



Lifetime and Usage of Global, Thread-local, and Static Data

May 16, 2014

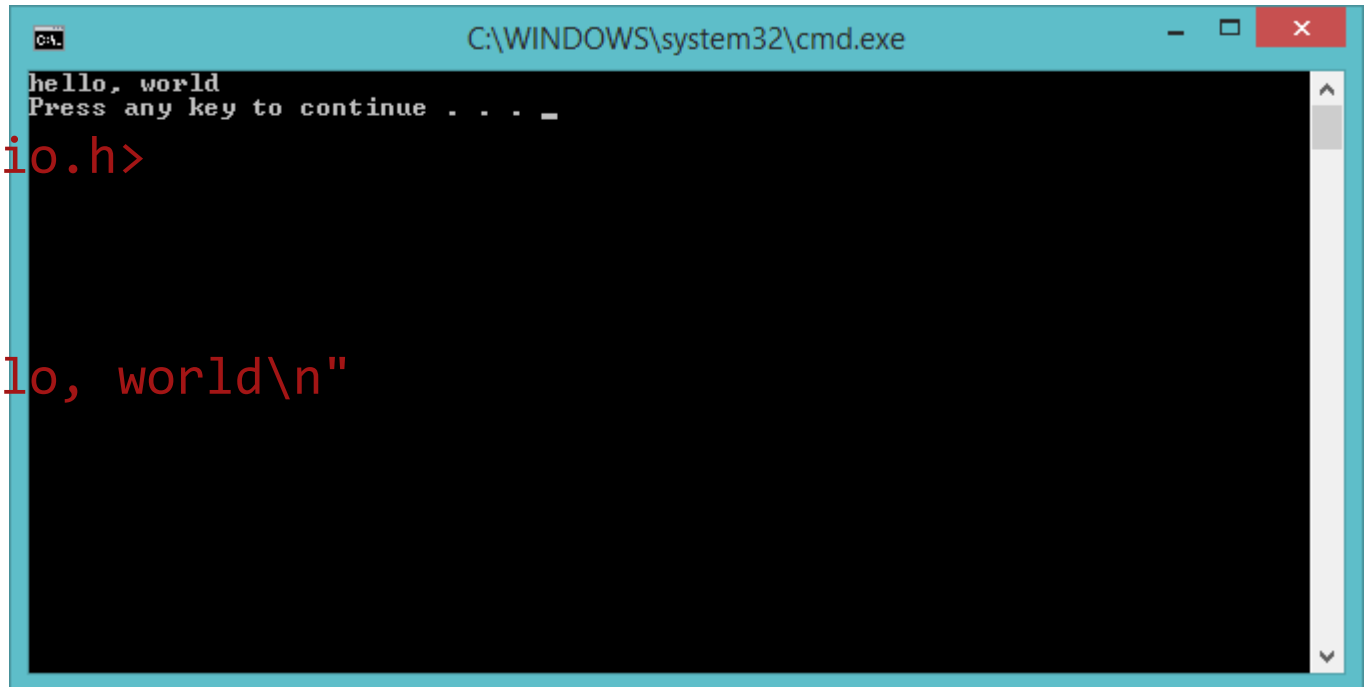
Imagination at work.

DON'T



```
#include <stdio.h>
```

```
int main() {  
    printf("hello, world\n")  
}
```

