

Лекция

Умные указатели

Умные указатели

Пример «самодельного» умного указателя.

```
1.  using Type = Foo;
2.  class SP {
3.  private:
4.      Type* pointer{};
5.  public:
6.      SP(Type* p) : pointer(p) {}
7.      operator Type*() { return pointer; }
8.      Type* operator->() { return pointer; }
9.  };

10. void f(Foo*);
11. SP pf(new Foo);
12. f(pf);
13. pf->MemberOfFoo();
```

Умные указатели

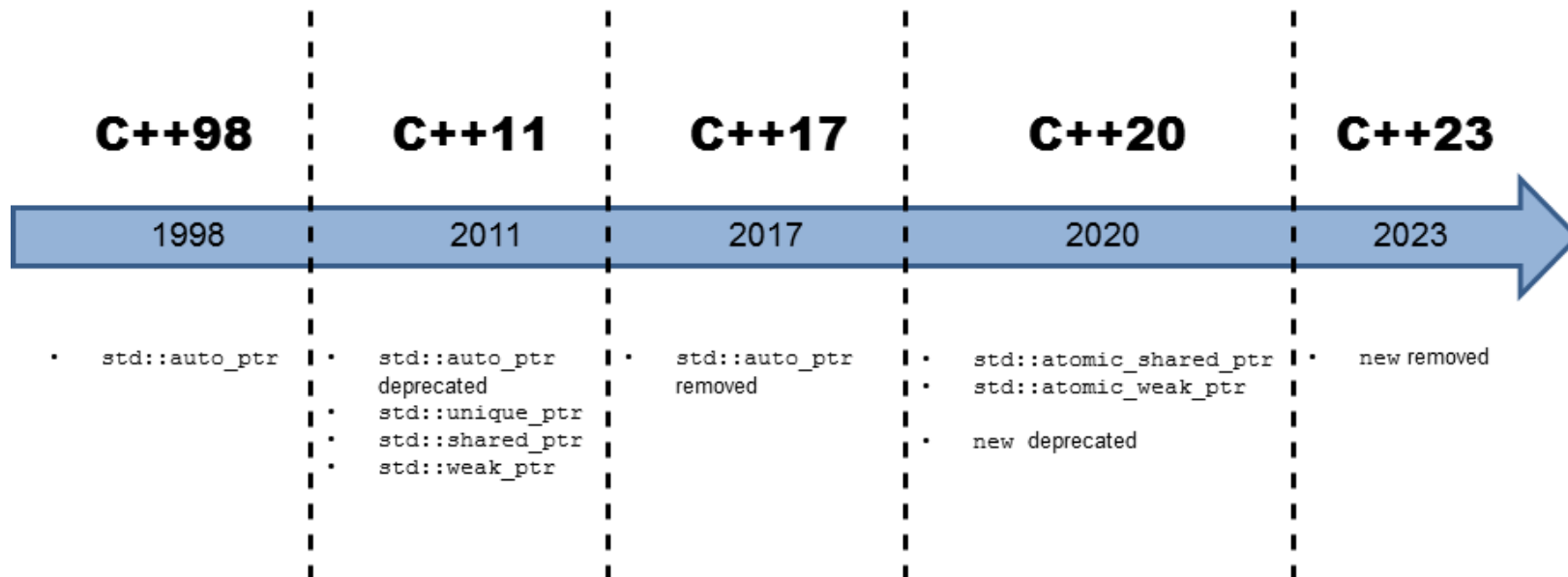
Пример «самодельного» умного указателя

```
1.  template <class Type>
2.  class SP {
3.  private:
4.      Type* pointer{};
5.  public:
6.      SP(Type* p) : pointer(p) {}
7.      operator Type*() { return pointer; }
8.      Type* operator->() { return pointer; }
9.  };

10. void f(Foo*);
11. SP<Foo> pf(new Foo);
12. f(pf);
13. pf->MemberOfFoo();
```

