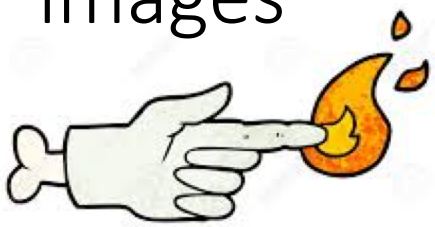
Will Your Code Survive the Attack of the Zombie Pointers?

The perils of lifetime-end pointer zap

Paul E. McKenney, Maged Michael, and Michael Wong

Zombie pointer images



200 × 200

700 × 451

590 × 874

Attributed from google image search

Back in 1973, When Dinosaurs Walked the Earth…

- Smith and Brown mentioned a LIFO push algorithm on an IBM 370
  - Written in IBM 370 BAL (“Business Assembly Language”)
- Simplest known concurrent algorithm demonstrating zombie pointers
- We don’t know when this algorithms was invented
  - Opportunity for aspiring software archaeologists!!!

Back in 1973, When Dinosaurs Walked the Earth... *

- Smith and Brown mentioned a LIFO push algorithm on an IBM 370
  - Written in IBM 370 BAL ("Business Assembly Language")
- Simplest known concurrent algorithm demonstrating zombie pointers
- We don't know when this algorithms was invented
  - Opportunity for aspiring software archaeologists!!!
- Coincidentally, in 1973, Paul programmed a computer for the first time
  - High school class as a sophomore
  - Used nearest computer, which was 14 miles from Paul's home town
  - But a lowly IBM 360, not a fancy new 370
  - And yes, this did involve both punched cards and FORTRAN. Why do you ask?
  - And no, for whatever reason, this class taught neither concurrency nor C++

* They still walk they earth. We call some of them "ostriches".

Back in 1973, When Dinosaurs Walked the Earth... *

- Smith and Brown mentioned a LIFO push algorithm on an IBM 370
  - Written in IBM 370 BAL ("Business Assembly Language")
- Simplest known concurrent algorithm demonstrating zombie pointers
- We don't know when this algorithms was invented
  - Opportunity for aspiring software archaeologists!!!
- Coincidentally, in 1973, Paul programmed a computer for the first time
  - High school class as a sophomore
  - Used nearest computer, which was 14 miles from Paul's home town
  - But a lowly IBM 360, not a fancy new 370
  - And yes, this did involve both punched cards and FORTRAN. Why do you ask?
  - And no, for whatever reason, this class taught neither concurrency nor C++
  - Nor did the next year's version of this class use the 360

* They still walk they earth. We call some of them "ostriches".

LIFO Push in C++: Data Members & Empty Check

```
template<typename T>
class LifoPush {
	class Node {
	public:
		T val;
		Node *next;
		Node(T v) : val(v) { }
	};

	std::atomic<Node *> top{nullptr};

public:
	bool list_empty()
	{
		return top.load() == nullptr;
	}
```

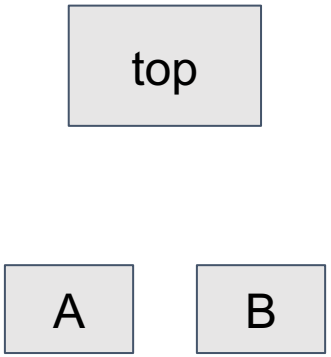
LIFO Push in C++: Push and Pop All

```
void list_push(T v)
{
        Node *newnode = new Node(v);
        newnode->next = top.load();
        while (!top.compare_exchange_weak(newnode->next, newnode))
                ;
}

template<typename F>
void list_pop_all(F f)  // Pop all instead of pop for simplicity
{
        Node *p = top.exchange(nullptr);
        while (p) {
                Node *next = p->next;
                f(p->val);
                delete p;
                p = next;
        }
}
```

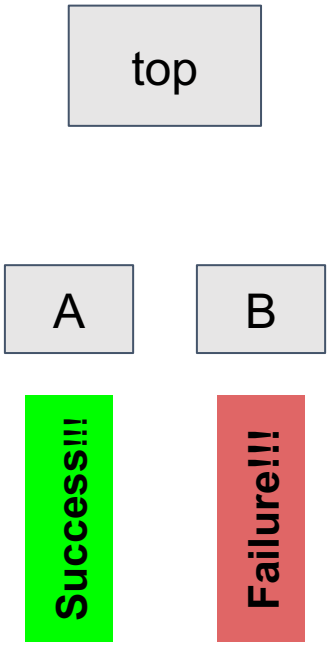
But What About Racing list_push()???

But What About Racing list_push()???



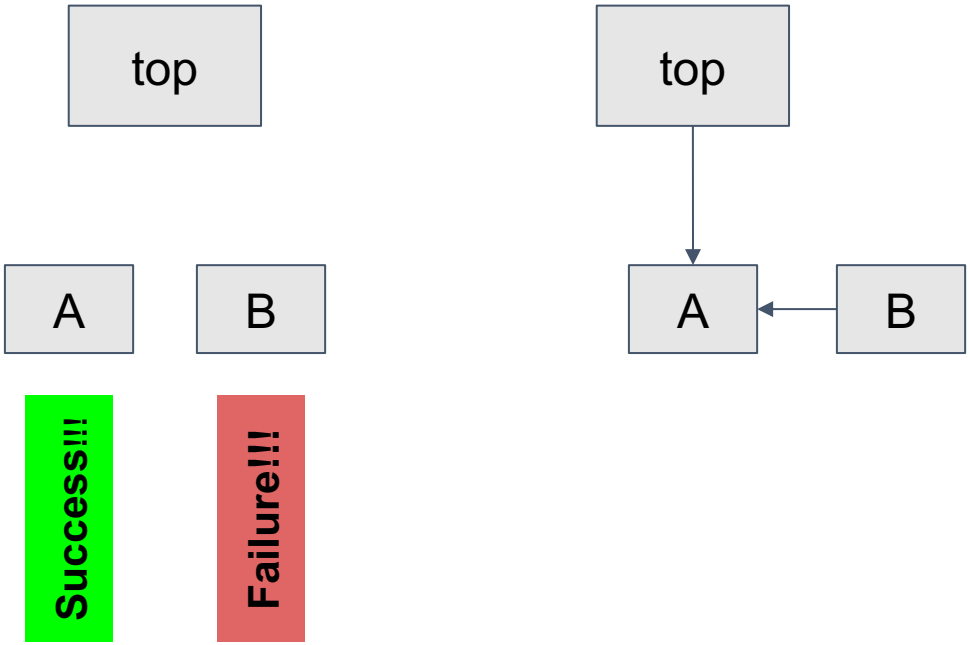

```
Node *newnode = new Node(v);
newnode->next = top.load();
```

But What About Racing list_push()???