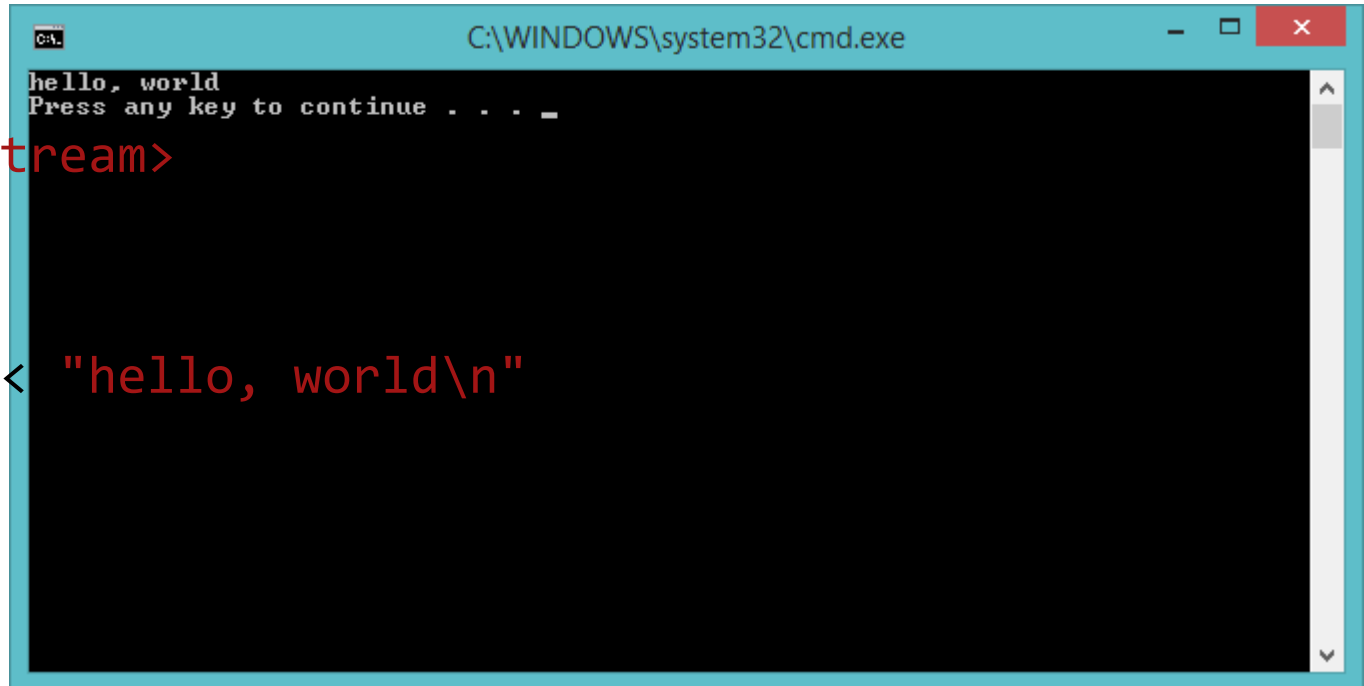


A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background and a light blue title bar. The output of a C program is displayed in white text: "hello, world" followed by a new line, and then "Press any key to continue . . . _". A vertical scrollbar is visible on the right side of the window.



```
#include <iostream>
```

```
int main() {  
    std::cout << "hello, world\n"  
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background and white text. It displays the output of a C++ program: "hello, world" followed by a new line, and then "Press any key to continue . . . _" with a cursor. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.



```
#include <iostream>
```

```
int main() {  
    std::operator<<(std::cout, "hello, world\n");  
}
```



Storage Duration

- Static storage duration
- Thread storage duration
- Automatic storage duration
- Dynamic storage duration



Static Storage Duration

All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have *static storage duration*. The storage for these entities shall last for the duration of the program.

3.7.1 ¶1



Thread Storage Duration

All variables declared with the `thread_local` keyword have *thread storage duration*. The storage for these entities shall last for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.

3.7.2 ¶1



Automatic Storage Duration

Block-scope variables explicitly declared register or not explicitly declared static or extern have *automatic storage duration*. The storage for these entities lasts until the block in which they are created exits.

3.7.3 ¶1



Dynamic Storage Duration

Objects can be created dynamically during program execution, using *new-expressions*, and destroyed using *delete-expressions*. A C++ implementation provides access to, and management of, dynamic storage via the global *allocation functions* operator `new` and operator `new[]` and the global *deallocation functions* operator `delete` and operator `delete[]`.

3.7.4 ¶1



Problems With Global Variables

- Unintentional sharing
- Minimal lifetime control
- Spaghetti code
- Not thread safe
- Obfuscates code execution
- Initialization dependencies



Example



