Оглавление

Модель памяти в C++

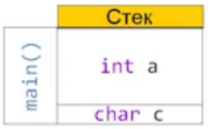
3.1. Модель памяти

В этом разделе мы изучим, как эффективно использовать основные типы языка C++. Для этого необходимо изучить, как они устроены внутри.

3.1.1. Введение в модель памяти: стек

Рассмотрим пример:

```
void second() {
  int s_a = 3;
  double s_d = 2.0;
}

void first() {
  int f_a = 2;
  char f_c = 'a';
  second();
}

int main() {
  int a = 1;
  char c = 'r';
  first();
  second();
  a = 2;
  c = 'q';
}
```



У каждой функции есть локальные переменные. Они хранятся в специальной области оперативной памяти, которая называется стек.

Когда в программе запускается очередная функция, то на стеке резервируется блок памяти, достаточный для хранения локальных переменных. Кроме того, в этом блоке хранится служебная информация, такая как адрес возврата. Такая область памяти называется стековым фреймом функции. Когда функция `main` запускает функцию `first`, то на стеке ниже функции `main` резервируется фрейм для функции `first`. То же самое происходит, когда функция `first` вызывает функцию `second`.

Когда функция `second` завершает свою работу, то вершина стека перемещается на фрейм предыдущей функции. При этом фрейм отработавшей функции просто остаётся на стеке. Выйдем из функции `first` в функцию `main`, запустим функцию `second` из `main`.

Стековый фрейм перетирает данные, которые были в стеке от предыдущих вызовов. Выйдем из функции `second` и выйдем из функции `main`. Когда выходим из программы, то вершина стека поднимается до самого верха.

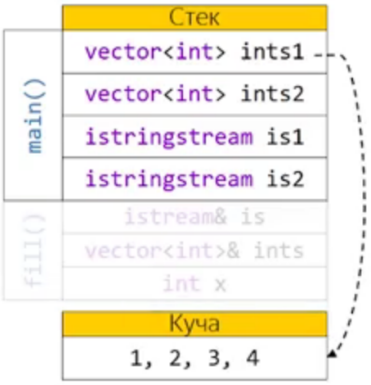
3.1.2. Введение в модель памяти: куча

Рассмотрим пример

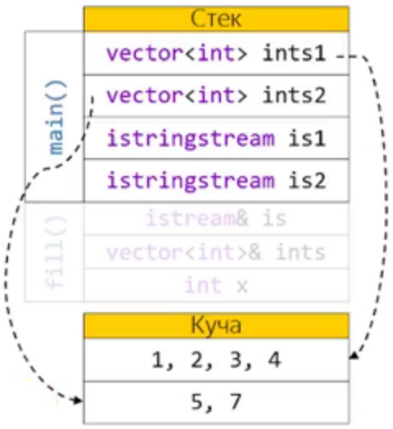
```
void fill(istream& is, vector<int>& ints) {
  int x;
  while (is >> x) {
    ints.push_back(x);
  }
}

int main() {
  vector<int> ints1, ints2;
  istringstream is1("1 2 3 4");
  fill(is1, ints1);
  istringstream is2("5 7");
  fill(is2, ints2);
}
```

В функции `main` объявлены два вектора целых чисел, также объявлены два потока ввода из строк. Кроме того, есть функция `fill`, она читает из потока числа и помещает их в вектор. В результате работы программы в векторе `ints1` будут храниться цифры 1, 2, 3, 4; в векторе `ints2` будут храниться цифры 5, 7. Предположим, что числа, которые функция `fill` разместит в вектор, располагаются ниже её стекового фрейма. Это не так, потому что при повторном запуске `fill`, она перезапишет эти числа. Получается, в данном случае нам не подходит стек, потому что объекты, создаваемые функцией `fill`, должны жить дольше, чем функция `fill`. На стеке память постоянно будет перезаписываться вызовами новых функций.

Для хранения элементов вектора нам подходит источник памяти, который называется «куча». Во время работы программа может в любой момент обращаться в кучу и выделять в ней блок памяти произвольного размера или освобождать ранее выделенный. Посмотрим, как программа из примера использует кучу для хранения элементов вектора.



После завершения работы функции `fill` в куче будет выделен блок памяти, достаточный для хранения четырех целых чисел. При этом вектор `ints1` будет ссылаться на эту область памяти в куче и будет иметь доступ к своим элементам.



Когда `fill` отработает во второй раз, то в куче будет выделен отдельный блок памяти, достаточный для хранения двух целых чисел.

Подведём итог. В модели памяти C++ есть два источника памяти: стек и куча. В стеке размещаются локальные переменные функций. В куче размещаются объекты, которые должны жить дольше, чем создавшая их функция.