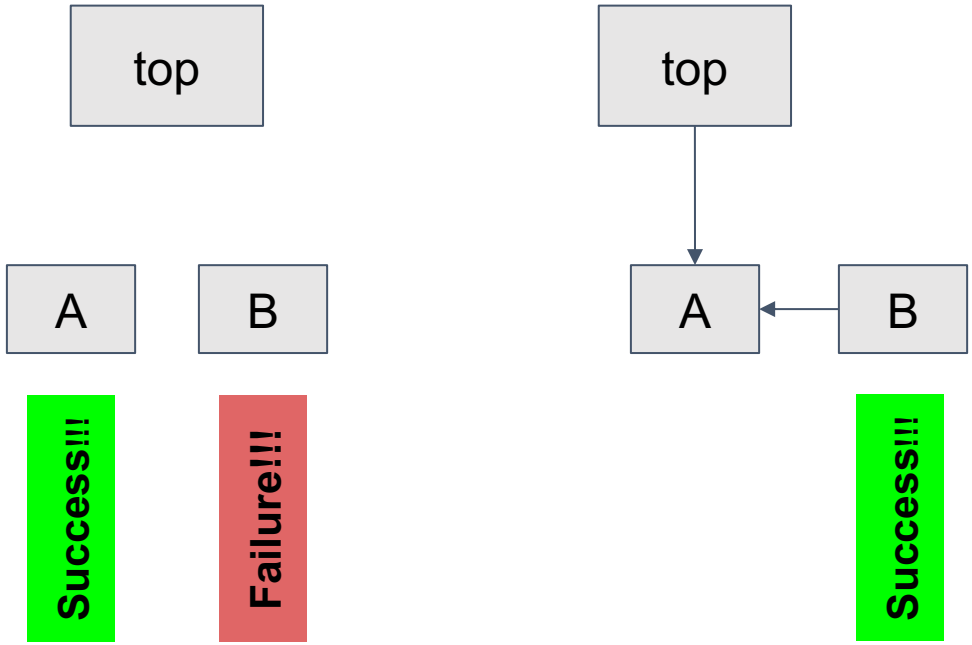

```
top.compare_exchange_weak(newnode->next, newnode)
```

But What About Racing list_push()???



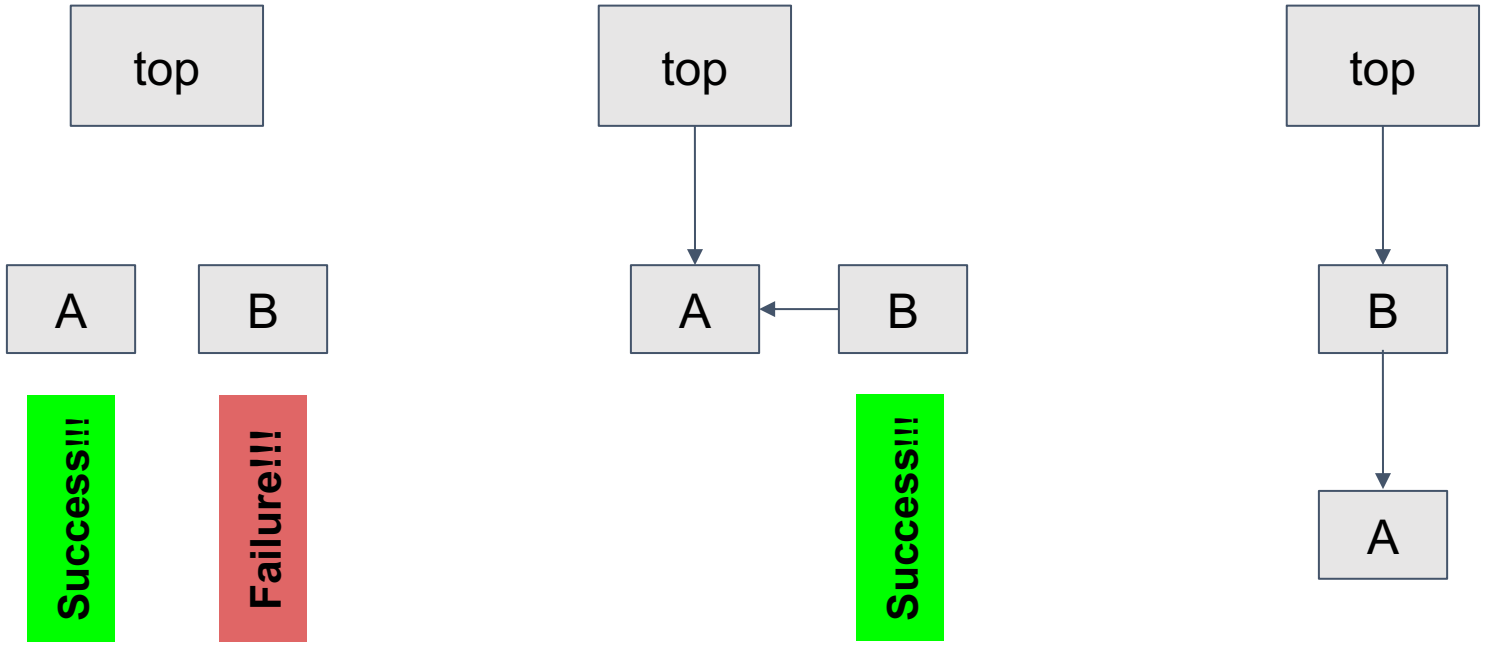

```
top.compare_exchange_weak(newnode->next, newnode)…
```

But What About Racing list_push()???

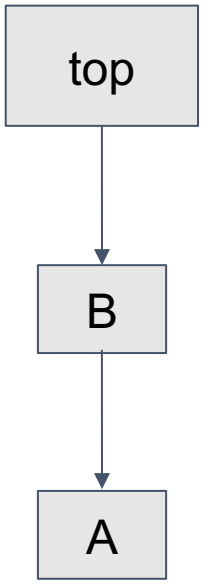



```
top.compare_exchange_weak(newnode->next, newnode)…
```

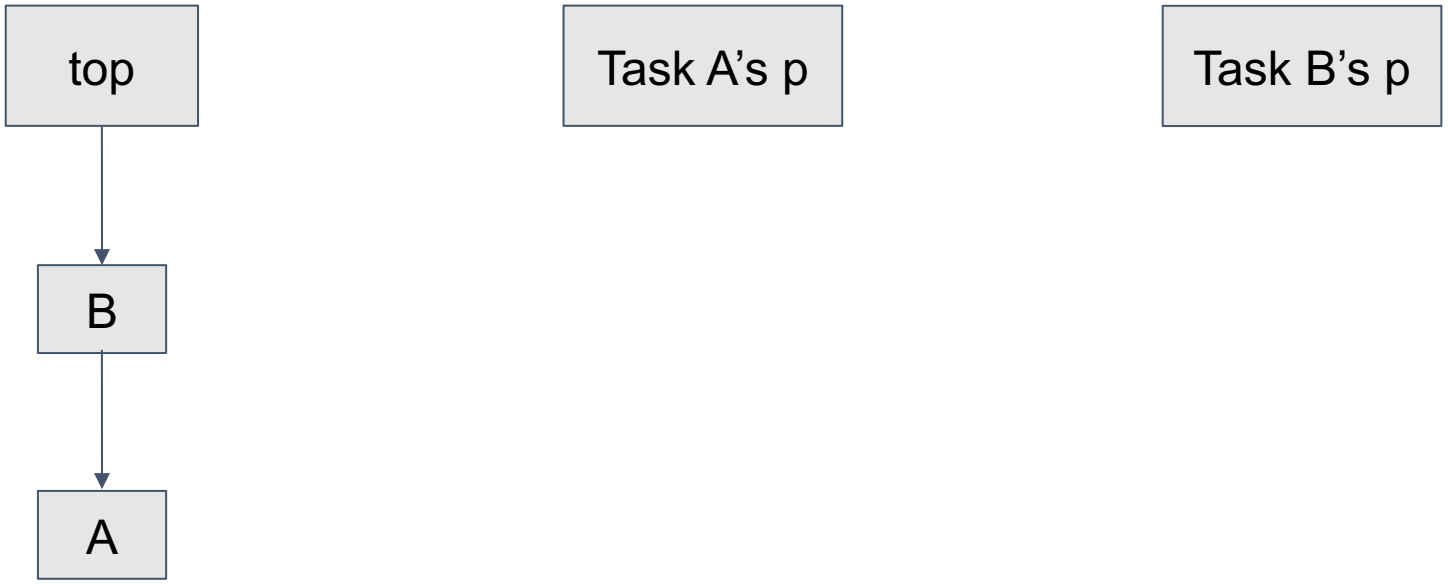
But What About Racing list_push()???