```
int main() {  
}
```



```
#include <vector>
```

```
int main() {  
    std::vector<void(*)()> test_cases;  
}
```



```
#include <vector>
```

```
void my_test() {  
    // Test something!  
}
```

```
int main() {  
    std::vector<void(*)()> test_cases;  
}
```



```
#include <vector>
```

```
void my_test() {  
    // Test something!  
}
```

```
int main() {  
    std::vector<void(*)()> test_cases;  
    test_cases.push_back(my_test);  
}
```




```
#include <vector>
```

```
void my_test() {  
    // Test something!  
}
```

```
int main() {  
    std::vector<void(*)()> test_cases;  
    test_cases.push_back(my_test);  
    for (auto test : test_cases) {  
        test();  
    }  
}
```



```
#include <vector>

std::vector<void(*)()> test_cases;

void my_test() {
    // Test something!
}

int main() {
    test_cases.push_back(my_test);
    for (auto test : test_cases) {
        test();
    }
}
```



```
#include <vector>

std::vector<void(*)()> test_cases;

void my_test() {
    // Test something!
}

test_cases.push_back(my_test);

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



```
#include <vector>

std::vector<void(*)()> test_cases;

void my_test() {
    // Test something!
}

struct my_test_helper {
};

static my_test_helper register_my_test;

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



```
#include <vector>

std::vector<void(*)()> test_cases;

void my_test() {
    // Test something!
}

struct my_test_helper {
    my_test_helper() {
        test_cases.push_back(my_test);
    }
};

static my_test_helper register_my_test;

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



```
#include <vector>

std::vector<void(*)()> test_cases;

struct test_helper {
    test_helper(void(*test)()) {
        test_cases.push_back(test);
    }
};

void my_test() {
    // Test something!
}

static test_helper register_my_test(my_test);

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



```

#include <vector>

std::vector<void(*)()> test_cases;

struct test_helper {
    test_helper(void(*test)()) {
        test_cases.push_back(test);
    }
};

{
    void my_test();
    static test_helper register_my_test(my_test);
    void my_test() {
        // Test something!
    }
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}

```



```

#include <vector>

std::vector<void(*)()> test_cases;

struct test_helper {
    test_helper(void(*test)()) {
        test_cases.push_back(test);
    }
};

#define CREATE_TEST(name) void name();\
    static test_helper register_##name(name);\
    void name()

CREATE_TEST(my_test) {
    // Test something!
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}

```



Boost.Test



```
#define BOOST_TEST_MODULE MyTest
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(my_test) {
    // Test something!
}
```



```

struct my_test : public BOOST_AUTO_TEST_CASE_FIXTURE {
    void test_method();
};

static void my_test_invoker() {
    my_test t;
    t.test_method();
}

struct my_test_id {};

static boost::unit_test::ut_detail::auto_test_unit_registrar
my_test_registrar24(
    boost::unit_test::make_test_case(&my_test_invoker, "my_test"),
    boost::unit_test::ut_detail::auto_tc_exp_fail<
        my_test_id>::instance()->value()
);

void my_test::test_method() {
    // Test something!
}

```



Example – Part 2



```

#include <vector>

std::vector<void(*)()> test_cases;

struct test_helper {
    test_helper(void(*test)()) {
        test_cases.push_back(test);
    }
};

#define CREATE_TEST(name) void name();\
    static test_helper register_##name(name);\
    void name()

CREATE_TEST(my_test) {
    // Test something!
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}

```



```

#ifndef TEST_H_
#define TEST_H_

struct test_helper {
    test_helper(void(*test)());
};

#define CREATE_TEST(name) void name();\
    static test_helper register_##name(name);\
    void name()

#endif

```

```

#include "test.h"
#include <vector>
static std::vector<void(*)()> test_cases;

test_helper::test_helper(void(*test)()) {
    test_cases.push_back(test);
}

CREATE_TEST(my_test) {
    // Test something!
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}

```



```
#include "test.h"
```

```
CREATE_TEST(my_test) {  
    // Test something!  
}
```

```
CREATE_TEST(second_test) {  
    // Test something else!  
}
```

```
#include "test.h"
```

```
#include <vector>
```

```
static std::vector<void(*)()> test_cases;
```

```
test_helper::test_helper(void(*test)()) {  
    test_cases.push_back(test);  
}
```

