Лекция 3. “Универсальная” сортировка (указатели на функции). Три вида памяти: стек.

Евгений Линский

“Универсальная” сортировка.

Задача: написать сортировку работающую с любым типом данных (int, float, product_t). Подзадачи:

1. передать в функцию “любой тип данных”
2. переставлять “любые” элементы (swap)
3. сравнивать “любые” элементы (cmp)

Любой тип данных: void*.

```
1  void qsort(void* array, size_t n, ...);
2
3  product_t array1[100];
4  int arrray2[20];
5  qsort(array1, 100, ...);
6  qsort(array2, 20, ...)
```

- void* — не работает адресная арифметика (почему?)
- В C++ неявное приведение типа в void* не вызывает ошибки (в C все можно неявно)
- В C++ требуется явное приведение типа из void* (в C все можно неявно)
- void* malloc(...)

Переставлять любые элементы: swap.

```
1  void qsort(void* array, size_t n, size_t elem_size, ...) {
2    char* p = array;
3    //смысл: swap(&array[i], &array[j])
4    swap(p + i * elem_size, p + j * elem_size, elem_size);
5  }
6  void swap(char *start, char *end, size_t elem_size) {
7    int i = 0;
8    while( i < elem_size ) {
9      ... //сами
10   }
11 }
```

Q: почему у параметра swap тип char*?

Сравнивать любые элементы: cmp.

Указатель — адрес в памяти:

- В памяти хранятся переменные и двоичный код (двоичные инструкции нашей программы)
- Можно хранить адрес переменной (указатель)
- Можно хранить адрес кода (указатель на функцию)

```
1  void dummy(int x) {
2      printf( "%d\n", x );
3  }
4
5  int main() {
6    void (*func)(int);
7    func = &dummy;
8    (*func)(2);
9    // можно и так:
10   func = dummy;
11   func(2);
12   return 0;
13 }
```

Сравнивать любые элементы: cmp.

```
1  void qsort(void* array, size_t size, size_t elem_size,
2             int (*cmp)(void* p1, void* p2)) {
3             ...
4  }
5
6  int cmp_int(void* p1, void* p2) {
7    int* pi1 = p1; int* pi2 = p2;
8    return (*pi1 - *pi2);
9  }
10 int cmp_product_by_weight(void* p1, void* p2) {
11   product_t* pp1 = p1; product_t* pp2 = p2;
12   return ((*pp1).weight - (*pp2).weight);
13 }
14 product_t array1[100];
15 int arrray2[20];
16 qsort(array1, 100, sizeof(array1[0]),
17    cmp_product_by_weight);
18 qsort(array2, 20, sizeof(array2[0]), cmp_int);
```

Три вида памяти в программе на C

1. **Стек** (stack)
   - локальные переменные функций, параметры функций
   - код для выделения и освобождения генерирует компилятор
   - выделяется при “входе” в функцию, освобождается при “выходе” из функции
2. **Глобальная память** (static variables)
   - глобальные переменные (вне функций), статические переменные (static)
   - код для выделения и освобождения генерирует компилятор
   - выделяется при загрузке в память, освобождается при завершении программы
   - глобальные инициализируются в *каком-то* порядке, статические — при входе в функцию
3. **Куча** (heap)
   - код для выделения и освобождения пишет программист

Карта памяти

- Расположение частей может отличаться на разных платформах.
- В общем случае адресация неважна.
- Ниже один из вариантов (“упрощенный linux”, 4 Gb)

OS kernel (например, 1 Gb) ....
Stack, растет вниз ↓(например, 10 Mb)
.... Heap, растет вверх ↑(например, ∼2,9 Gb)
Static variables (например, 10 Mb)
Двоичный код программы (например, 10 Mb)

Стек

```
1  int sum(int a, int b) {
2    int s = a + b;
3    return s;
4  }
5  int main() {
6    int c = 1; int d = 2;
7    int e = sum(c, d);
8    return 0;
9  }
```

Stack, растет вниз ↓(например, 10 Mb)

Кадр main	c, d, e, RV, RA
Кадр sum	a, b, s, RV, RA

Вошли в функцию — выделили кадр (frame), вышли из функции — освободили кадр

- RV — return value (возвращаемое значение)
- RA — return address (на какой адрес вернуться в main после окончания sum)

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

1. Передача копии параметра: main –> store (сохранить на стек), sum –> load (загрузить со стека)

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

1. Передача копии параметра: main –> store (сохранить на стек), sum –> load (загрузить со стека)
2. Зарезервировать несколько регистров под передачу параметров и RV и передавать через них (т.е. не выгружать в память)

Стек. Вычислительная сложность.

Stack, растет вниз ↓(например, 10 Mb)

Кадр main	c, d, e, RV, RA
Кадр sum	a, b, s, RV, RA

Вошли в функцию — выделили кадр (frame), вышли из функции — освободили кадр

- В процессоре есть регистр (например, sp), который хранит адрес “головы” стека
- Выделение кадра — уменьшение регистра на размер кадра, освобождение — увеличение (быстрые операции)
- Код, который расчитывает размер кадра и меняет sp, генерирует компилятор
- Выделение локальной переменной — это очень быстро, происходит один раз на функцию

Стек. Рекурсия.

```
1  int factorial(int n) {
2    if (n == 1)
3      return 1;
4    else
5      return factorial(n-1) * n;
6  }
```

Почему никто не любит такое?

Стек. Рекурсия.

```
1  int factorial(int n) {
2    if (n == 1)
3      return 1;
4    else
5      return factorial(n-1) * n;
6  }
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу

Стек. Рекурсия.

```
1  int factorial(int n) {
2    if (n == 1)
3      return 1;
4    else
5      return factorial(n-1) * n;
6  }
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу
2. Можно переписать циклом без потери элегантности

Стек. Рекурсия.

```
1  int factorial(int n) {
2    if (n == 1)
3      return 1;
4    else
5      return factorial(n-1) * n;
6  }
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу
2. Можно переписать циклом без потери элегантности

Иногда компилятор может сам соптимизировать в цикл (хвостовая рекурсия).