3.1.3. Оператор `new`

Возможность выделения памяти в куче доступна любому программисту. Чтобы выделить в куче память для хранения одного значения типа `int` нужно объявить локальную переменную типа `int*`.

```
int* pInt = new int;
```

`new` – это специальный оператор, который выполняет выделение памяти из кучи. Оператор `int*` подсказывает, для хранения какого типа нам нужна память. При этом `pInt` – это указатель на значение типа `int`. Сама переменная `pInt` будет храниться в стеке, но она будет указывать на область памяти в куче.

Указатель – это адрес в памяти. Память в C++ представляется как линейный массив байтов. Указатель – это индекс в этом массиве.

Как обратиться к значению, на которое указывает указатель? Синтаксис при работе с указатеями такой же, как при работе с итераторами.

Рассмотрим другой пример.

```
string* s = new string;
*s = "Hello";
cout << *s << ' ' << s->size() << endl;
// Hello 5
```

Оператор `*`, примененный к указателю, возвращает ссылку на объект в куче.

```
string* s = new string;
*s = "Hello";
string& ref_to_s = *s;
ref_to_s += ", world";
cout << *s << endl;
// Hello, world
```

Инициализация объектов, выделенных на куче, выполняется аналогично инициализации объектов, создаваемых на стеке при объявлении локальных переменных.

3.1.4. Оператор `delete`

Рассмотрим пример. У нас есть вектор целых чисел `v`, в отдельной области видимости объявлен итератор `iter`, в который записан результат поиска пятерки в нашем векторе `v`.

```
vector<int> v = {1, 2, 3, 4, 5};
{
  auto iter = find(begin(v), end(v), 5);
}
```

Что станет с пятёркой после выхода `iter` из области видимости? Ничего не произойдет, потому что итератор просто указывает позицию в контейнере и никак не управляет значением, на которое он указывает. Сам итератор разрушился, а значение в векторе никуда не делось, оно продолжает там храниться и мы можем им пользоваться.

Рассмотрим другой пример. Мы в отдельной области видимости выделяем в куче память для хранения целого числа и инициализируем её пятеркой, указатель на эту память мы записываем

в переменную `pFive`. Что станет с пятёркой в куче после выхода `pFive` из области видимости? Ничего – пятёрка останется в куче, и произойдёт утечка памяти, то есть память, выделенная на куче, будет числиться за нашим процессом и у нас не будет возможности обратиться к этой памяти и освободить её, потому что мы потеряли указатель на эту память, он вышел за пределы области видимости.

Напишем программу, которая будет считывать со входа число `n` и будет считать сумму `n` случайных 64-битных чисел.

```
int main() {
  int n;
  cin >> n;

  mt19937_64 random gen;  // генератор случайных чисел
  uint64_t sum = 0;
  for (int i = 0; i < n; ++i) {
    uint64_t x = random_gen();
    sum += x;
  }
  cout << sum;
}
// на вход подаём 10
// на выходе 4762160605604824475
```

Программа работает нормально. Теперь будем выделять в куче память для хранения очередного случайного числа.

```
for (int i = 0; i < n; ++i) {
  auto x = new uint64_t;
  *x = random_gen();
  sum += *x;
}
```

Такая программа также работает, но происходит утечка памяти. Это может стать проблемой при запуске программы на больших `n`. Бороться с этим можно при помощи оператора `delete`. Он освобождает память, выделенную оператором `new`.

```
for (int i = 0; i < n; ++i) {
  auto x = new uint64_t;
  *x = random_gen();
  sum += *x;
```

```
  delete x;
}
```

Программа работает, при этом количество выделенной памяти не растёт.

3.1.5. `new` и `delete` для объектов классового типа

Продемонстрируем, что при вызове оператора `new` для объектов классового типа оператор `new` не только выделяет необходимую память в куче, но и вызывает конструктор.

```
struct Widget {
  Widget() {
    cout << "constructor" << endl;
  }
};

int main() {
  new Widget;
}
// constructor
```

Когда для этого объекта вызывается деструктор? Напишем его.

```
~Widget() {
  cout << "destructor" << endl;
}
```

При выполнении программы в консоль вывелось только `constructor`. Деструктор не был вызван. Деструктор вызывает оператор `delete`.

```
int main() {
  Widget* w = new Widget;
  delete w;
}
// constructor
// destructor
```

Мы убедились, что оператор `delete` не только осовобождает место в памяти, но и вызывает деструктор для соответствующего объекта в куче.