ISO C++ слёт в Джексонвилле

- Комитет по стандартизации решил, что указатели будут запрещены (deprecated) к C++20 и с большой вероятностью будут удалены из стандарта C++23. Согласно новости с сайта Fluent C++: [C++ Will no Longer Have Pointers](#).



```

#include <chrono>
#include <iostream>
static const long long numInt= 100000000;
int main() {
    auto start = std::chrono::system_clock::now();
    for (long long i=0; i < numInt; ++i) {
        int* tmp(new int(i)); delete tmp;
        // std::shared_ptr<int> tmp(new int(i));
        // std::shared_ptr<int> tmp(std::make_shared<int>(i));
        // std::unique_ptr<int> tmp(new int(i));
        // std::unique_ptr<int> tmp(std::make_unique<int>(i));
    }
    auto d = std::chrono::system_clock::now() - start;
    std::cout << "time native: " << d.count() << "sec." << std::endl;
}

```

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Ownership + Memory overhead

C++ reference – не владеет, обеспечивает доступ к чужому ресурсу, ресурс *НЕ может* быть пустым, *НЕ может* поменять ресурс

C raw pointer – не владеет, обеспечивает доступ к чужому ресурсу, ресурс *может быть* пустым и заменён другим

std::unique_ptr – эксклюзивный владелец ресурса, очищает ресурс при удалении указателя, сохраняет все свойства raw указателя

- По-умолчанию не имеет дополнительной памяти
- Можно параметризовать специальной функцией очищения.

std::shared_ptr – владеет ресурсом, и может поделиться с подобными указателями, может захватывать эксклюзивное владение, сохраняет все свойства raw указателя.

- Содержит **счётчик ссылок**. Если происходит копирование, счётчик ссылок на ресурс увеличивается.

std::weak_ptr – захватывает ресурс во временное владение в момент существования ресурса, сохраняет все свойства raw указателя.

- содержит дополнительный счётчик ссылок

std::unique_ptr

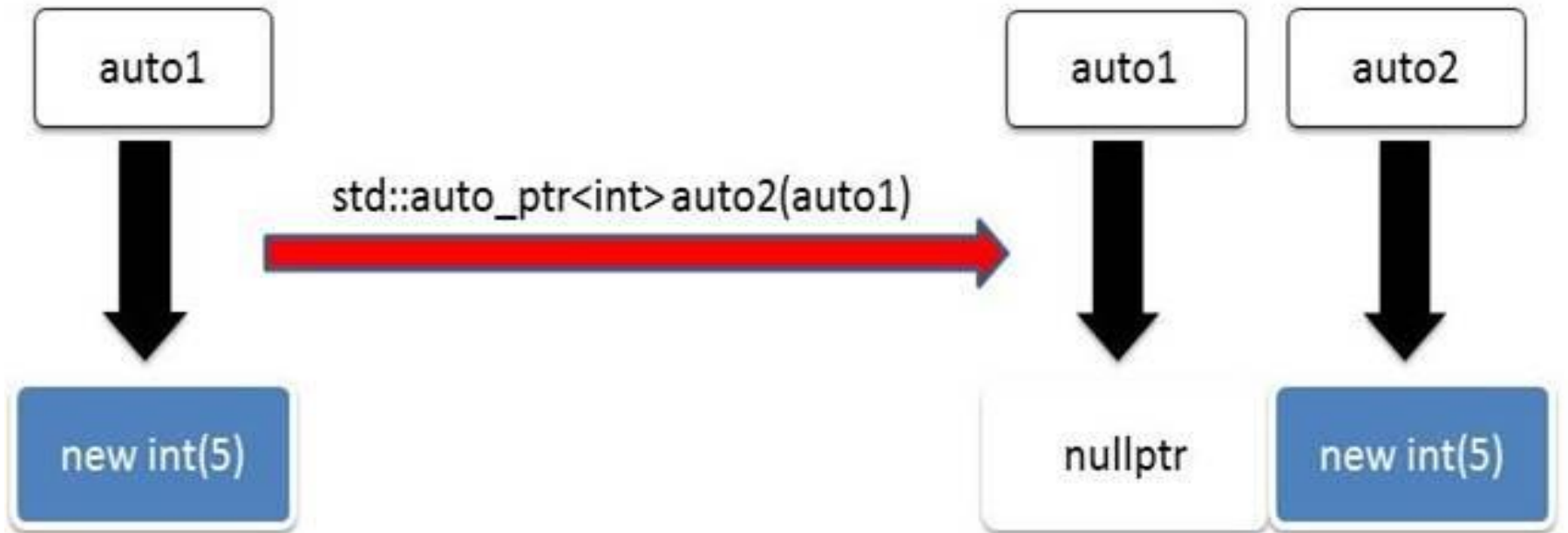
Невозможность скопировать std::unique_ptr

```
1. #include <memory>
2. int func()
3. {
4.     std::unique_ptr<CFoo> PFoo1(new CFoo());
5.     std::unique_ptr<CFoo> PFoo2;
6.     PFoo2 = PFoo1; // Ошибка компиляции
7. }
```