```
int main() {  
    for (auto test : test_cases) {  
        test();  
    }  
}
```



Order of Initialization and Destruction




```
#include <iostream>
```

```
struct A {
```

```
    A(const char* val) : v{ val } {
```

```
        std::cout << val << "\n";
```

```
    }
```

```
    ~A() {
```

```
        std::cout << "~" << "Press any key to continue . . ."
```

```
    }
```

```
    const char* v;
```

```
};
```

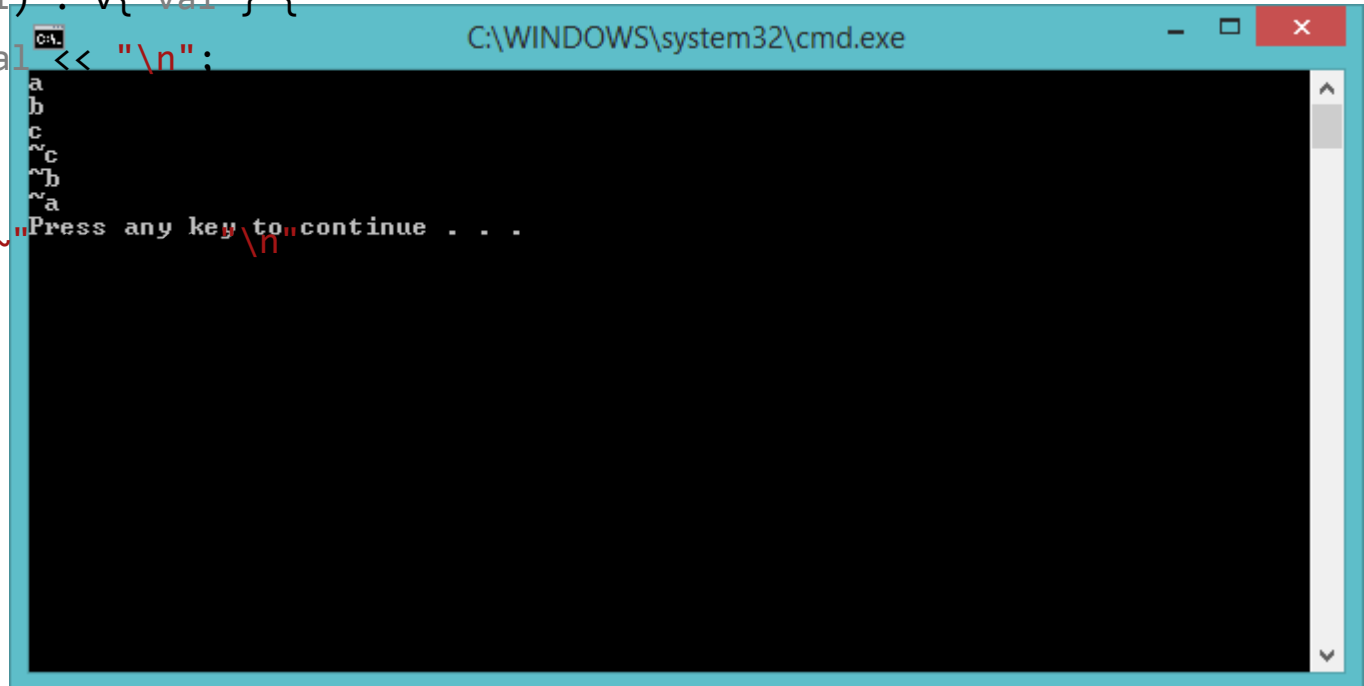
```
int main() {
```

```
    A a{ "a" };
```

```
    A b{ "b" };
```

```
    A c{ "c" };
```

```
}
```



