# Лекция 2.
# Как бороться с утечками ресурсов?

CS Club Novosibirsk, 2019

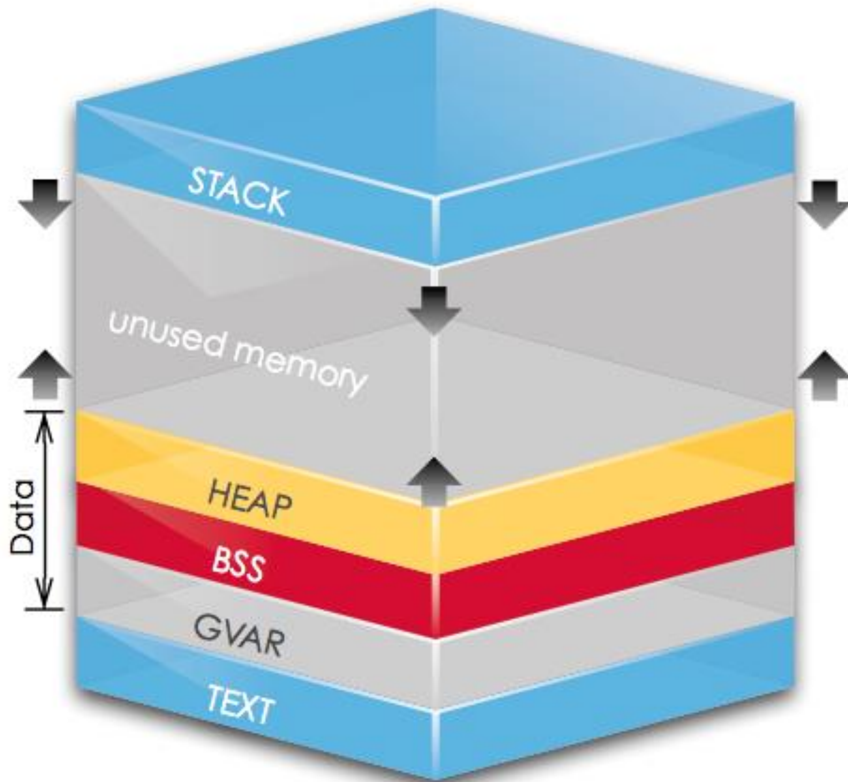
# Часть 1. Память

# Процесс и потоки

- Процесс – ресурсы:
  - адресное пространство (память)
  - объекты ядра (файловые дескрипторы, объекты синхронизации, сокеты, …)

- Поток – выполнение инструкций
  - последовательность команд
  - стек
  - thread local storage (TLS)
  - используют общие ресурсы процесса