OK, But What About Racing Pop-All?

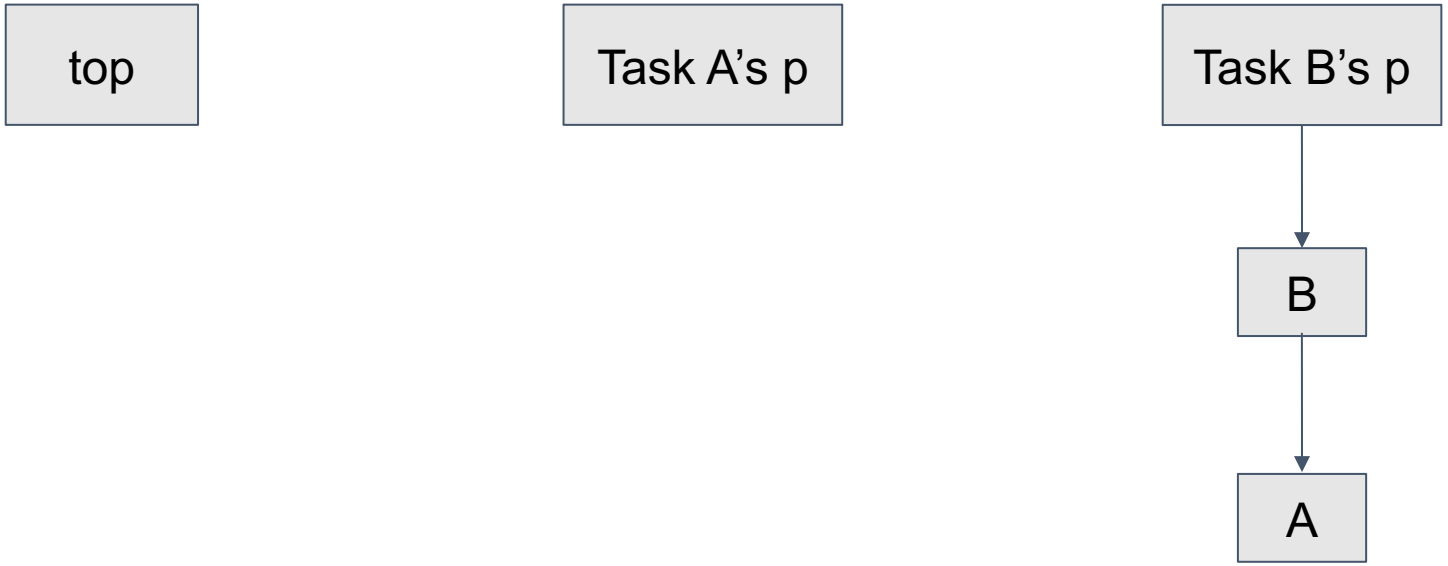


OK, But What About Racing Pop-All?



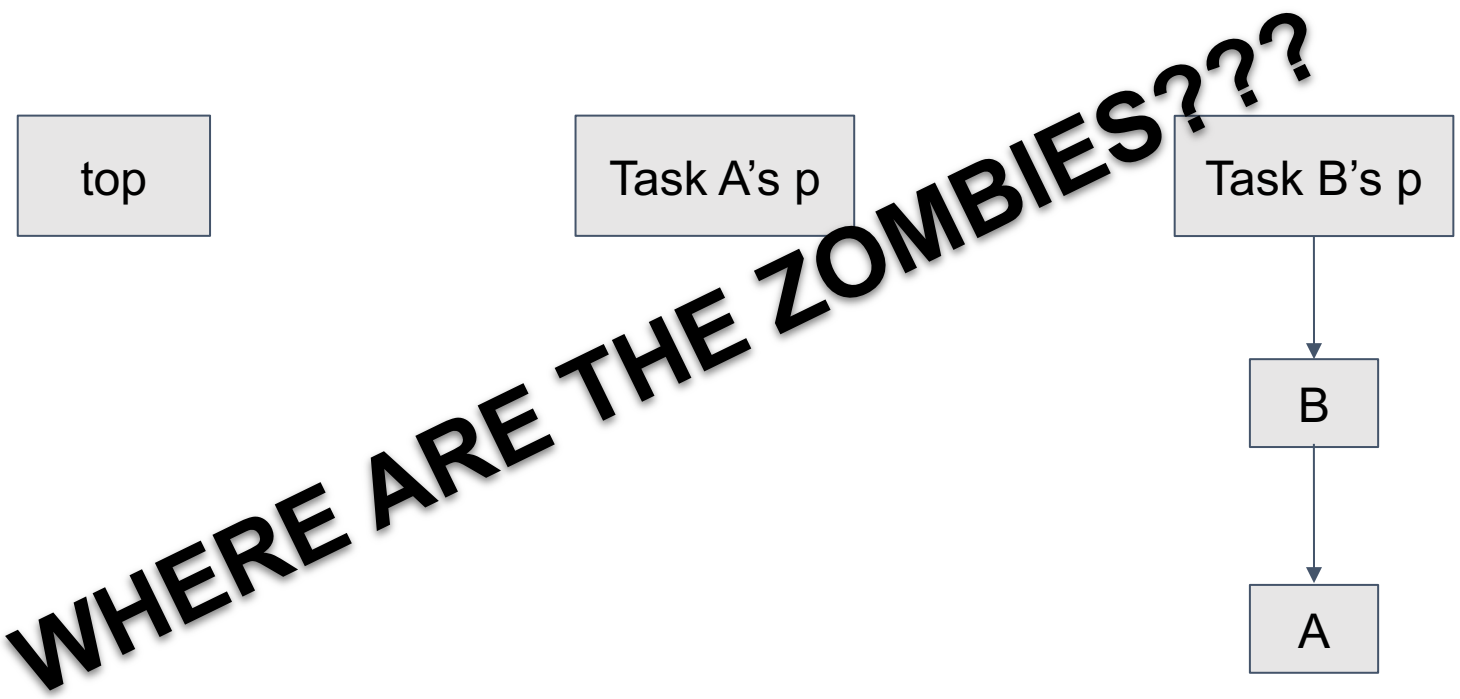

```
Node *p = top.exchange(nullptr);
```

OK, But What About Racing Pop-All?




```
Node *p = top.exchange(nullptr);
```

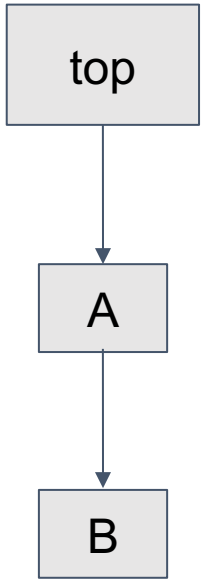
OK, But What About Racing Pop-All?