```
#include <iostream>
```

```
struct A {
```

```
    A(const char* val) : v{ val } {
```

```
        std::cout << val << "\n";
```

```
    }
```

```
    ~A() {
```

```
        std::cout << "~" << val << "\n";
```

```
    }
```

```
    const char* v;
```

```
};
```

```
int main() {
```

```
    auto a = new A{ "a"
```

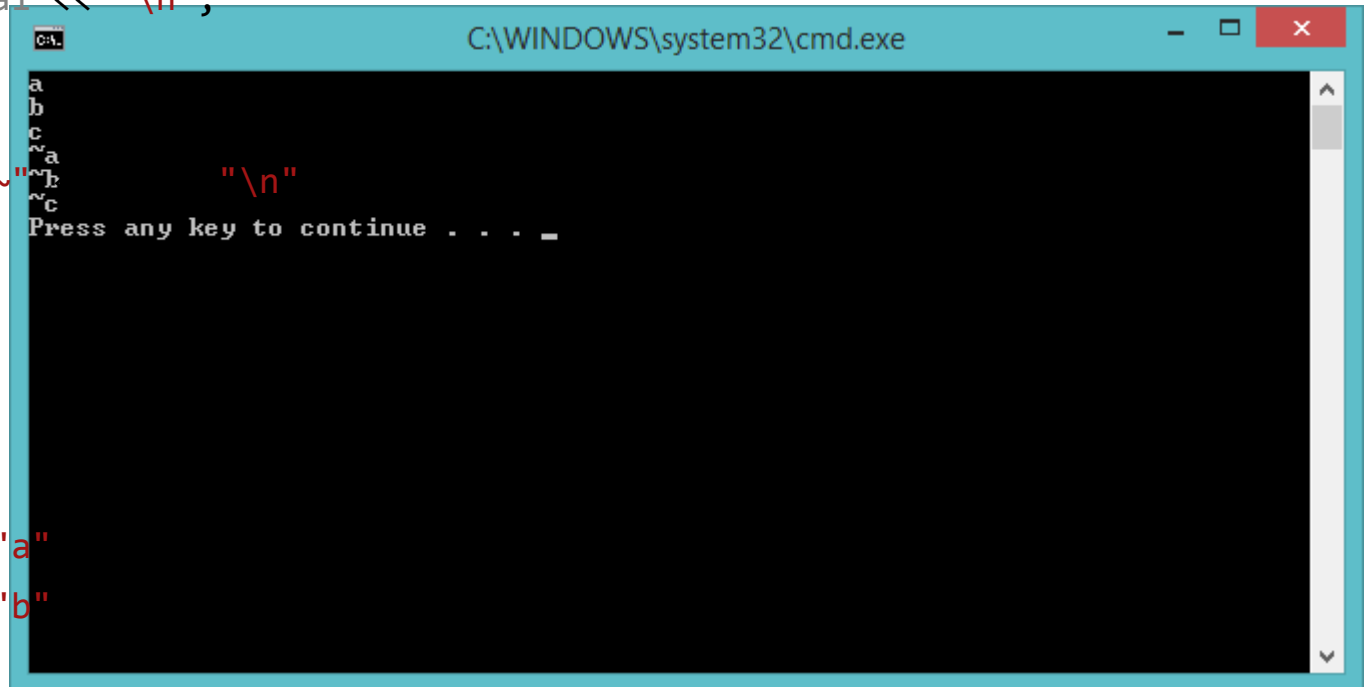
```
    auto b = new A{ "b"
```

```
    A c{ "c" };
```

```
    delete a;
```

```
    delete b;
```

```
}
```



```
C:\WINDOWS\system32\cmd.exe
a
b
c
~a
~b
~c
Press any key to continue . . . _
```