# Устройство памяти процесса

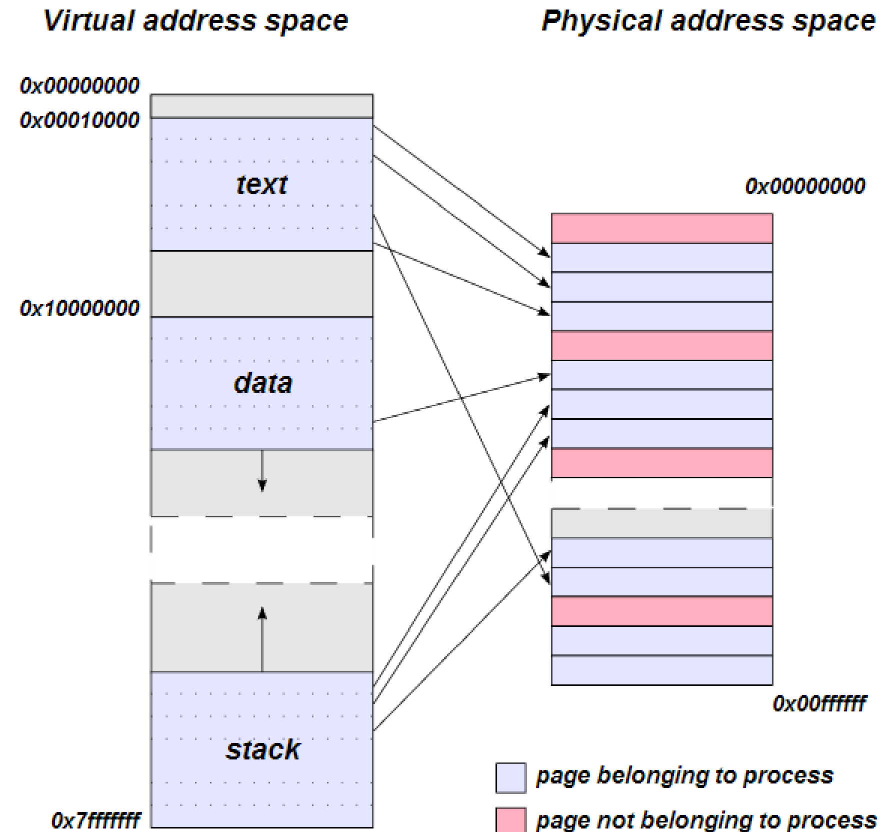



- from http://www.sw-at.com

- Сегмент кода (text)
- Сегмент данных:
  - Глобальные переменные
  - BSS (глобальные переменные без инициализации)
  - Heap (может быть не один)
- Сегмент стека
  - стек может быть не один

# Страничная память

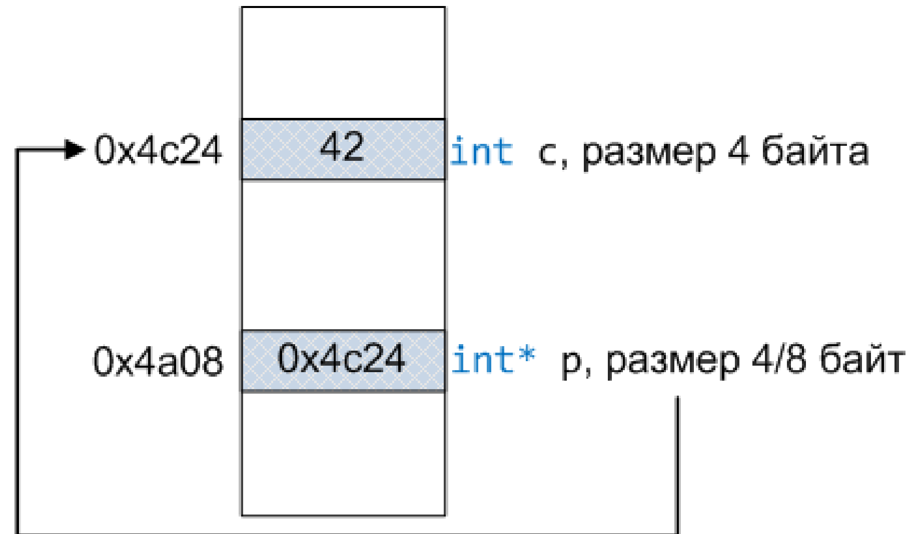
- Задачи:
  - избежать фрагментацию
  - изоляция процессов
  - страницы только для чтения и неисполняемые
  - свопинг
  - отображение в память файлов
  - разделяемая процессами общая память




from http://en.wikipedia.org/wiki/Page_table

# Указатели

- Обычная переменная
  - Размер: машинное слово
  - Значение: адрес другой переменной




```cpp
1. int c = 42;
2. int* p = &c;
```

# Разыменование. Взятие адреса

- Взятие адреса:

```
1. int  c = 42;
2. int *p = &c;
3.
4. std::cout << &c;
```

```
1. 0x0038f7d8
```

- Разыменование:

```
1. int  c = 10;
2. int *p = &c;
3.
4. *p = 5;
5. std::cout << *p << " " << c << endl;
```

```
1. 5 5
```

# Нулевой указатель и nullptr

- Гарантируется, что нет объектов с нулевым адресом – используем как указатель, который не ссылается на объект

