Указатели в C++

Традиционно память в компьютере рассматривается как последовательный набор ячеек-байтов. Каждая ячейка памяти однозначно определяется ее адресом, адреса нумеруются от 0 и по возрастающей пока ячейки физически не кончатся.

Во время работы программа находится в памяти, поэтому каждый элемент программы обладает собственным адресом (например, константы, переменные и даже функции). Занимаемая элементом память определяется типом этого элемента и способом его определения.

В C/C++ есть оператор &, который позволяет получить адрес элемента (этот оператор унарный, не путать с бинарным &, который обозначает битовую конъюнкцию аргументов). Рассмотрим следующий код:

```
int a = 5;
int b = a;
printf("a: %d, (%u)\n", a, &a);
printf("b: %d, (%u)\n", b, &b);
```

Вывод:

```
a: 5, (2048930876)
b: 5, (2048930872)
```

Сначала выводится значение переменной a (число 5), а потом в скобках адрес в памяти, с которым ассоциирована сейчас переменная a и по которому лежит это самое значение 5. Понятно, что при каждом запуске программа может находиться в различных местах памяти, поэтому числа в скобках могут меняться. Вторая строчка выводит аналогичную информацию для переменной b.

Из этого примера можно сделать несколько важных выводов:

- Каждая переменная имеет свой отдельный адрес в памяти, причем от запуска к запуску этот адрес может меняться.

- Когда вы пишете имя переменной, реально будет использоваться ее значение. Таким образом, присваивание b = a записывает в b значение переменной a, адреса же при этом не меняются.
- Переменные, объявленные рядом в коде, будут иметь соседние адреса. Адрес переменнной b на 4 меньше, чем адрес переменной a (2048930876 - 2048930872 = 4), т.е. в памяти переменные a и b идут друг за другом.

Это все , конечно же, верно только для переменных, определенных на стеке, но об этом чуть позже.

Очень важная операция, которую можно сделать с адресом — сохранить его в другой переменной для последующего использования. Для хранения адресов памяти в C++ существует специальный тип данных, называемый указателем. Задается он унарным оператором *, причем от умножения компилятор его отличает тоже по контексту, как и &.

При определении указателя необходимо задавать тип переменной, на которую он указывает. Для инициализации указателя нужно использовать константу NULL (nullptr в C++11). Такой указатель будет "никуда не показывать". Выглядит это как-то так:

```
int *pointer = NULL;
int someValue = 5;
int *somePointer = &someValue;
```

Неинициализированные указатели — очень страшный грех и заведомый источник огромного количества сложнообнаружимых ошибок в коде, поэтому нужно приучить себя сразу же занулять все указатели при создании (ну или инициализировать их чем-то полезным, если есть такая возможность).

Простейшее применение хорошего проинициализированного указателя — работа со значением, на которое он ссылается. Чтобы обратиться к значению через указатель, следует разыменовать его тем же оператором *, который использовался для объявления этого указателя:

```
*somePointer = 10;
```

Стековая и динамическая память

Иногда количество, типы и срок жизни элементов в программе точно известны заранее, но так происходит далеко не всегда. Даже вы уже в своих задачах сталкивались с тем, что хочется выделить память под массив, однако размер этой памяти зависит от того, сколько элементов в массиве захочет пользователь. Понятно, что проблема эта общая — нужен механизм создания и уничтожения объектов данных не на стадии компиляции, а во время выполнения программы.

Память, с которой работает программист, делится на стековую и динамическую (на самом деле это не полный список, но для наших целей его хватит).

Стековая память — это специальный кусок памяти, резервируемый при запуске программы (до вызова функции main()) из свободной оперативной памяти и используемый в дальнейшем для размещения локальных объектов: объектов, определяемых в теле функций и получаемых функциями через параметры в момент вызова. Такую память еще часто называют автоматической, но мы будем просто называть ее стеком. Создание и удаление данных на стеке компилятор осуществляет

автоматически.