std::unique_ptr

Перемещение std::unique_ptr

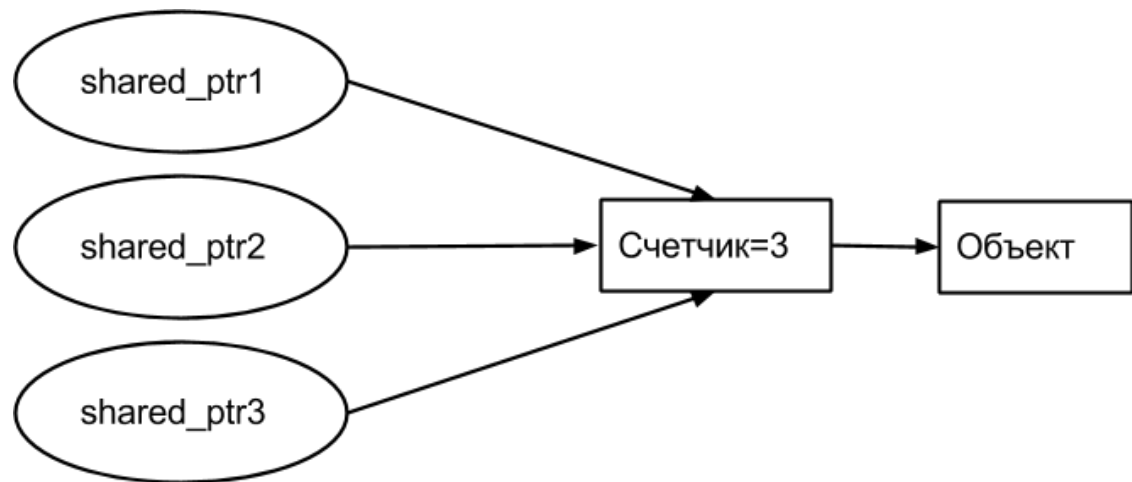
```
1. #include <memory>
2. int func()
3. {
4.     std::unique_ptr<CFoo> PFoo1(new CFoo());
5.     std::unique_ptr<CFoo> PFoo2;
6.     PFoo2 = std::move(PFoo1);
7. }
```