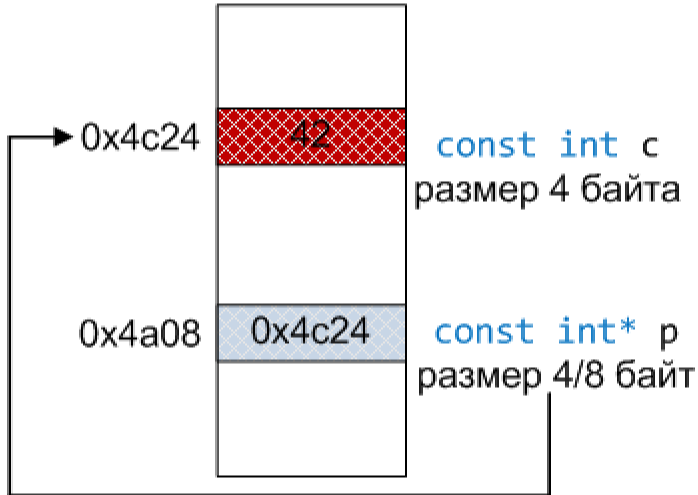
```
1.  void make(int value)   { cout << "int value"; }
2.  void make(char* object){ cout << "char* object"; }
3.  
4.  int main()
5.  {
6.      char* uno = 0;
7.  
8.      const int NULL = 0;
9.      char* due = NULL;
10. 
11.     char* tre = nullptr;
12. 
13.     make(0);
14.     make(nullptr);
15. 
16.     return 0;
17. };
```

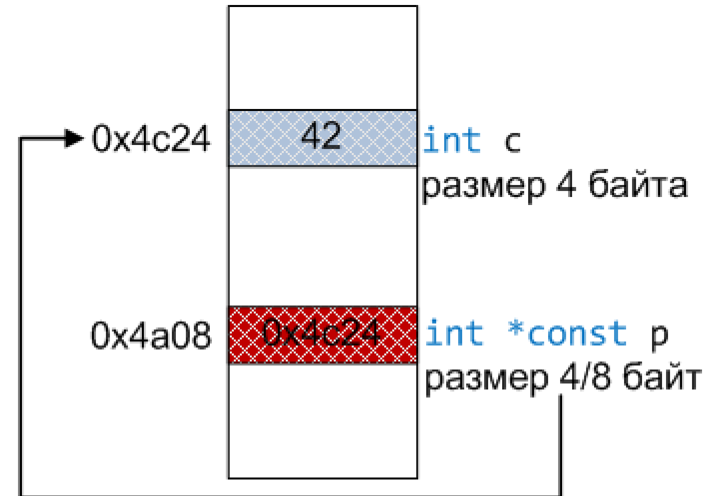
# Константы

- Используйте вместо magic numbers
- Часто оптимизируются на этапе компиляции

```
1.  const double pi = 3.14;
2.  const double e; // error: must be initialized if not extern
3.  const double coef[3] = {1, 2, 1};
4.
5.  // just C++ type, not memory allocation type
6.  int value = 5;
7.  const int* pvalue = &value;
8.
9.  value   = 7; // good
10. *pvalue = 9; // error
11.
12. // no optimization
13. extern const int answer;
14. const double* ppi = &pi;
```

# Константные указатели




```
1.
2.  const int c = 0;
3.  const int *p = &c;
4.  int const *q = &c;
5.
6.  *p = 5; // error
7.   p = 0; // ok
```

```
1.  int c = 5;
2.  int *const p = &c;
3.
4.  *p = 5; // ok
5.   p = 0; // error
6.
7.  int const *const full = &c;
```

# Выделение памяти

- Выделение/освобождение памяти в heap: операторы `new/delete`
- При нехватке памяти генерируется исключение `std::bad_alloc`

```
1.
2.  try
3.  {
4.      int* p   = new int (42);
5.      int* arr = new int [get_count()];
6.
7.      delete p;
8.      delete [] arr;
9.  }
10. catch(std::bad_alloc const&)
11. {
12.     // ...
13. }
```

