когда new возвращает null (т.е. память не выделяется)

```
{
```

// обработка этого случая

```
std::cout << "Could not allocate memory";
```

```
}
```

Поскольку невыделение памяти оператором new происходит крайне редко, то обычно программисты забывают выполнять эту проверку!

Нулевые указатели и динамическое выделение памяти

Нулевые указатели (указатели со значением 0 (nullptr)) особенно полезны в процессе выделения динамической памяти. Их наличие как бы говорит: «этому указателю не выделено никакой памяти». А это в свою очередь можно использовать для выполнения условного выделения памяти:

// если ptr-у до сих пор не выделено памяти, выделяем её

```
if (!ptr)
```

```
ptr = new int;
```

Удаление нулевого указателя ни на что не влияет. Таким образом, в следующем нет необходимости:

```
if (ptr)
```

```
delete ptr;
```

Вместо этого вы можете просто написать:

```
delete ptr;
```

Если ptr не является нулевым, то динамически выделенная переменная будет удалена. Если значением указателя является ноль, то ничего не произойдет.

Утечка памяти

Динамически выделенная память не имеет области видимости. То есть она остается выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершится (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости нормальных переменных. Это несоответствие может вызвать интересное поведение.

Рассмотрим следующую функцию:

```
void doSomething()
{
    int *ptr = new int;
}
```

Здесь мы динамически выделяем целочисленную переменную, но никогда не освобождаем память через `delete`. Поскольку указатели следуют всем тем же правилам, что и обычные переменные, то, когда функция завершится, `ptr` выйдет из области видимости. Поскольку `ptr` – это единственная переменная, хранящая адрес динамически выделенной целочисленной переменной, то, когда `ptr` уничтожится — больше не останется ссылок (адресов) на динамически выделенную память. Это означает, что программа «потеряет» адрес динамически выделенной памяти. И в результате эту динамически выделенную целочисленную переменную нельзя будет удалить.

Это называется **утечкой памяти**. Утечка памяти происходит, когда ваша программа теряет адрес некоторой динамически выделенной части памяти (например, переменной или массива), прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где та находится. Операционная система также не может использовать эту память, поскольку считается, что та по-прежнему используется вашей программой.

Утечки памяти съедают свободную память во время выполнения программы, уменьшая количество доступной памяти не только для этой программы, но и для других программ также. Программы с серьезными проблемами с утечкой памяти могут съесть всю доступную память, в результате чего весь ваш компьютер будет медленнее работать или даже произойдет сбой. Только после того, как выполнение вашей программы завершится, операционная система сможет очистить и *вернуть* всю просочившуюся память.

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
int value = 7;  
int *ptr = new int; // выделяем память  
ptr = &value; // старый адрес утерян - произойдет утечка памяти
```

Это легко решается удалением указателя перед операцией переприсваивания:

```
int value = 7;  
int *ptr = new int; // выделяем память  
delete ptr; // возвращаем память обратно в операционную систему  
ptr = &value; // переприсваиваем указателю адрес value
```

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

```
int *ptr = new int;  
ptr = new int; // старый адрес утерян - произойдет утечка памяти
```

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти!

Точно так же этого можно избежать удалением указателя перед операцией переприсваивания.