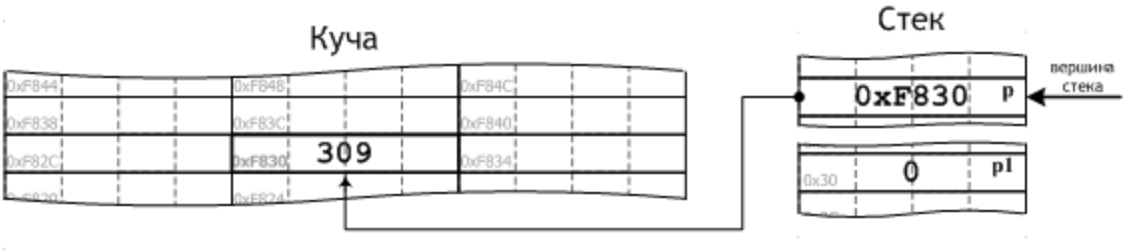
Динамическая память — это совокупность блоков памяти, выделяемых из доступной свободной оперативной памяти непосредственно во время выполнения программы под размещение конкретных объектов. Управление этой памятью — всецело задача программиста. Такую память называют кучей. Куча и стек — это разные регионы адресного пространства, часто расположенные на значительном удалении друг от друга, в связи с чем адреса объектов, размещённых в куче, обычно весьма существенно отличаются от адресов объектов, размещённых в стеке.

В C есть функции malloc() и free(), которые позволяют выделять и освобождать куски памяти требуемого размера. Это очень низкоуровневые операции, которые никак не учитывают данные, под которые эта память выделяется, к тому же это библиотечные функции. В C++ для этого используются операторы new и delete (важность этих операторов подтверждает даже то, что в C++ они введены в основной синтаксис языка).

Рассмотрим следующий код:

```
int* p = new int(309);
```

Этот код выделяет в куче непрерывный участок памяти, по размеру равный числу байт, необходимых для хранения целого числа, затем на стеке определяется указатель p, который сразу инициализируется адресом первого байта этого нововыделенного куска динамической памяти. Формат вызова оператора new таков: ключевое слово new, за которым следует тип создаваемого объекта, за которым в круглых скобках может следовать список параметров, необходимых для начальной инициализации объекта. Выглядеть это будет как-то так:



Оператор new создаёт в куче объект типа int со значением 309 и возвращает адрес этого объекта — например, число 0xF830 (адрес в шестнадцатеричной записи). Компилятор создаёт в стеке объект-указатель p и инициализирует его адресом, возвращённым оператором new — в ячейки памяти, отведённые в стеке под p, записывается число 0xF830.

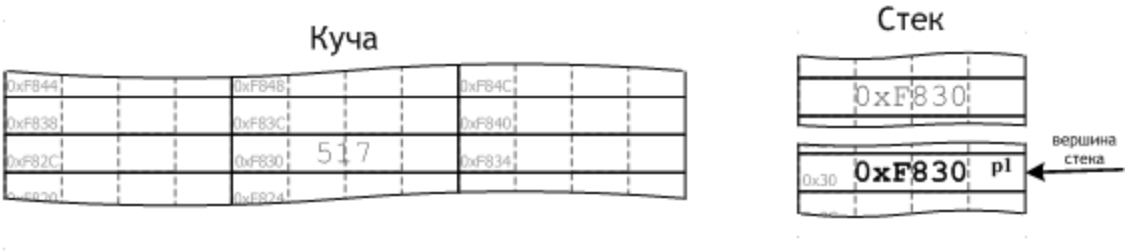
В итоге рассматриваемая строчка порождает в памяти два объекта: объект целочисленного типа int со значением 309 в куче и объект-указатель p в стеке. Адрес созданного в куче объекта является значением стекового объекта-указателя p.