# new & delete *

- Placement new:

```
1. void* p = …;
2. T* pt = new (p) T(…);
3. pt->~T();
```

- Переопределение операторов:

```
1. void* operator new   (size_t );
2. void  operator delete(void* p);
3.
4. void* operator new   [] (size_t );
5. void  operator delete[] (void *p);
```

# Утечки памяти (memory leaks)*

- Windows (debug runtime):

```
1.  int main()
2.  {
3.      _CrtSetDbgFlag(
4.          _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG) |
5.          CRTDBG_LEAK_CHECK_DF);
6.
7.      // ...
8.  }
```

- Unix, linux: valgrind (with debug symbols)

```
1.  % valgrind --tool=memcheck --leak-check=yes my_test
2.
3.  ==5015== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
4.  ==5015== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
5.  ==5015== by 0x804840F: main (in /home/cpp/my_test.cpp:15)
```

# Часть 2. Как бороться с утечками?

# Пример утечки ресурсов

- Есть ли проблемы в этом коде?

```
1.  void save_to(char* filename, document const& doc)
2.  {
3.      // allocates memory and concatenates with folder prefix
4.      char* path = to_sys_path(filename);
5.
6.      FILE* f = fopen(path, "bw+");
7.      if (f == 0)
8.          return;
9.
10.     // allocates memory and construct contiguous buffer
11.     size_t size = doc.size();
12.     if (fwrite(doc.data(), 1, doc.size(), f) == size)
13.         doc.set_modified(false);
14.
15.     delete [] path;
16.     fclose(f);
17. }
```

# Пример утечки ресурсов

- А что, если `set_modified` бросает исключения?

```
1.  void save_to(char* filename, document const& doc)
2.  {
3.      // allocates memory and concatenates with folder prefix
4.      char* path = to_sys_path(filename);
5.  
6.      FILE* f = fopen(path, "bw+");
7.      if (f == 0)
8.          return;
9.  
10.     // allocates memory and construct continuous buffer
11.     size_t size = doc.size();
12.     if (fwrite(doc.data(), 1, doc.size(), f) == size)
13.         doc.set_modified(false);
14. 
15.     delete [] path;
16.     fclose(f);
17. }
```

# RAII, определение

- Resource Acquisition Is Initialization
- Симметричная работа с ресурсом:
  - Получаем ресурс (или создаем) в конструкторе
  - Освобождаем его в деструкторе
- Когда локальный объект выходит из области действия (своего скоупа), вызывается деструктор – освобождается связанный с ним ресурс.
- Объекты удаляются в порядке обратном их созданию – удобно для удаления зависимых объектов.

# Пример RAII класса

```
1.  struct out_bin_file
2.  {
3.      out_bin_file(const char* name)
4.          : file_(fopen(name, "bw+")){}
5.
6.     ~out_bin_file(){ fclose(file_); }
7.
8.  private:
9.      out_bin_file(out_bin_file const&);
10.     out_bin_file& operator=(out_bin_file const&);
11.
12. private:
13.     FILE* file_;
14. };
15.
16. void foo()
17. {
18.     // this file will be always closed
19.     // before returning from this function
20.     out_bin_file file("~/some.txt");
21.
22.     if (...)
23.         return;
24.
25.     throw std::runtime_error("oops...");
26.     //.. some code
27. }
```

# Пример RAII класса

```
1.   struct out_bin_file
2.   {
3.       out_bin_file(const char* name)
4.           : file_(fopen(name, "bw+")){}
5.
6.      ~out_bin_file(){ fclose(file_); }
7.
8.       //.. from C++11
9.       out_bin_file(out_bin_file const&)            = delete;
10.      out_bin_file& operator=(out_bin_file const&) = delete;
11.
12.  private:
13.      FILE* file_;
14.  };
15.
16.  void foo()
17.  {
18.      // this file will be always closed
19.      // before returning from this function
20.      out_bin_file file("~/some.txt");
21.
22.      if (...)
23.          return;
24.
25.      throw std::runtime_error("oops...");
26.      //.. some code
27.  }
```