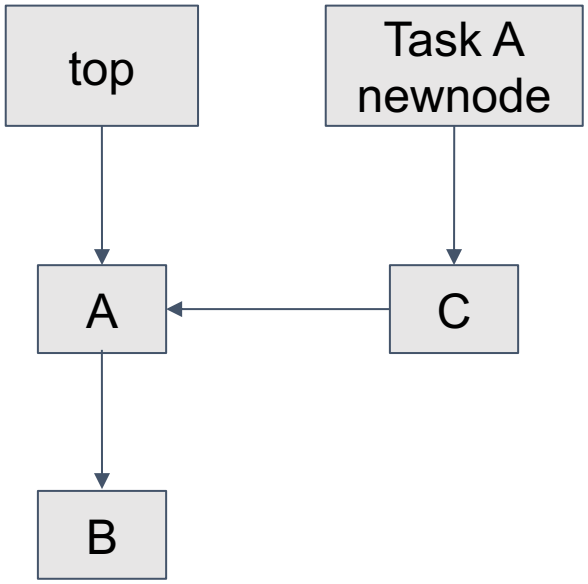

```
Node *p = top.exchange(nullptr);
```

Fine!!! Racing Pushes and Pop-All, Then!!!

- Assume a simple compiler
  - Or -OINT_MIN or assembly or...
- Start with a stack containing A and B
- Task A will do a push of C
- Task B will do a pop-all followed by a push of D

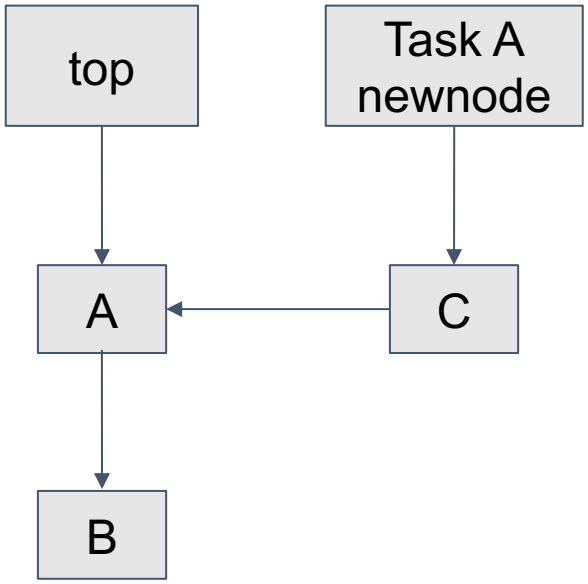


First, Task A Starts Pushing C:



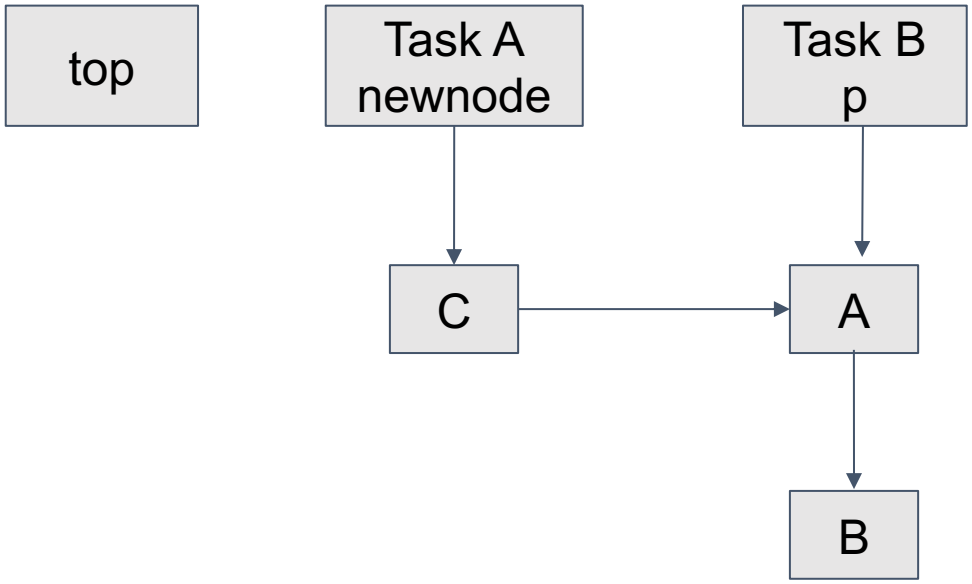

```
Node *newnode = new Node(v);
newnode->next = top.load();
```

First, Task A Starts Pushing C, Then Pauses



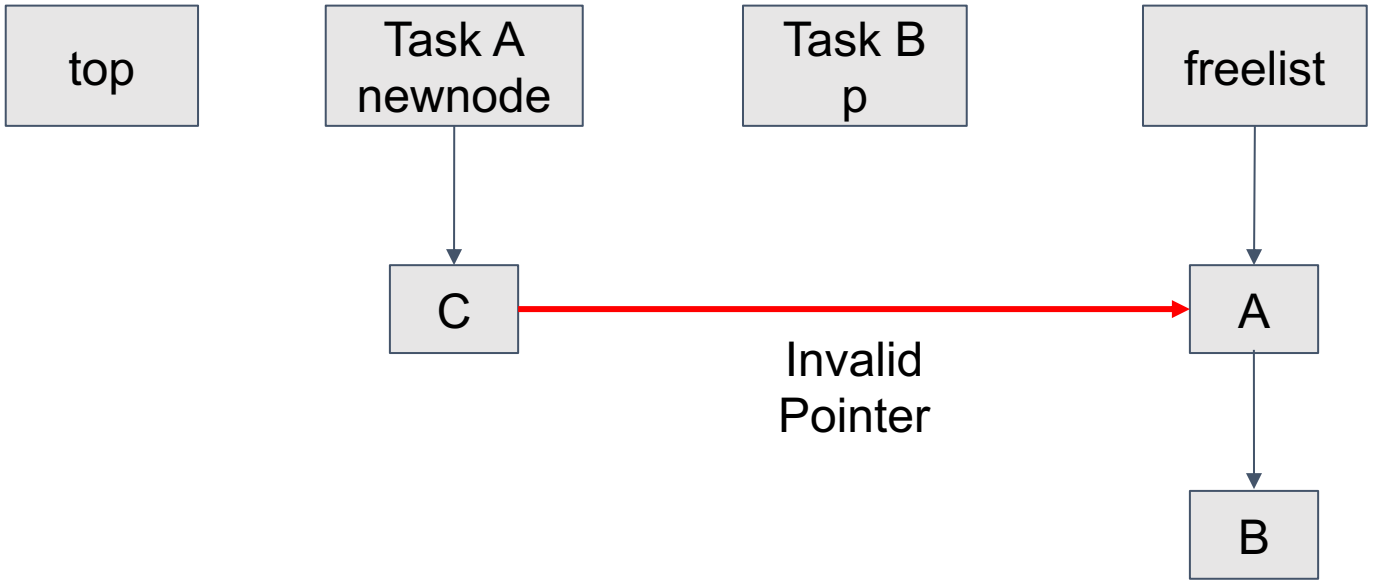

```
Node *newnode = new Node(v);
newnode->next = top.load();
```