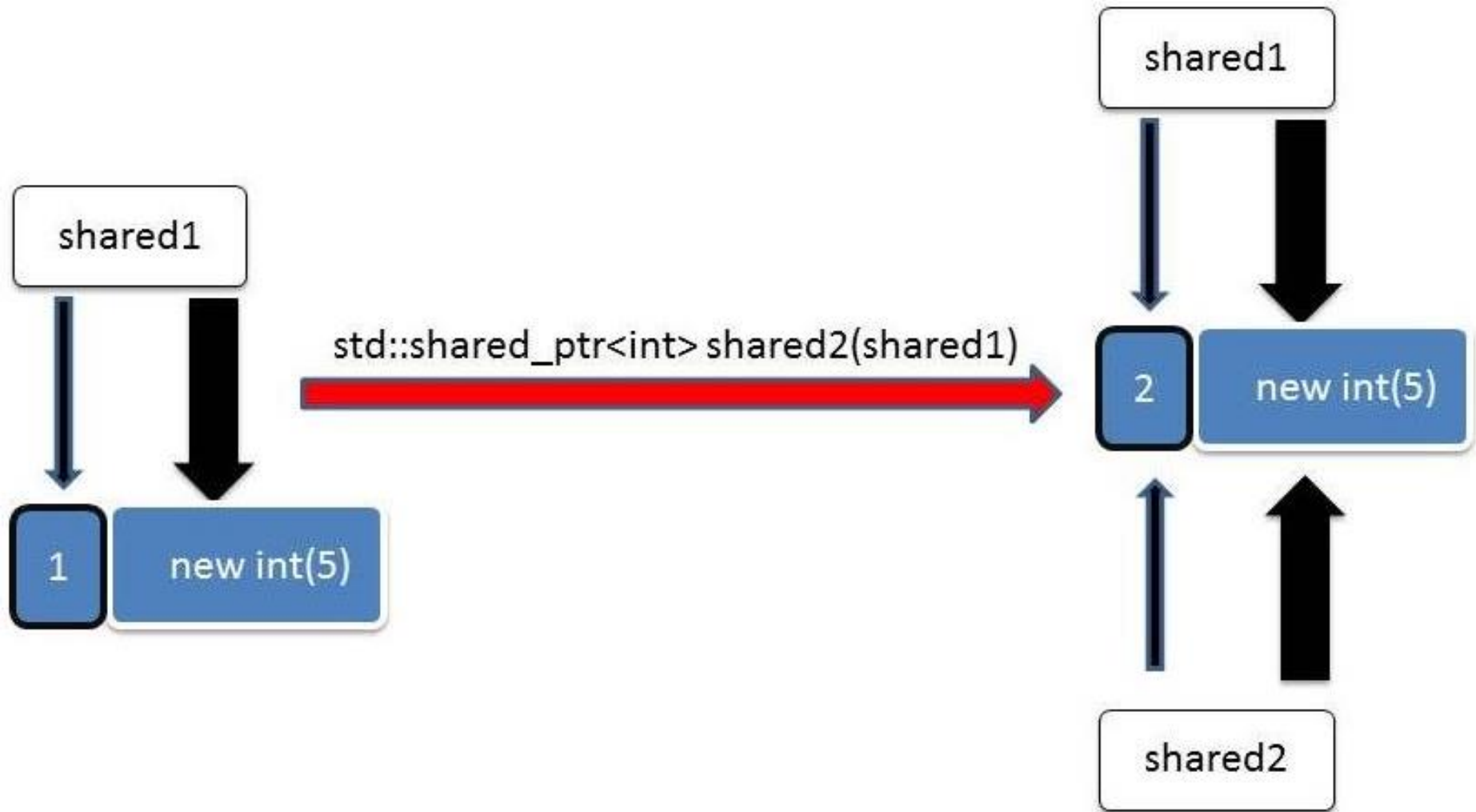


std::shared_ptr

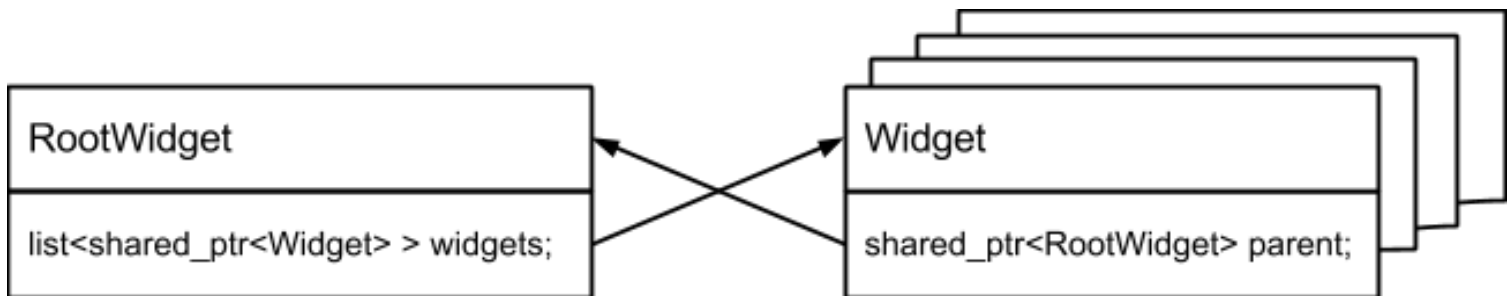
Пример использования std::shared_ptr

```
1. #include <memory>
2. int func()
3. {
4.     std::shared_ptr<CFoo> PFoo1(new CFoo());
5.     std::shared_ptr<CFoo> PFoo2;
6.     PFoo2 = PFoo1;
7. }
```

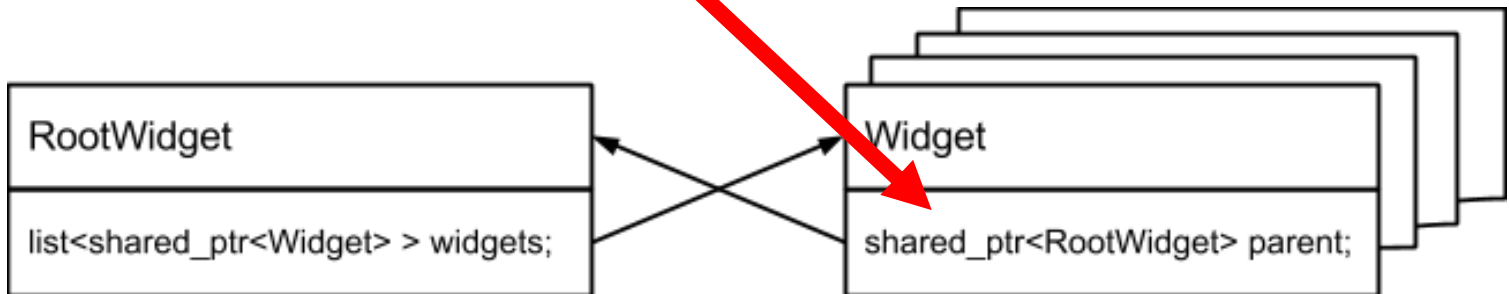




std::shared_ptr



std::weak_ptr



std::unique_ptr

Incomplete types and std::unique_ptr

```
1. #include <memory>

2. class FooImpl;
3. struct FooImplDeleter { void operator()(FooImpl *p); }

4. class Foo {
5.     //...
6. private:
7.     std::unique_ptr<FooImpl, FooImplDeleter> impl_;
8. };

9. // Foo.cpp
10. // ...
11. void FooImplDeleter::operator()(FooImpl *p) {
12.     delete p;
13. }
```

std::thread

Start threads

```
1. #include <thread>
2. void tFunc1() {
3.     // do smth ..
4. }
5. void tFunc2(int i, double d, const std::string &s) {
6.     std::cout << i << ", " << d << ", " << s << std::endl;
7. }

8. int main(int argc, char **argv) {
9.     int k = 0;
10.    std::thread th1(tFunc1);
11.    std::thread th2(tFunc2 , 1, 2.34, "example");
12.    std::thread th3([argc](int &k){
13.        std::cout << k << ", " << argc << std::endl;
14.    }, std::ref(k));
15.    th1.join();
16.    th2.join();
17.    th3.detach();
18.    return 0;
19.}
```