3.1.6. Операторы new[] и delete[]

Напишем свой собственный `vector`. Начнём с простого интерфейса:

```
template <typename T>
class SimpleVector {
public:
  explicit SimpleVector (size_t size);
  ~SimpleVector();

private:
  T* data;
};

int main() {
  SimpleVector<int> sv(5);
}
```

Объявим шаблон класса `SimpleVector`. В интерфейсе нашего класса пока будут только конструктор и деструктор. Конструктор принимает количество элементов в нашем векторе. Также у нас будет приватное поле `data`, которая будет указателем на тип `T`.

Реализуем конструктор. Нам в куче нужно выделить `size` объектов. Пока мы умеем создавать только один объект. Несколько объектов можно создать с помощью оператора `new[]`. Он создает блок памяти для хранения необходимого количества объектов.

```
explicit SimpleVector (size_t size){
  data = new_T[size];
}
```

Сейчас в нашей программе есть утечка памяти. Освобождать её следует в деструкторе.

```
~SimpleVector() {
  delete[] data;
}
```

Если вмето `delete[]` пропишем просто `delete`, то программа будет работать некорректно.

3.1.7. Введение в арифметику указателей

Добавим в наш вектор возможность обращаться к отдельным элементам. У нас есть указатель `data` на первый элемент вектора. Указатели на другие элементы получаются очень просто: нужно добавить целое число, например, `data + 1` указывает на второй элемент вектора.

Можем написать оператор доступа по индексу.

```
T& operator[] (size_t index) {
  return *(data + index);
}
```

Теперь можно заполнять вектор числами и выводить их на экран.

```
int main() {
  SimpleVector<int> sv(5);
  for (int i = 0; i < 5; ++i) {
    sv[i] = 5 - i;
  }
  for (int i = 0; i < 5; ++i) {
    cout << sv[i] << ' ';
  }
}

// 5 4 3 2 1
```

В нашей программе мы выделили память на 5 элементов. Но если, например, попытаться вывести на экран `sv[12]`, то код скомпилируется и может даже отработать:

```
SimpleVector<int> sv(5);
cout << sv[12] << endl;
// 7827296
```

Язык C++ не контролирует доступ к данным, которые мы осуществляем через указатель. Сейчас мы прочитали элемент, который лежит за границами нашего вектора.

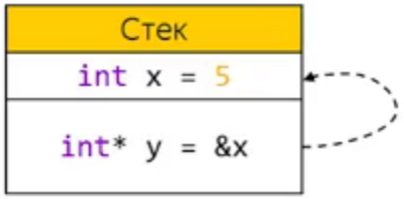
```
SimpleVector<string> sv(5);
cout << sv[12] << endl;
```

Такая программа упадёт, потому что мы обращаемся к случайному участку памяти, в котором не выполняются инварианты класса `string`.

К указателям можно не только прибавлять целые числа, но и вычитать их.

Локальные переменные хранятся на стеке. Стек – это область оперативной памяти, поэтому у его элементов тоже есть адрес. Его можно получить с помощью оператора `&`:

```
int main() {
  int x = 5;
  int* y = &x;
}
```



Переменная `y` хранит в себе адрес переменной `x`.

```
int main() {
  int x = 5;
  int* y = &x;
  *y = 7;
  cout << x << endl;
```

```
}
// 7

int main() {
  int a = 43;
  int b = 71;
  int c = 89;
  cout << *(&b - 1) << ' ' << *(&b + 1);
}
// 43 89

void f() {
  int a = 43;
  int b = 71;
}

int main() {
  int c = 89;
  for (int i = 0; i < 20; ++i){
    f();
    int x = *(&c - i);
    cout << i << ' ' << x << endl;
  }
}
// 0 89
// ...
// 14 43
// 15 71
// ...
```

Программа выводит 20 строк. В строках 14 и 15 записаны числа 43 и 71. Они совпадают с переменными `a` и `b` нашей программы. Если мы возьмем, например, `a = 434` и `b = 711`, то в строках 14 и 15 окажутся уже 434 и 711 соответственно.

В функции `main` мы запустили функцию `f`. Она запустилась, выделила на стеке фрейм и разместила в нем свои переменные. Мы не знаем точно, где на стеке эти переменные лежат, пробуем разные смещения относительно своей переменной `c`, пытаемся найти стековый фрейм функции `f`. Мы установили, что в четырнадцати `int`'ах от переменной `c` лежит переменная `a`. Мы убедились, что после завершения функции с её стековым фреймом ничего не происходит.

```
int main() {
```

```
  int c = 89;
  int* d = &c;
  int* e = d + 1;
  cout << d << endl
       << e << endl;
}
// 0x62fe3c
// 0x62fe40
```

В консоль вывелись два шестнадцатеричных числа. Если мы вычтем одно из другого, то получим 4. Если запустить программу с

```
int* e = d + 3;
```

то получаем 12. Этот пример иллюстрирует то, что при прибавлению к указателю числа целочисленное значение, которое хранится в указателе, изменяется на число, умноженное на размер типа, на который указывает наш указатель.