Next, Task B Does Pop-All:



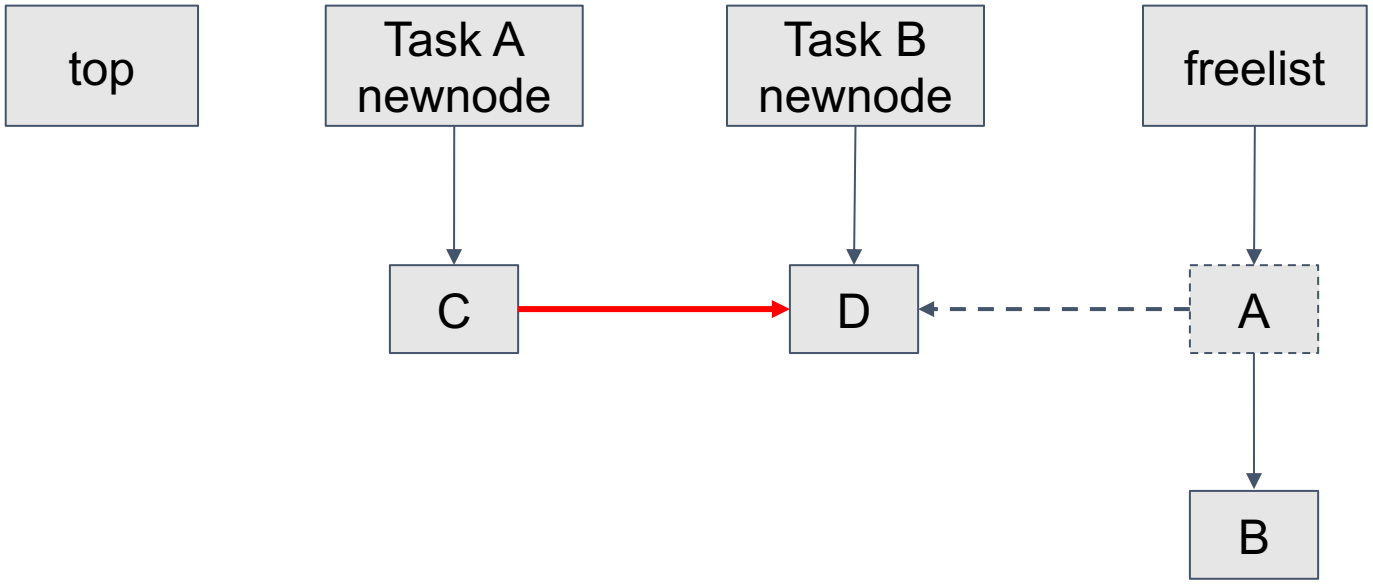

```
Node *p = top.exchange(nullptr);
```

Then Task B Does Some Deletes:



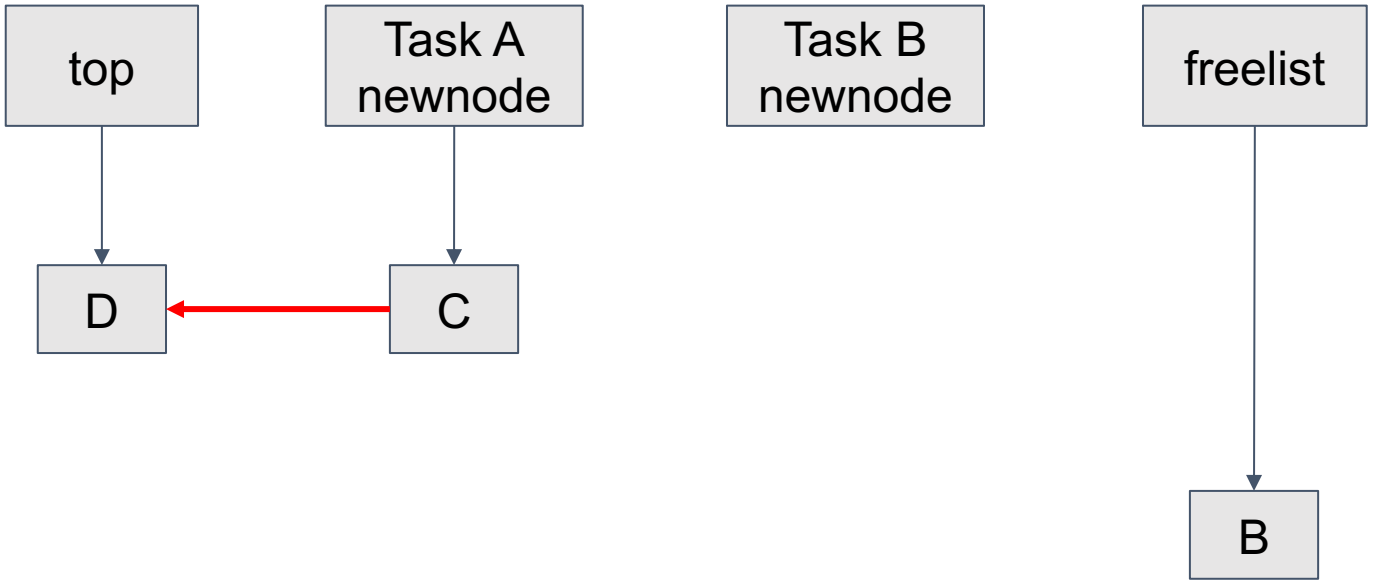

```
delete p;
```

Now Task B Starts a Push (Task A Still Paused):