```
#include <iostream>
```

```
struct A {
```

```
    A(const char* val) : v{ val } {
```

```
        std::cout << val << "\n";
```

```
    }
```

```
    ~A() {
```

```
        std::cout << "~" << "\n";
```

```
    }
```

```
    const char* v;
```

```
};
```

```
A a{ "a" };
```

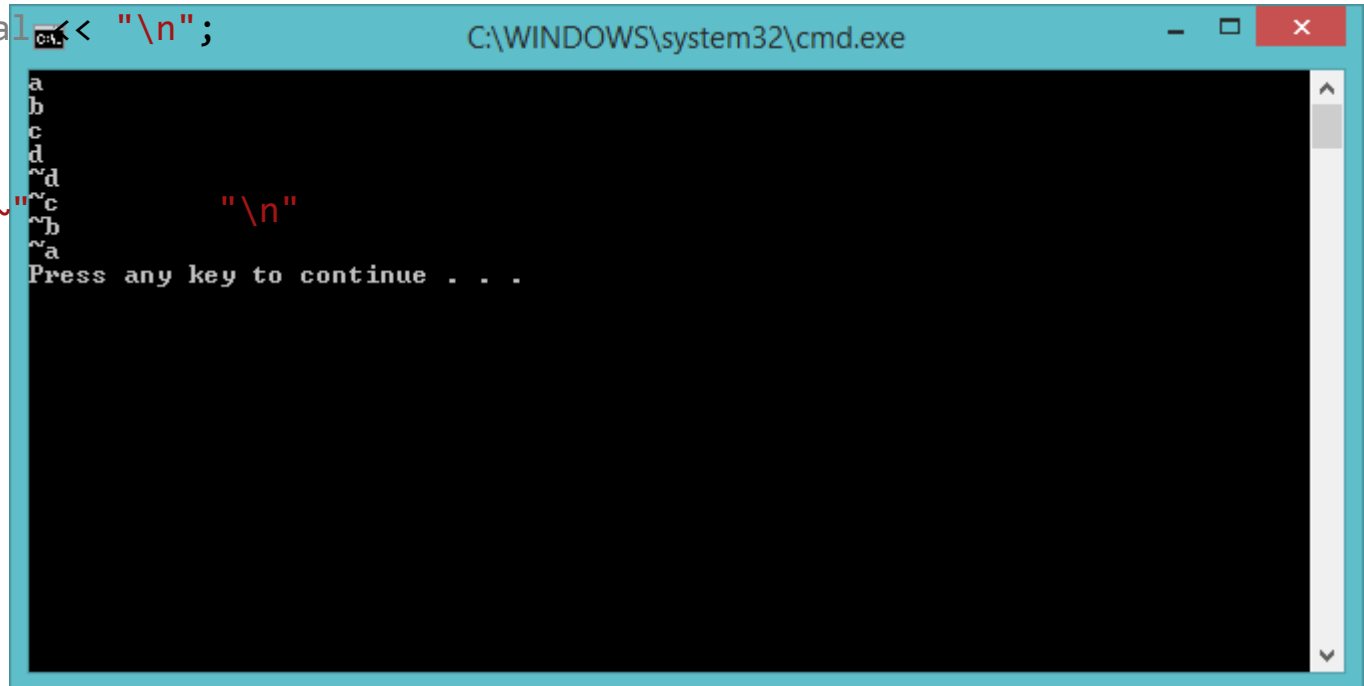
```
A b{ "b" };
```

```
int main() {
```

```
    A c{ "c" };
```

```
    A d{ "d" };
```

```
}
```



Static Initialization

- All variables with static storage duration are zero initialized
- All initializers with all constant expression parameters are executed



Dynamic Initialization – Ordered Initialization

- Explicitly specialized class template static data members
- Other non-local variables with static storage duration have ordered initialization



Dynamic Initialization – Unordered Initialization

- Other class template static data members



Uninitialized Memory

```
#include <stdlib>
```

```
struct C {  
    C(){  
        a = rand();  
        b = rand();  
    }  
    short a;  
    int b;  
};
```

```
C first;  
short second;  
char third{'a'};
```

Name	Address	Value
first.a	0x0000	0xde
	0x0001	0xad
padding	0x0002	0xbe
	0x0003	0xef
first.b	0x0004	0xde
	0x0005	0xad
	0x0006	0xbe
	0x0007	0xef
second	0x0008	0xde
	0x0009	0xad
third	0x000A	0xbe



Zero Initialized

```
#include <stdlib>
```

```
struct C {  
    C(){  
        a = rand();  
        b = rand();  
    }  
    short a;  
    int b;  
};
```