std::shared_ptr на *this из класса

Caution!

```
1. template<class T>
2. class enable_shared_from_this {
3.     weak_ptr<T> weak_this_;
4. public:
5.     shared_ptr<T> shared_from_this() {
6.         // Преобразование слабой ссылки в сильную
7.         // через конструктор shared_ptr
8.         shared_ptr<T> p( weak_this_ );
9.         return p;
10.    };

11. class Widget: public enable_shared_from_this<Widget>
12. {};
```

Multi-threading library (C++11)

- **std::shared_ptr** - «распределённый» по определению, но **НЕ thread-safe**.
- При модификации указателя из разных потоков, нужны Lock-и, для доступа на чтение они не требуются.
- Обновление счётчика ссылок – **атомарная операция**, можно гарантировать, что ресурс будет удалён единожды.
- C++20 atomic smart pointers:
 - **std::atomic_shared_ptr**
 - **std::atomic_weak_ptr**

Обёртка запуска worker'а с любыми аргументами функции

```
Template <typename Function, typename... Args>
void WorkerThreadRunTask(SomeData &data, Function task,
                        Args... args) {
    try {
        std::shared_ptr<std::thread> pWorkerThread{};
        pWorkerThread = std::make_shared<std::thread>(task, args...));
    } catch (const std::exception &e) {
        std::cerr << e.what() << std::endl;
    }
}
```

```

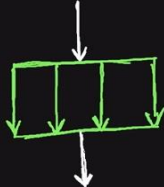
export OMP_NUM_THREADS=4;
#include <iostream>
int main(int argc, char** argv)
{

```

```

    omp_set_num_threads(4)
    #pragma omp parallel num_threads(4)
    {
        std::cout << "Hello, world!" << std::endl;
    }
    return 0;
}

```



```

#pragma omp parallel
{

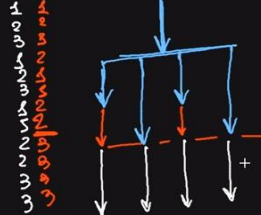
```

```

    std::cout << 1 << std::endl;
    std::cout << 2 << std::endl;
    #pragma omp barrier
    std::cout << 3 << std::endl;
}

```

3



```

#pragma omp parallel shared(A,B,C) private(i,n)

```

```

#pragma omp parallel for reduction(+:sum)

```

```

for(int i=0; i<1e7; i++) {
    sum = sum + f(i);
}

```



```

#pragma omp parallel {
    #pragma omp for
}

```

3

```

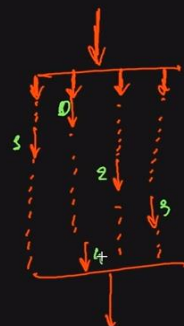
#pragma omp parallel for ordered
for(int i=0; i<5e7; i++)
{

```

```

    std::cout << "Cycle!" << std::endl;
    #pragma omp ordered
    {
        std::cout << i << std::endl;
    }
}

```



```

#pragma omp parallel
{
    #pragma omp atomic
    n++;
}

```

```

#pragma omp task
{
    #pragma omp taskwait
}

```

```

#pragma omp parallel num_threads(4)
{

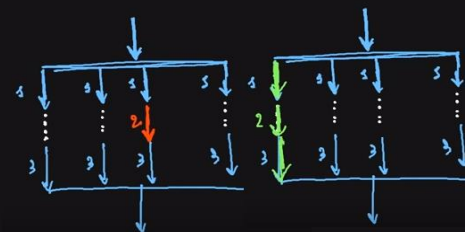
```

```

    std::cout << 1 << std::endl;
    #pragma omp single master nowait
    {
        std::cout << 2 << std::endl;
    }
    // Tim Sonep
    std::cout << 3 << std::endl;
}