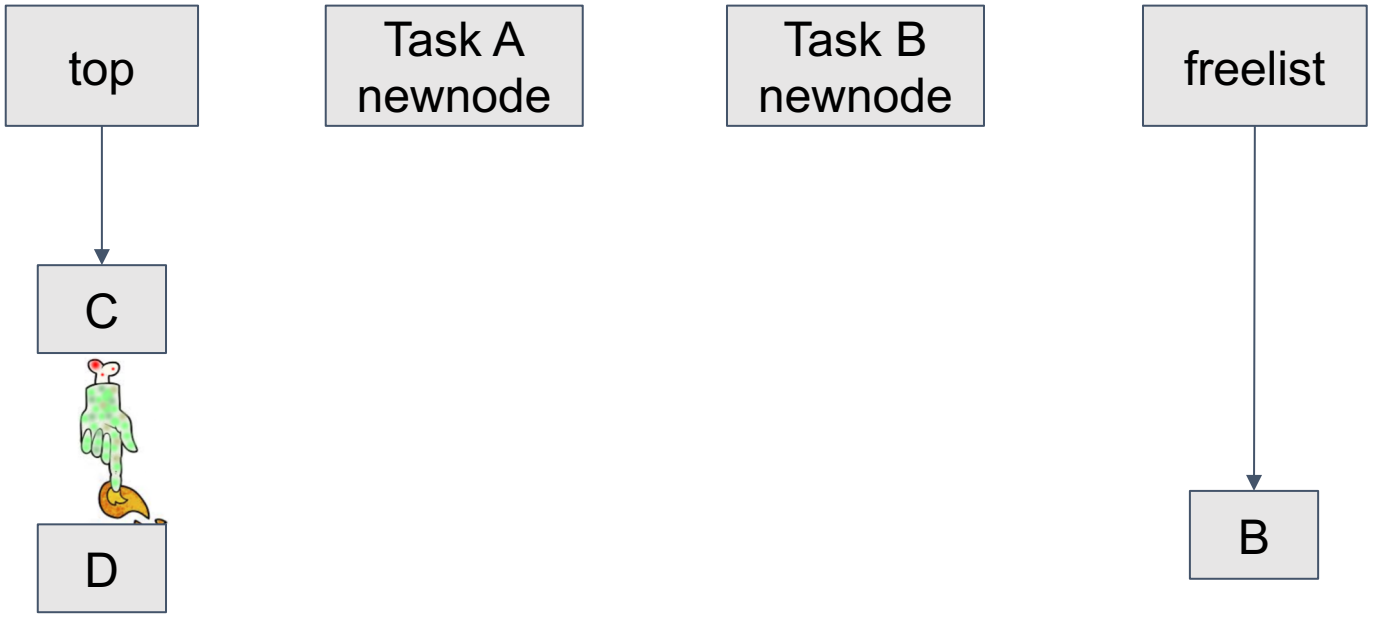

```
Node *newnode = new Node(v);
newnode->next = top.load();
```

Now Task B Completes Its Push:



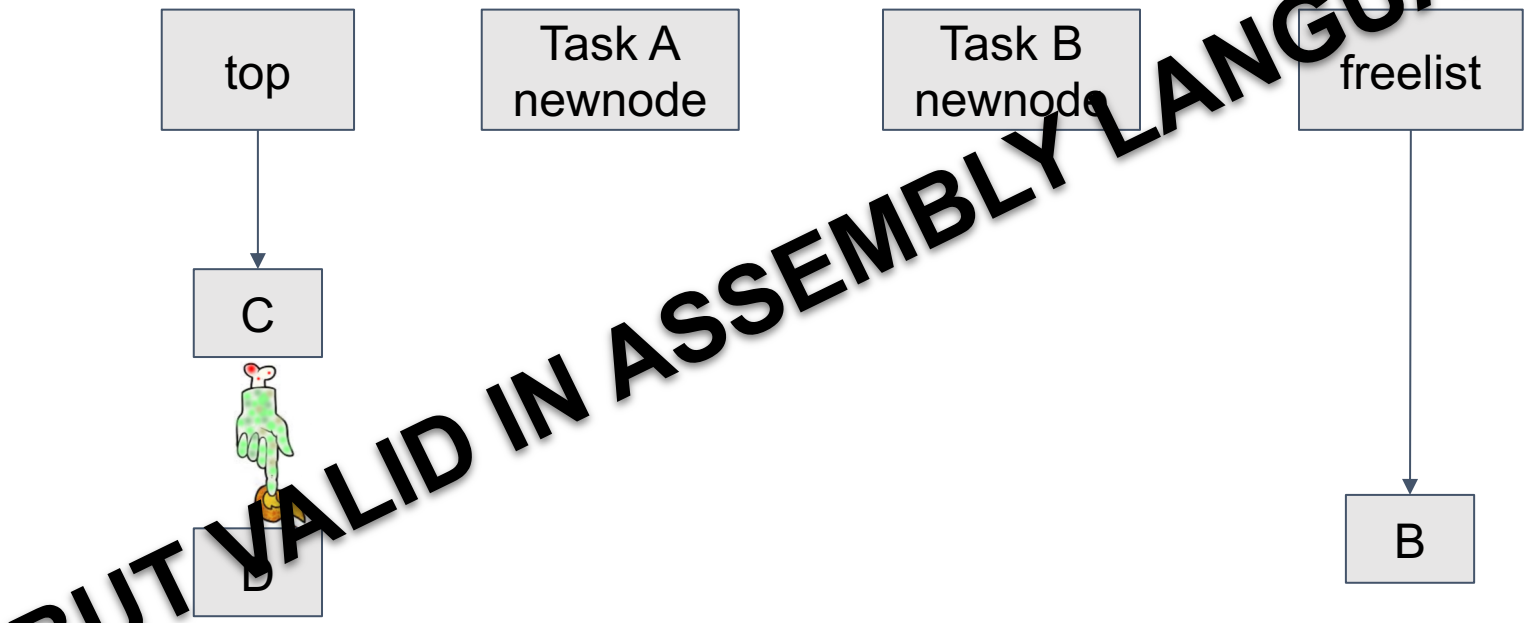

```
top.compare_exchange_weak(newnode->next, newnode)
```

And Task A Finally Gets to Complete Its Push:




```
top.compare_exchange_weak(newnode->next, newnode)
```

And Task A Finally Gets to Complete Its Push…



http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1726r0.pdf

How Did We Get Here???

- 1989 C standard: Algorithms like LIFO stack push are invalid
- 1998 C++ standard: What he said!
- 2001 C DR 260: And we really mean it!!!
- 2011 C and C++ standards add concurrency

How Did We Get Here???

- **1973: First known description of LIFO stack push**
- 1989 C standard: Algorithms like LIFO stack push are invalid
- 1998 C++ standard: What he said!
- 2001 C DR 260: And we really mean it!!!
- 2011 C and C++ standards add concurrency

Zombie Pointers vs. Pointer Provenance???

- Zombie pointers are known to have had their pointed-to objects deleted
- Pointer provenance involves pairs of pointers that are assumed valid
- LIFO stack push has issues with both!
- But this presentation focuses on zombie pointers

Library Need Not Be Written in Standard C++!!!

- So just put LIFO stack push in the library and get on with life!!!

Library Need Not Be Written in Standard C++!!!

- So just put LIFO stack push in the library and get on with life!!!
- Except that it is only one example from a large class of algorithms:
  - Algorithms that allow new references to objects to be obtained unconditionally, and that also allow objects to be freed unconditionally. LIFO push is one example, more on next slide
- For completeness, the other two classes are as follows:
  - Algorithms that must ask permission both before obtaining new references and before freeing objects. Lock-based algorithms are here, as are some reference-counting use cases
  - Algorithms that allow new references to objects to be obtained unconditionally, but that must ask permission before freeing objects. RCU and other reference-counting use cases are here

Library Need Not Be Written in Standard C++!!!

