Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

19 сентября 2017 г.

Оглавление

Лекция 3

Адреса и указатели

3.1 Доступ к памяти

Память, выделяемая программе, неоднородна – она делится на **сегменты**. Первый сегмент — коды программы (режим "только чтение"). Второй – сегмент DATA – содержит данные программы. Более того, элементы DATA (к примеру, начальные значения переменных) записывается в .exe файл. Эти данные уходят в оперативку при запуске программы. Переменные, записанные в DATA, называются *статическими*. Помимо этого, для функционирования программы необходима область *стека* — такой сегмент памяти, который заполнятся снизу вверх и изначально пуст. Когда ОС только загрузилась, в стек выделяется некоторое количество автоматической памяти – для *локальных переменных*.
Память программе выделяется на время ее работы; после ее завершения память высвобождается.

Код 3.1: Пример многократных вызовов функции

```
1  #include<iostream>
2
3  using namespace std;
4
5  void count(int n)
6  {
7   int k = 1;
8   while(k <= n)
9   {
10   cout << k << " ";
11   k++;
12  }
13  cout << "\n";
14 }
15
16 int main()
```

```
17 {
18  count(5);
19  count(10);
20 }
```

Рассмотрим, как происходит работа с памятью в примере 3.1. Перед вызовом функции в стек программы будет выделено место для хранения "5"(оно в программе будет интерпретировано как n), stack pointer поднимается. stack pointer - это указатель, показывающий на текущее положение стека. Затем, после вызова функции, ее адрес возврата будет записан в стек - stack pointer двинется выше. Потом в стек уйдет "1"(мы же объявили ее).
Попробуем написать функцию, считающую количество рекурсивных вызовов.

Код 3.2: Функция для подсчета количества собственных вызовов

```
1  #include<iostream>
2
3  using namespace std;
4
5  int k2 = 0;
6
7  void count2()
8  {
9   k2++;
10  cout << k2 << " ";
11 }
12
13 void reducer()
14 {
15  k2--;
16 }
17
18 int main()
19 {
20  count2();
21  reducer();
22  // any your staff ..
23 }
```

Здесь k2 — глобальная переменная. Ее присваивание происходит в *момент компиляции* программы (т.е. она запишется в .exe и будет переноситься в память в момент загрузки). Функция **reducer** уменьшает значение k2.
Однако глобальные переменные могут быть опасны! Считается, что нормальная функция не заглядывает в глобальные переменные (в таких случаях функцию называют "чистой в противном - "грязной").

3.2 Модификатор static

Код 3.3: Та же программа через static

```
#include<iostream>

using namespace std;

//int k2 = 0;

void count2()
{
 static int k3 = 0;
 k3 ++ ;
 cout << k3 << endl;
}

void reducer()
{
 k3--; // won't work, k3 is not defined in this block
}

int main()
{
 count2();
 reducer();
 // any your staff ..
}

// Recursive func sample
void count(int n)
{
 int k = 1;
 if(n > 0)
  count(n-1);
}
```

Мы переделали программу 3.2, теперь переменная k3 не видна в других блоках, но по-прежнему она записывается в сам .exe . Если бы функция была рекурсивной, то была бы такая ситуация, когда переменных k было бы много - на каждом слое рекурсии она создается. Они просто будут записаны в разные ячейки памяти и уйдут после того, как рекурсивный вызов завершится.

Код 3.4: Подсчет рекурсии

```
#include<iostream>

using namespace std;
```

```
4
5  //int k2 = 0;
6
7  void count2()
8  {
9   static int k3 = 0;
10  k3++ ;
11  cout << k3 << endl;
12  if(n > 0)
13   count2(n-1)
14  k3--;
15 }
16
17 int main()
18 {
19  count2();
20  // any your staff ..
21 }
```

В коде выше как раз **static** делает k одним для всех вызовов рекурсии — оно при каждом вызове не будет присваиваться в ноль. Вместо этого переменная будет увеличиваться на 1, таким образом, мы сможем увидеть глубину рекурсии.

3.3 Указатель

Указатель - переменная адресного типа.

Код 3.5: Адрес x можно указать

```
1 int x;
2 &x; // pointer to x
3     // it has type 'int*'
4 int* p = &x;
5 double* pd = &x; // won't work
```

Указатели тоже работают только со своими типами: в коде выше нельзя в **double*** записать число **int**. Дело в том, что возникнет ложная интерпретация – ссылка на double превратися в ссылку на int, а поскольку в памяти double и int имеют разные размеры, то можно даже вылезти за пределы своей памяти. Компилятор это запретит, однако все можно обойти:

Код 3.6: Пример обхода заперта

```
1 int x;
2 double* pd = (double*) &x;
```

```
3
4  int main()
5  {
6  return 0;
7  }
```

У указателей могут тоже быть указатели:

Код 3.7: Указатель на указатель

```
1  int x;
2  int* p = &x;
3  int** pp = &p;
4  int*** ppp = &pp;
5  // ...
```

Обратиться к переменной можно несколько способами:

Код 3.8: Разница

```
1  int x;
2  int* p = &x;
3  cout << x; // print x
4  cout << p; // print ADDRESS of x
5  cout << *p; // print x
```

В последней строке происходит *разыменование* адреса - грубо говоря, считывание данных из того адреса, куда указывает указатель. Разыменовывать надо последовательно:

Код 3.9: Двойное разыменование

```
1  cout << x;
2  cout << *p;
3  cout << **pp;
4  cout << ***ppp;
```

3.4 Несколько слов о массивах

Массивов в C++ нет. Массив - это константный указатель на область памяти, представляющую содержимое!

Код 3.10: Работа массива

```
1 int A[5] = {0, 1, 2, 3, 4};
2 int* p = A; // works!
3 *p = 7
4 p + 1 // magic, read in text
5
6 // Furthermore:
7 A[i] === *(A+i)
```

Команда объявления массива по факту означает следующее: создать n переменных типа int.

С указателями можно прийти к *адресной арифметике*: так, указатель $p+1$ выше означает, что мы сдвигаемся по памяти от места, куда указывает p, на **sizeof(1)**. Это иллюстрирует строка 7: A[i] абсолютно равносильна указателю *(A+i). Проверки на i *нет*, то есть, в теории, мы можем убежать вообще в произвольную область памяти.

3.5 Динамическая память

Код 3.11: Пример выделения из дин. памяти

```
1  int get()
2  {
3   int* p = new int[1000]
4   return p;
5  }
6
7  int main()
8  {
9   int* q;
10  q = get();
11  // ...
12  // after your code
13  q = get(); // takes another amount of mem
14  delete[] q;
15 }
```

Динамическая память не является автоматической, т.е. ей надо явным образом указывать, когда ее выделять, а когда удалять. Если без конца писать вызов get(), то можно постоянно выбивать у системы память и это все приведет к *memory leak* - утечке памяти.

3.6 Передача параметров по указателю

Передача по параметрам бывает полезна, когда хочется поменять значение переменной внутри самой функции. Функции передается указатель и она, имея ссылку непо-

средственно на область памяти, может ее менять:

Код 3.12: Передача указателей функции

```
1  void changer(int* x)
2  {
3   (*x)++;
4  }
5
6  // You can use another construction in C++
7
8  void changer1(int& y)
9  {
10      y++;
11 }
12
13
14 int main()
15 {
16  int x;
17  changer(&x);
18  changer1(x); // no need for &
19  cout << x;
20 }
```

Подробнее – в следующей лекции.