# Пример RAII класса

```
1.   struct out_bin_file
2.       : boost::noncopyable
3.   {
4.       out_bin_file(const char* name)
5.           : file_(fopen(name, "bw+")){}
6.
7.      ~out_bin_file(){ fclose(file_); }
8.
9.   private:
10.      FILE* file_;
11.  };
12.
13.  void foo()
14.  {
15.      // this file will be always closed
16.      // before returning from this function
17.      out_bin_file file("~/some.txt");
18.
19.      if (...)
20.          return;
21.
22.      throw std::runtime_error("oops...");
23.      //.. some code
24.  }
```

# Снова пример утечки ресурсов

```
1.  void save_to(char* filename, document const& doc)
2.  {
3.      // allocates memory and concatenates with folder prefix
4.      char* path = to_sys_path(filename);
5.  
6.      FILE* f = fopen(path, "bw+");
7.      if (f == 0)
8.          return;
9.  
10.     // allocates memory and construct continuous buffer
11.     size_t size = doc.size();
12.     if (fwrite(doc.data(), 1, doc.size(), f) == size)
13.         doc.set_modified(false);
14. 
15.     delete [] path;
16.     fclose(f);
17. }
```

# А теперь с использованием RAII

```
1. void save_to(string const& filename, document const& doc)
2. {
3.     ofstream ofs(to_sys_path(filename), ios_base::binary);
4. 
5.     if(ofs.write(doc.data(), doc.size()))
6.         doc.set_modified(false);
7. }
```

- Такой код не содержит утечек ресурсов
- Он стал компактнее, выразительнее и проще для понимания.

# Умные указатели

- Почти те же указатели, только слегка умнее
  - представляют собой RAII классы
  - часто поддерживают тот же интерфейс, что и обычные указатели: `op->`, `op*`, `op<` (например, чтобы положить в `std::set`)
  - сами управляют временем жизни объекта – вовремя вызывают деструкторы и освобождают память

# Польза умных указателей

- Автоматическое освобождение памяти при удалении самого указателя
- Безопасность исключений

```
1.  void foo1()
2.  {
3.      shared_ptr<my_class> ptr(new my_class("arg"));
4.      // or shorter definition:
5.      auto ptr = make_shared<my_class>("arg");
6.  
7.      ptr->bar(); // if throws exception, nothing bad happens
8.  }
9.  
10. void foo2()
11. {
12.     my_class* ptr = new my_class(/*...*/);
13.     ptr->bar(); // oops, troubles in case of exception
14.     delete ptr; // common trouble is to forget to delete
15. }
```

# Популярные умные указатели

- `boost::scoped_ptr` (уже нет)
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- `boost::intrusive_ptr`

- Deprecated: `std::auto_ptr` (заменен на `unique_ptr`)

# scoped_ptr

- Удобен для хранения указателя на стеке или полем класса. Не позволяет копироваться.

```
1.  template<class T> struct scoped_ptr : noncopyable
2.  {
3.  public:
4.      typedef T element_type;
5.  
6.      explicit scoped_ptr(T* p = 0);
7.      ~scoped_ptr();
8.  
9.      void reset(T* p = 0);
10. 
11.     T& operator *() const;
12.     T* operator->() const;
13.     T* get() const;
14. 
15.     explicit operator bool() const;
16. };
17. //-------
18. scoped_ptr<int> p(new int(5));
```

# Почему explicit конструктор?

```
1.  //what if scoped_ptr had NOT explicit constructor
2.  void foo(scoped_ptr<my_class> ptr)
3.  {
4.      /*...*/
5.  }
6.  
7.  auto p = new my_class(/*...*/);
8.  foo(p);      // epic fail, p is not valid after this call
9.  
10. p->do_smth();// error
11. delete p;    // one more error
```

# Возможности `scoped_ptr`

- Самый простой, быстрый
- Нельзя копировать и перемещать (move)
- Нельзя использовать в stl контейнерах
- Для массива: scoped_array
- При определении не требует полный тип, для инстанцировании – требует
- Нет особых причин использовать, если доступен std::unique_ptr