```
C first;  
short second;  
char third{'a'};
```

Name	Address	Value
first.a	0x0000	0x00
	0x0001	0x00
padding	0x0002	0x00
	0x0003	0x00
first.b	0x0004	0x00
	0x0005	0x00
	0x0006	0x00
	0x0007	0x00
second	0x0008	0x00
	0x0009	0x00
third	0x000A	0x00



Constant Initialization

```
#include <stdlib>
```

```
struct C {  
    C(){  
        a = rand();  
        b = rand();  
    }  
    short a;  
    int b;  
};
```

```
C first;  
short second;  
char third{'a'};
```

Name	Address	Value
first.a	0x0000	0x00
	0x0001	0x00
padding	0x0002	0x00
	0x0003	0x00
first.b	0x0004	0x00
	0x0005	0x00
	0x0006	0x00
	0x0007	0x00
second	0x0008	0x00
	0x0009	0x00
third	0x000A	0x61



Dynamic Initialization

```
#include <stdlib>
```

```
struct C {  
    C(){  
        a = rand();  
        b = rand();  
    }  
    short a;  
    int b;  
};
```

```
C first;  
short second;  
char third{'a'};
```

Name	Address	Value
first.a	0x0000	0x1a
	0x0001	0x03
padding	0x0002	0x00
	0x0003	0x00
first.b	0x0004	0x7d
	0x0005	0xf0
	0x0006	0x4d
	0x0007	0x33
second	0x0008	0x00
	0x0009	0x00
third	0x000A	0x61



Relative ordering between
translation units is
unspecified



```
struct A {
    A(const char*);
    ~A();
    const char* v;
};
```

```
A e{ "e" };
A f{ "f" };
```

```
struct A {
    A(const char*);
    ~A();
    const char* v;
};
```

```
#include <iostream>
```

```
A::A(const char* val) : v{ val } {
    std::cout << v << "\n";
}
A::~~A() {
    std::cout << "~" << v << "\n";
}
```

```
A a{ "a" };
A b{ "b" };
```

```
int main() {
    A c{ "c" };
    A d{ "d" };
}
```



Solution

When using a global variable in a constructor, include the header associated with the global in your type's header.



```

#ifndef HEADER_H_
#define HEADER_H_

struct A {
    A();
    int get();
    int i;
};

struct B {
    B();
    int get();
    int i;
};

extern A a;
extern B b;

#endif

```

```

#include "test.h"

A a;

A::A() : i{ b.get() } {

int A::get() {
    return i;
}

```

```

#include "test.h"

B b;

B::B() : i{ a.get() } {

int B::get() {
    return i;
}

```



```

#ifndef HEADER_H_
#define HEADER_H_

struct A {
    A();
    int get();
    int i;
};

struct B {
    B();
    int get();
    int i;
};

extern A a;
extern B b;

#endif

```

```

#include "test.h"

```

```

A a;

```

```

A::A() : i{ 5 } {
}

```

```

int A::get() {
    return i;
}

```

```

#include "test.h"

```

```

B b;

```

```

B::B() : i{ a.get() } {
}

```

```

int B::get() {
    return i;
}

```



Constraining Initialization Order

- Singleton
- Schwarz Counter



Meyers Singleton

```
std::vector<void(*)()> test_cases()  
{  
    static std::vector<void(*)()> cases;  
    return cases;  
}
```



Schwarz Counter

Developed by Jerry Schwarz

Constrains initialization order between translation units



```

#ifndef HEADER_H_
#define HEADER_H_
struct A_schwarz {
    A_schwarz();
};
static A_schwarz schwarz;
struct A {
    A();
    int get();
    int i;
};
struct B {
    B();
    int i;
};
extern A a;
extern B b;
#endif

```

```

#include "test.h"
#include <new>
static int schwarz_count;
A_schwarz::A_schwarz() {
    if (schwarz_count++ != 0)
        new(&a) A();
}
A a;
A::A() : i{ 5 } {}
int A::get() { return i; }

```

```

#include "test.h"

B b;