Для того, чтобы освободить выделенный таким образом участок памяти, необходимо воспользовать оператором delete:

```
delete p;
```

Важный момент — оператор delete освобождает выделенную память, но никак не меняет значение самого указателя p. Т.е. после выполнения delete p в p будет лежать

тот же самый адрес, только диспетчер памяти операционной системы будет считать эту память уже свободной.



К слову, оператор delete также никак не меняет те данные, которые лежат в освобождаемом куске памяти. Это важно, если вы потом захотите обратиться к чему-то по этому указателю — там могут лежать те же самые данные, а может уже и какие-то другие (если кто-то успел получить этот кусок себе и записать туда что-то). Именно поэтому если вы хотите использовать этот указатель как-то дальше после выполнения delete (например, в цикле), надо его занулить (присвоить ему значение NULL или nullptr в зависимости от версии используемого C++).

Способы передачи аргументов в функции

Рассмотрим варианты передачи аргументов в функции. В C++ их существует ровно три.

Передача параметра по значению

Это то, как большая часть из вас писала функции до этого момента.

```
void print(int value)
{
    cout << value << endl;
}
…
int x = 5;
print(x);
```

Функции, описанные таким образом, создают внутренние копии всех своих аргументов, передаваемых по значению. Важное следствие — все изменения, сделанные с таким аргументом внутри функции, будут иметь силу только локально внутри этой функции, т.е. функцию swap(), например, таким образом не реализовать. Чтобы писать функции, которые будут менять значение передаваемых аргументов, нужно передавать параметр либо по указателю, либо по ссылке.

Передача параметра по указателю

Выглядит это следующим образом:

```
void increment(int *value)
{
    (*value)++;
}
```

```
…
int x = 5;
increment(&x);
```

На то, что аргумент передается по указателю, указывает знак * после типа и перед именем этого аргумента. Функция increment() получает в качестве аргумента указатель на какую-то ячейку памяти, разыменовывает его и получает возможность работать со значением, на которое этот указатель указывает. Кто любит проверять все своими руками, может напечатать этот указатель и адрес переменной, которая будет использована при вызове этой функции — это будет один и тот же адрес.

Такой механизм есть и в языке C, причем там это единственный вариант изменять внешние объекты, так что им активно пользуются везде, где можно и где нельзя. Например, в C широко распространенная практика — передавать аргументы как указатели на тип void — int f(void *arg). Прелесть такого подхода в том, что в f() можно передать адрес чего угодно. Все, что угодно вообще!!1 Потом внутри f() это что-то будет отконвертировано к нужному типу и как-то будет использоваться. Большая нетривиальная программа с частым использованеим void * — это без преувеличения ночной кошмар любого программиста, поскольку компилятор теперь не имеет никакой возможности проверить корректность типов передаваемых значений и вся ответственность ложится на программистов (которые, как известно, склонны ошибаться чуть более, чем везде). В более высокоуровневых, чем C, языках такой подход считается дурным тоном и настоятельно не рекомендуется к использованию.

Возвращаясь к передаче аргументов в функции, в C++ появился новый способ передачи адресов функциям — по ссылке (о нем следующий параграф). Стоит отметить, что для работы с массивами (в частности, со строками в стиле языка C) подход с указателями работает хорошо, поскольку имя массива — это по сути указатель на первый его элемент, так что в большом числе случаев они взаимозаменяемы.

Передача параметра по ссылке

Выглядит это как-то так:

```
void increment(int &value)
{
    value++;
}
…
int x = 5;
increment(x);
```

Общий принцип работы такого механизма похож на предыдущий пример — в функцию так же передается адрес аргумента. Различие между передачей по ссылке и по указателю состоит в том, что вызов функции с передачей аргумента по значению выглядит куда более понятно и наглядно, нежели в прошлом варианте с указателями.

Литература

Очень хорошая статья про указатели и динамическую память вообще, я честно спер оттуда картинки: http://www.rsdn.ru/article/cpp/ObjectsAndPointers.xml