```

3



```

#pragma omp parallel
{
    #pragma omp critical
    {
        //
    }
}

```

```

#pragma omp sections
{

```

```

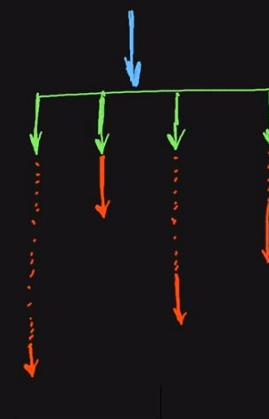
    #pragma omp section
    {
        //
    }

```

```

    #pragma omp section
    {
        //
    }
    //
}

```



Куча (Heap)

Часть динамической памяти процесса, предназначена для выделения участков памяти произвольного размера в рамках его адресного пространства.

Работа с коллекциями и структурами неизвестной (во время компиляции) длины и размера.

Подсистема выделения памяти - memory allocator

Аллокатор

Основные требования к аллокатору памяти:

- Оптимальное используемое пространство
- Исключение фрагментации
- Минимальное время работы
- Обеспечение локальности памяти
- Настраиваемость
- Совместимость со стандартами
- Переносимость
- Обнаружение наибольшего числа ошибок

Сборка мусора:

- подсчет ссылок
- трассировка/с выставлением флагов (Mark/Sweep)

Учёт свободных участков

- **битовая карта (bitmap)** — все блоки памяти одного размера, каждому блоку памяти ставится в соответствие 1 бит = занят/свободен
- **связный список** — каждому непрерывному блоку памяти ставится в соответствие запись в связном списке блоков, в которой указывается начало, размер участка, занят/свободен
- **несколько связных списков** для участков разных типов и предназначений (Buddy allocation algorithm)

csmalloc

Делаем утечки.

```
1. void Leak(char *inStr)
2. {
3.     char *str = (char*)malloc(strlen(inStr));
4.     memcpy(str, inStr, strlen(inStr));
5. }

6. char* AvoidLeak(char *inStr)
7. {
8.     char *str = (char*)malloc(strlen(inStr));
9.     memcpy(str, inStr, strlen(inStr));
10.    return str;
11. }
```

сcalloc

Функция main с утечками

```
1. int main()
2. {
3.     char *str;

4.     Leak("This leaks 19 bytes");
5.     str = AvoidLeak("This is not a 26 byte leak");
6.     free(str);
7.     str = AvoidLeak("12 byte leak");
8.     exit(0);
9. }
```

ссмаллок

Результат

```
1.  * 61.3% = 19 Bytes of garbage allocated in 1 allocation
2.  |           |           0x40047306 in <???>
3.  |           |           0x080493eb in <main>
4.  |           |           at test1.c:20
5.  |           |           0x0804935c in <Leak>
6.  |           |           at test1.c:5
7.  |           `-----> 0x08052fb7 in <malloc>
8.  |                           at src/wrapper.c:318
9.  |
10. * 38.7% = 12 Bytes of garbage allocated in 1 allocation
11. |           |           0x40047306 in <???>
12. |           |           0x0804941e in <main>
13. |           |           at test1.c:23
14. |           |           0x080493a4 in <AvoidLeak>
15. |           |           at test1.c:11
16. |           `-----> 0x08052fb7 in <malloc>
17. |                           at src/wrapper.c:318
18. `-----
```

dmalloc

Результат dmalloc

1. not freed: '0x45008' (12 bytes) from 'ra=0x1f8f4'
2. not freed: '0x45028' (12 bytes) from 'unknown'
3. not freed: '0x45048' (10 bytes) from 'argv.c:1077'
4. known memory not freed: 1 pointer, 10 bytes
5. unknown memory not freed: 2 pointers, 24 bytes

tcmalloc

- Работает быстрее, чем malloc из glibc
LD_PRELOAD="/usr/lib/libtcmalloc.so"

СПАСИБО ЗА ВНИМАНИЕ