B::B() : i{ a.get() } {
}

```



Example – Part 3



```
#include "test.h"
```

```
CREATE_TEST(my_test) {  
    // Test something!  
}
```

```
CREATE_TEST(second_test) {  
    // Test something else!  
}
```

```
#include "test.h"
```

```
#include <vector>
```

```
static std::vector<void(*)()> test_cases;
```

```
test_helper::test_helper(void(*test)()) {  
    test_cases.push_back(test);  
}
```

```
int main() {  
    for (auto test : test_cases) {  
        test();  
    }  
}
```



Singleton

```
#include "test.h"
```

```
CREATE_TEST(my_test) {  
    // Test something!  
}
```

```
CREATE_TEST(second_test) {  
    // Test something else!  
}
```

```
#include "test.h"  
#include <vector>
```

```
std::vector<void(*)()>& test_cases() {  
    static std::vector<void(*)()> cases;  
    return cases;  
}
```

```
test_helper::test_helper(void(*test)()) {  
    test_cases().push_back(test);  
}
```

```
int main() {  
    for (auto test : test_cases()) {  
        test();  
    }  
}
```



Schwarz Counter

```
#ifndef TEST_H_
#define TEST_H_
struct test_schwarz {
    test_schwarz();
};
static test_schwarz test_cases_schwarz;
struct test_helper {
    test_helper(void(*test)());
};
#define CREATE_TEST(name) void name();\
    static test_helper register_##name(name);\
    void name()
#endif
```



Schwarz Counter

```
#include "test.h"
```

```
CREATE_TEST(my_test) {  
    // Test something!  
}  
  
CREATE_TEST(second_test) {  
    // Test something else!  
}
```

```
#include "test.h"  
#include <vector>
```

```
static std::vector<void(*)()> test_cases;  
static int schwarz_count;
```

```
test_schwarz::test_schwarz() {  
    if (!schwarz_count++) {  
        test_cases = std::vector<void(*)()>{};  
    }  
}
```

```
test_helper::test_helper(void(*test)()) {  
    test_cases.push_back(test);  
}
```

```
int main() {  
    for (auto test : test_cases) {  
        test();  
    }  
}
```



Schwarz Counter

```
#ifndef TEST_H_
#define TEST_H_
struct test_schwarz {
    test_schwarz();
    ~test_schwarz();
};
static test_schwarz test_cases_schwarz;
struct test_helper {
    test_helper(void(*test)());
};
#define CREATE_TEST(name) void name();\
    static test_helper register_##name(name);\
    void name()
#endif
```



Schwarz Counter

```
#include "test.h"
#include <type_traits>
#include <vector>

static std::aligned_storage<sizeof(std::vector<void(*)>>::type test_cases_location;
static std::vector<void(*)>& test_cases =
    *reinterpret_cast<std::vector<void(*)>>*>(&test_cases_location);
static int schwarz_count;

test_schwarz::test_schwarz() {
    if (!schwarz_count++) {
        new(&test_cases_location) std::vector<void(*)>{};
    }
}

test_schwarz::~~test_schwarz() {}

test_helper::test_helper(void(*test)()) {
    test_cases.push_back(test);
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



Schwarz Counter

```
#include "test.h"
#include <type_traits>
#include <vector>

static std::aligned_storage<sizeof(std::vector<void(*)>>::type test_cases_location;
static std::vector<void(*)>& test_cases =
    *reinterpret_cast<std::vector<void(*)>>*>(&test_cases_location);
static int schwarz_count;

test_schwarz::test_schwarz() {
    if (!schwarz_count++) {
        new(&test_cases_location) std::vector<void(*)>{};
    }
}

test_schwarz::~test_schwarz() {
    if (!--schwarz_count) {
        test_cases.std::vector<void(*)>::~~vector();
    }
}

test_helper::test_helper(void(*test)()) {
    test_cases.push_back(test);
}

int main() {
    for (auto test : test_cases) {
        test();
    }
}
```



Conclusions

- Global constructors can be problematic
- Avoid circular constructor dependencies
- Ensure ordering of dependencies
- Avoid globals for data sharing