- So just put LIFO stack push in the library and get on with life!!!
- Except that it is only one example! Here are additional examples:
  - Optimized sharded locks (not easily buried in library)
  - Weak pointers
  - Identity-only pointers
  - Certain (single-threaded!) algorithms based on realloc()
  - Pointers as keys
  - Debug prints of pointers
  - Debug code to catch double-delete bugs
  - Garbage collectors
  - Handling of aliased pointers
  - Hazard pointers (not easily buried in library, and described by Maged Michael)

Hazard Pointer Example

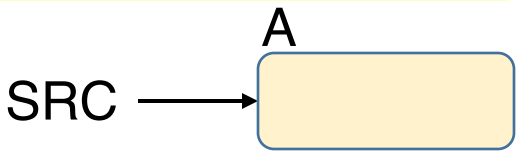
Hazard Pointers

A hazard pointer is a single-writer multi-reader pointer.

When a thread writes the address of object A to hazard pointer HP, it announces that
if A is removed after setting HP,
then A must not be reclaimed
as long as HP continues to hold the address of A.

Reader/Protector

```
HP = A
if SRC == A
  safe to use A
```

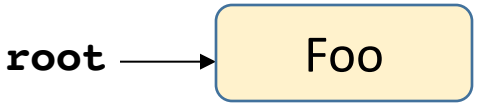


Remover/Reclaimer

```
SRC = B  // B != A
if all HP != A
  safe to reclaim A
```

Example with Zombie Pointer

```
// User code
std::atomic<Foo*> root;
```

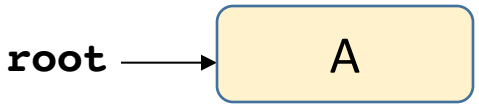



```
void Foo::update(Foo* newfoo) {
  Foo* oldfoo = root.load();
  root.store(newfoo);
  // Reclaim when safe
  oldfoo->retire();
}
```

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

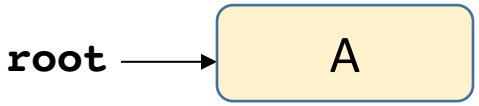
```
// Simplified library code
std::atomic<void*> hazptr_;
template <typename T>
bool hazard_pointer::try_protect(T*& ptr, std::atomic<T*>& src) {
  T* p = ptr;
  hazptr_.store(p);
  ptr = src.load();
  if (ptr == p) return true;
  // etc. failure case
}
```

Sequence of Events




```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

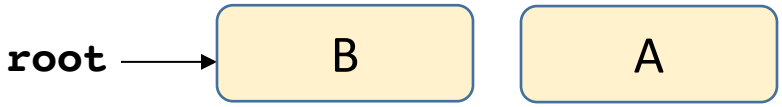
Thread 1 loads address of A from root into ptr // User code



T1: ptr=&A

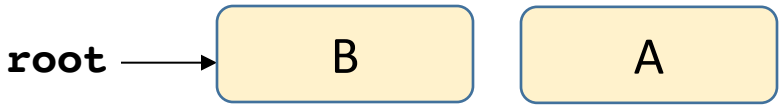
```
void Foo::update(Foo* newfoo) {
  Foo* oldfoo = root.load();
  root.store(newfoo);
  // Reclaim when safe
  oldfoo->retire();
}
```

Thread 2 sets root to point to B, removes and retires A



T1: ptr=&A

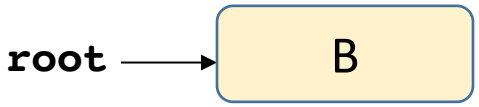
Sequence of Events



T1: ptr=&A

A is reclaimed

Thread 1's ptr is a zombie pointer (still in user code)



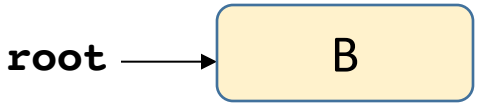
T1: ptr=??

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

Thread 1 calls hazard_pointer::try_protect(ptr, root)

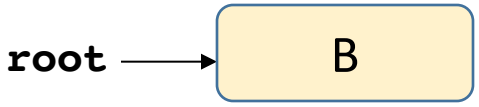
Sequence of Events

```
// try_protect(ptr, src = root)
  T* p = ptr;
  hp_.store(p);
  ptr = src.load();
  if (ptr == p) return true;
```



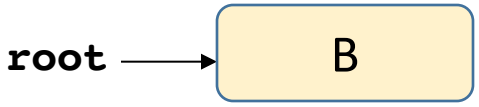
T1: ptr=??

Thread 1 reads from zombie pointer ptr into pointer p



T1: ptr=?? p=??

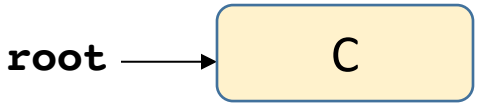
Thread 1 stores value of p into its hazard pointer



T1: ptr=?? p=?? hp_=??

Thread 2 constructs C in reallocated A's memory

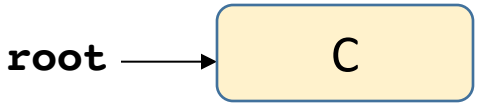
Thread 2 sets root to point to C (old address of A)



T1: ptr=?? p=?? hp_=??

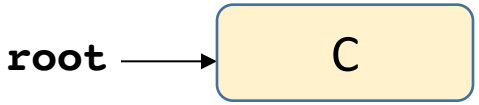
Sequence of Events

```
// try_protect(ptr, src = root)
  T* p = ptr;
  hp_.store(p);
  ptr = src.load();
  if (ptr == p) return true;
```



T1: ptr=?? p=?? hp_=??

Thread 1 loads address of C from root into ptr



T1: ptr=&C p=?? hp_=??

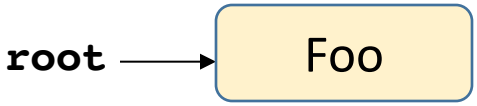
Thread 1 comparison of ptr with p may succeed

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

Thread 1 accesses C protected by a zombie pointer value

Pointer Zap Happened in User Code

```
// User code
std::atomic<Foo*> root;
```



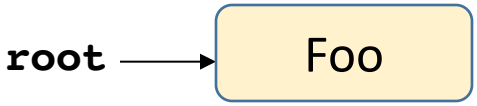

```
void Foo::update(Foo* newfoo) {
  Foo* oldfoo = root.load();
  root.store(newfoo);
  // Reclaim when safe
  oldfoo->retire();
}
```

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

```
// Simplified library code that leads to dereferencing zombie pointer
std::atomic<void*> hazptr_;
template <typename T>
bool hazard_pointer::try_protect(T*& ptr, std::atomic<T*>& src) {
  T* p = ptr;
  hazptr_.store(p);
  ptr = src.load();
  if (ptr == p) return true;
  // etc. failure case
}
```

Example Dereferencing a Zombie Pointer

```
// User code
std::atomic<Foo*> root;
```

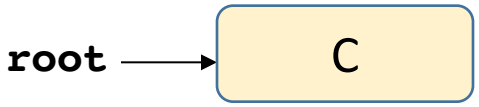



```
void Foo::update(Foo* newfoo) {
  Foo* oldfoo = root.load();
  root.store(newfoo);
  // Reclaim when safe
  oldfoo->retire();
}
```

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

```
// Simplified library code that leads to dereferencing zombie pointer
std::atomic<void*> hazptr_;
template <typename T>
bool hazard_pointer::try_protect(T*& ptr, std::atomic<T*>& src) {
  T* p = ptr;
  hazptr_.store(p);
  ptr = src.load();
  if (ptr == p) return true;
  if (src.load() == p) return true; // ptr is not overwritten
  // etc. failure case
}
```

Sequence of Events (fast forward)



T1: ptr=?? p=?? hp_=??