# Возможности `scoped_ptr`

- Самый простой, быстрый
- Нельзя копировать и перемещать (move)
- Нельзя использовать в stl контейнерах
- Для массива: scoped_array
- При определении не требует полный тип, для инстанцировании – требует
- Нет особых причин использовать, если доступен std::unique_ptr

# Требование полноты типа

```
1   #include <boost/scoped_ptr.hpp>
2
3   // b.h
4   struct A;
5   struct B
6   {
7       boost::scoped_ptr<A> a;
8       // some declarations
9       // but no explicit destructor
10  };
11
12  // main.cpp
13  #include "b.h"
14
15  int main()
16  {
17     B b;
18     return 0;
19  }
```

- Такой код приводит к ошибке компиляции

# checked_delete

```
1  // scoped_ptr.hpp
2  ~scoped_ptr() // never throws
3  {
4      boost::checked_delete(ptr);
5  }
6
7  // checked_delete.hpp
8  template<class T>
9  inline void checked_delete(T * x)
10 {
11     // intentionally complex - simplification causes regressions
12     typedef char type_must_be_complete[ sizeof(T)? 1: -1 ];
13     (void) sizeof(type_must_be_complete);
14     delete x;
15 }
```

- Либо стоит объявить полностью тип A в том же заголовочном файле после типа B
- Либо типу B необходимо добавить объявление деструктора (определение может быть в другом cpp-файле)

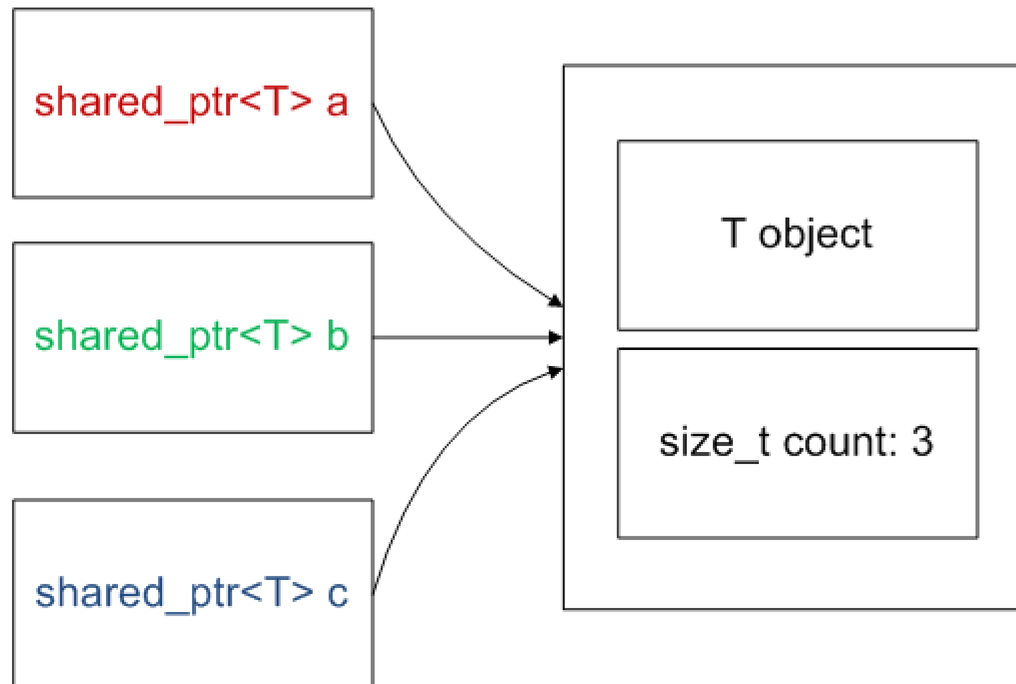
# std::unique_ptr

- Владеет объектом эксклюзивно
- Нельзя копировать, но можно перемещать
- Удобно использовать при возврате из функции
- Удобно использовать для *pimpl*
- Полный тип нужен лишь на момент удаления
- Удобно использовать для поля класса (но возможно, лучше `optional<T>`)
- Есть функция `T* release()`, отдает владение

```
1  template<class T, class Deleter = std::default_delete<T>>
2  class unique_ptr;
3
4  template<class T, class Deleter>
5  class unique_ptr<T[], Deleter>;
```