Перепишем оператор `[]`, используя менее громоздкий синтаксис.

```
T& operator[] (size_t index) {
  return data[index];
}
```

3.1.8. Добавляем в вектор `begin` и `end`

Добавим в наш вектор методы `begin` и `end`, чтобы по нему можно было итерироваться в цикле. Цикл `range-based for` разворачивается в следующую конструкцию:

```
SimpleVector<int> v(5);
for (auto i = v.begin(); i != v.end(); ++i) {

}
```

Требования к итераторам в `range-based for`:

- `begin()` указывает на первый элемент;

- `end()` указывает на последний элемент;
- увеличение итератора на один переводит его на следующий элемент.

Всем этим требованиям удовлетворяют указатели на элементы нашего вектора. В качестве `begin` можно использовать указатель на первый элемент нашего вектора, в качестве `end` – указатель на последний элемент. Соответственно, в качестве типа возвращаемого значения для операторов `begin()` и `end()` можем использовать указатель на элемент типа `T`.

В результате класс `SimpleVector` будет выглядеть следующим образом.

```
class SimpleVector {
public:

  explicit SimpleVector (size_t size) {
    data = new T[size];
    end_ = data + size;
  }

  ~SimpleVector() {
    delete[] data;
  }

  T& operator[] (size_t index) {
    return data[index];
  }

  T* begin() { return data; }
  T* end()   { return end_; }

private:
  T* data;
  T* end_;
};
```

Теперь напишем функцию `print` для нашего вектора, которая будет печатать его на консоль.

```
template <typename T>
void Print(const SimpleVector<T>& v) {
  for (const auto& x : v) {
    cout << x << ' ';
```

```
  }
}
```

Такая функция работать не будет из-за проблем с константностью. Вектор `v` передается по константной ссылке, поэтому и вызывать мы можем только константные методы объекта `v`. Методы `begin` и `end` константными не являются. Сделаем их константными:

```
T* begin() const { return data; }
T* end()   const { return end_; }
```

В текущей функции `Print` мы можем случайно поменять наш вектор, например, прописав строку

```
*i = 42;
```

Теперь вызвав функцию `Print` два раза, мы увидим на экране:

```
// 5 3 4 -1
// 42 42 42 42
```

Чтобы избежать этой проблемы, в методах begin() и end() нам нужно возвращать указатели на константы:

```
const T* begin() const { return data; }
const T* end()   const { return end_; }
```

Теперь компилятор будет запрещать изменять вектор, принимаемый по константной сслыке.

Есть еще одна проблема. Теперь мы не можем через возвращаемые итераторы менять наш вектор, например, его сортировать. Нам нужно завести две пары `begin` и `end`: константную и неконстантную.

```
T* begin() { return data; }
T* end()   { return end_; }

const T* begin() const { return data; }
const T* end()   const { return end_; }
```

3.1.9. Константный указатель и указатель на константу

У константности указателей есть двойственность.

С одной стороны у нас есть указатель, с другой есть объект, на который он указывает. Соответственно мы можем менять сам указатель, а можем этот объект. Когда переменная объявлена как `T* ptr`, то это просто указатель и мы можем менять как объект, так и сам указатель.

Если же перед типом указателя мы пишем ключевое слово `const`, то получаем указатель на константу: мы можем изменять сам этот указатель, но объект мы менять не можем.

Если мы хотим менять объект, но не хотим менять сам указатель, мы можем объявить константный указатель.

Ключевое слово `const` у него идет после `*`. Тогда мы можем менять объект, на который указывает указатель, но не можем менять сам указатель. Если же мы хотим защититься от любых изменений данных, мы можем объявить константный указатель на константу:

3.1.10. Итоги раздела

Сравнение стека и кучи

	Стек	Куча
Создание объекта	`string s;`	`string* s = new string;`
Уничтожение объекта	Автоматически при выходе из области видимости	Вручную: `delete s;`
Применяется	Для локальных переменных	Для объектов, которые живут дольше, чем создавшая их функция

Недостатки использования `new/delete`

- можно забыть вызывать `delete`;
- можно перепутать `delete` и `delete[]`.

Всё, что связано с операторами `new`, `delete` и арифметикой указателей было рассказано, чтобы потом рассказать про внутреннее устройство контейнеров языка C++. В современном языке C++ нет ни одной причины применять `new` и `delete` в прикладном коде.