```
// try_protect(ptr, src = root)
  T* p = ptr;
  hp_.store(p);
  if (src.load() == p) return true;
```

Thread 1 comparison of &C with p may succeed

```
Value Foo::read() {
  hazard_pointer h;
  Foo* ptr = root.load();
  while (h.try_protect(ptr, root));
  /* Safe to dereference ptr */
  return ptr->value();
}
```

Thread 1 dereferences ptr (a zombie pointer)

Why should you care?

Why should you care?

- Not just for Concurrent use case
- Even single thread can have similar issues
- normal use of realloc() relies on comparing such pointer values

```
q = realloc(p, newsize);
if (q != p)
        update_my_pointers(p, q);
```

Pointer Provenance revisited

```
#include <stdio.h>
#include <string.h>
int y=2, x=1;

int main() {
  int *p = &x + 1;
  int *q = &y;

  printf("Addresses: p=%p
q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q,
sizeof(p)) == 0) {
      *p = 11;  // does this
have undefined behaviour?
      printf("x=%d y=%d *p=%d
*q=%d\n",x,y,*p,*q);
  }
}
```

GCC 8.1 -O2: x=1 y=2 *p=11 *q=2

ICC 19 -O2: x=1 y=2 *p=11 *q=11

Clang 6.0 -O2: x=1 y=11 *p=11 *q=11

No type-based aliasing here

Not affected by GCC or ICC's -fno-strict-aliasing

&x+1 one past pointer is explicitly permitted by ISO

Similar single threaded Use Cases from Cambridge

1. Using the pointer to the newly freed object as a key to container data structures, thus enabling further cleanup actions enabled by the free() .
2. Debug printing of the pointer (e.g., using “ %p ”)
3. Debugging code that caches pointers to recently freed objects (which are thus indeterminate) in order to detect double free() s.
4. Some garbage collectors need to load, store, and compare possibly indeterminate pointers as part of their mark/sweep pass.
5. If a pair of pointers might alias, the simplest code would free one, check to see whether the pointers are equal, and if not, free the other.
6. A loop freeing the elements of a linked list might check the just-freed pointer against NULL as the loop termination condition. (The referenced blog post suggests use of a break statement to avoid such comparisons.)

Possible resolution for C

1. Status Quo, do nothing (BAD)
2. Eliminate Pointer Lifetime-End Zap (Reasonably **Good**)
3. Limit Pointer Lifetime-End Zap
   a. based on Storage Duration (**Good**)
   b. exempt pointers loaded using C11 atomics or inline assembly from lifetime end-zap (**Good** and Bad)
   c. Marking of Pointe Fetches (Bad)
   d. Only for Pointers Crossing Function Boundaries (Maybe Bad)

Possible resolution for C++ (red is extra C++ sol'n)

1. Status Quo, do nothing (BAD)
2. Eliminate Pointer Lifetime-End Zap (Reasonably **Good**)
3. Limit Pointer Lifetime-End Zap
   a. based on Storage Duration (**Good)**
   b. exempt pointers loaded using C11 atomics or inline assembly from lifetime end-zap (**Good** and Bad)
   c. Marking of Pointe Fetches (Bad)
   d. Only for Pointers Crossing Function Boundaries (Maybe Bad)
4. Zap Only Those Pointers Passed to delete and Similar (Bad)
5. Store All Pointers as Integers (Bad)

1. Status Quo

- No change
- People keep applying their defacto workarounds
- Systems refuse to tell compiler which functions do memory allocation or deallocation
- Prevents compiler from applying any pointer lifetime-end zap optimizations, or any optimizations, or issuing diagnostics
- Bad: existing code will start breaking when compiler optimizers become more aggressive

2. Eliminate Pointer Lifetime-End Zap Altogether

- Eliminate pointer Lifetime-end zap in the standard
- But this would eliminate optimizations and diagnostics
- This is the other end of the spectrum
- WG21 SG12 likes this
- WG14 also may like this
- Can we work out all the checks
- Reasonably Good: but lose all the defensive NULL check strategies

3a. Limit Pointer Lifetime-End Zap Based on Storage Duration

- Concurrent use cases only involve allocated storage duration objects
- Compiler NULL'ing of pointers at lifetime end involve only automated storage duration objects
- So limit lifetime end zap to automatic storage duration (and maybe Thread-local storage)
- Accommodates all well known concurrent use cases and many single threaded use cases
- Good: seems consistent with programming and compiler practice

3b. Limit Pointer Lifetime-End Zap Based on pointer loads with C11 atomics or inline assembly

- WG21 SG12 member suggested that pointers loaded using C11 atomics or inline assembly be exempted from pointer lifetime-end zap
  - Also add volatile loads and stores
- Accommodate all current concurrent use cases
- But lock-based algorithm involving pointer validation may not work
- Also require adding language to define information flow to the standard
- Good and Bad: unclear whether it covers all cases, and we'd need to define information flow in the standard – remember consume?

3c. Limit Pointer Lifetime-End Zap Based on Marking of Pointer Fetches

- WG14 member suggested adding an attribute to mark pointer fetches
- WG21 member suggested using std::launder
- Good for new code, but not for existing code
- Minimal effect on compiler optimizations and diagnostics
- Bad: no practical way to find all that code

3d. Limit Pointer Lifetime-End Zap to Pointers Crossing Function Boundaries

- WG14 member suggest that load store compare cast to indeterminate pointers inside a function is fine
- But touching them that have crossed a function-call boundary would be subject to lifetime-end zap
- Maybe bad: HP example shows it is a big restriction

4. Zap Only Those Pointers Passed to `delete` and Similar

- Pointers actually passed to deallocators, for example, in delete p, the pointer p, become invalid, but other copies of that pointer, even those within the same function, are unaffected.
- OK: you get some protection from compiler
- Bad: delete is an operator but C-language free is a function

5. Store All Pointers as Integers

- Can still get into trouble if zombies appear just after conversion from integer to pointer
- Adds complexity to concurrent code that is already perceived as complex
- Most developers will perceive converting pointers to integers as unnatural act
- Current implementations nevertheless affect integers (are these bugs?)
- https://github.com/paulmckrcu/wg21-p1726-examples/blob/master/lifo-push/lifo-push-uintptr.cpp
- Bad: cognitive overload on user code in most complex concurrent cases


Defending Your Code From Zombie Pointers

We hope to change the standard, but what to do in the meantime?

- Where possible, hide your allocation functions from the compiler
- Use non-standard extensions such as inline assembly to keep the compiler in the dark and the zombies buried
- When writing new code, use defensive programming
- Keep this possibility in mind when chasing obscure pointer bugs
- Inform us of additional valuable use cases vulnerable to zombie pointers
- We encourage sanity tools that detect zombie pointers in existing code

References

- WG14: Pointer lifetime-end zap http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf
- C DR 260: http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm
- WG21 D1726R1: Pointer lifetime-end zap