# shared_ptr

- Поддерживает общий счетчик ссылок на выделенный объект
- Удаляет объект только, когда последний из ссылающихся shared_ptr’ов удаляется или принимает указатель на другой объект



# shared_ptr

- Удобен для разделения владением
- Можно возвращать из функций
- (*)Можно передавать между модулями - запоминает правильную функцию удаления (из нужной библиотеки)

```
1.  template<class T> struct shared_ptr
2.  {
3.      /* more than scoped_ptr has */
4.      shared_ptr(shared_ptr const & r);
5.      template<class Y> shared_ptr(shared_ptr<Y> const & r);
6.  
7.      shared_ptr(shared_ptr && r);
8.      template<class Y> shared_ptr(shared_ptr<Y> && r);
9.  
10.     bool unique() const;
11.     long use_count() const;
12.     /*...*/
13. };
```

# shared_ptr

- Можно класть в STL контейнеры (есть даже сравнение)
- Полный тип требует только на момент конструирования от указателя
- Избегайте циклов (используйте weak_ptr)
- Избегайте передачи временных указателей в shared_ptr в сложный выражениях:

```
1. void foo(shared_ptr<A> a, int){/*...*/}
2. int bar() {/*may throw exception*/}
3.
4. int main()
5. {
6.     // be careful, it's dangerous!
7.     foo(shared_ptr<A>(new A), bar());
8. }
```

# make_shared, allocate_shared

```
1  // usual way
2  shared_ptr<some_struct> ptr(new some_struct(a, b, c));
3
4  // better way
5  auto ptr = make_shared<some_struct>(a, b, c);
```

- Умный указатели изолируют не только операторы delete, но и new
- Выделяет память на счетчик одним блоком с объектом
- Для выделения со свои аллокатором используйте `allocate_shared`

# weak_ptr

```
1   struct client;
2   struct server
3   {
4       //...
5       using client_ptr = shared_ptr<client>;
6       vector<client_ptr> clients_;
7   };
8
9   struct client
10  {
11      // ...
12      weak_ptr<server> srv_;
13  }
14
15  //... in client member function
16  if(auto srv = srv_.lock())
17  {
18      srv->send(/*...*/)
19  }
```

# boost::intrusive_ptr

- Хранит счетчик ссылок непосредственно в объекте

\+ Нет дополнительных расходов на выделение памяти

\+ Можно передавать «сырой» указатель

\+ Самый быстрый из умных указателей, разделяющих владение

\- Требует вмешательство в класс

\- Могут быть проблемы при построении иерархии

- Если неочевидно, что `intrusive_ptr` даст вам выигрыш, попробуйте сперва `shared_ptr`

# shared_from_this*

```cpp
1   struct client
2       : enable_shared_from_this
3   {
4       typedef shared_ptr<client> ptr_t;
5       ptr_t create(/*...*/) { return ptr_t(new client(/*...*/)); }
6
7   private:
8       void on_connected(service* srv)
9       {
10          srv->handle_read(bind(&client::on_read, shared_from_this(), _1));
11      }
12      client(/*...*/){/*...*/} // private constructor (!)
13      void on_read(/*...*/){/*...*/}
14      //....
15  };
```

# boost optional

- Очень похож на указатель, но хранит по значению в качестве своего поля. Вместе с флагом инициализации.

```
1   optional<double> try_get_value()
2   {
3       if (has_value)
4           return value;
5       else
6           return nullopt;
7   }
8   //-------------------------------
9   struct A
10  {
11      A(some_struct& s, double d);
12  }
13  //...
14  optional<A> a (A(s, 5.)); // makes copy
15  optional<A> b (in_place, ref(s), 5.); // inplace construction
```

# Вопросы?

# Завтра

- Move semantics, rvalue reference, perfect forwarding
- Callbacks: lambda, bind & function

- Задача для самостоятельного решения: linked_ptr<T>