Generic Pointers

Generic Data Structures

Stacks

- We defined stacks of strings

- But,
  - the code for stacks of ints would be identical except for string changed to int

  - the code for stacks of *any type* would be identical except for string changed to this type

  - …

```
/*********** Implementation **********/

typedef struct list_node list;
struct list_node {
  string data;
  list* next;
};

typedef struct stack_header stack;
…

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, string x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

string pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Stacks

- Each time we need a stack for a new element type, we need to make a **copy** of the stack library

- This is **bad**
  - It's easy to make a mistake
  - We need to come up with new names
    - int_stack_t, int_stack_empty, …
    - int_list, int_list_node, is_int_segment, …
  - If we discover a bug, we need to fix it in **every copy** of the library
    - same if we discover a better implementation

- For a large application, this quickly becomes unmanageable

```
/*********** Implementation **********/

typedef struct list_node list;
struct list_node {
  string data;
  list* next;
};

typedef struct stack_header stack;
…

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, string x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

string pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Generic Data Structures

- Stacks are intrinsically **generic data structures**
  - They work the same way no matter the type of their elements
  - They do not modify elements
    - they only store them in the data structure and give them back

- We would like to implement them as a **generic library**
  - a **single** stack implementation that can be used for elements of any type
    - without copying it over and over
    - without a proliferation of function and type names
  - if we find a bug, there is **one place** where to fix it
    - if we are told of a better implementation, there is **one file** to change

Generic Stacks -- Take 1

- Here's an idea:
  - use a generic type name elem in the library
  - let the client define what elem is

- We note the type elem is to be defined by the client in the **client interface**

- The client needs to define what elem actually is in the **client definition code**

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
...

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

```
/******** Client definitions ********/

typedef string elem;
```

5

Generic Stacks -- Take 1

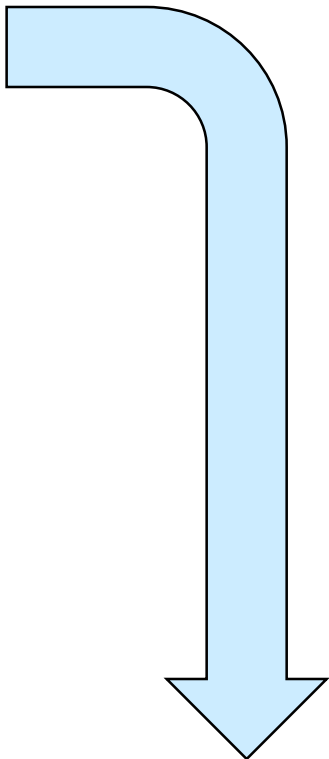**Pros:**

● A **single** library for any kind of stack

  ○ If the client needs a stack of ints in a different application,
    ➢ simply define elem as int

  ○ If another application requires a different stack type, just define elem appropriately

```
/********* Client definitions ********/

typedef int elem;
```

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
...

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Generic Stacks -- Take 1

Cons:

● Client application has to be split into **two** files

  ➢ Client definition file

  ➢ Rest of the client application

  ○ because
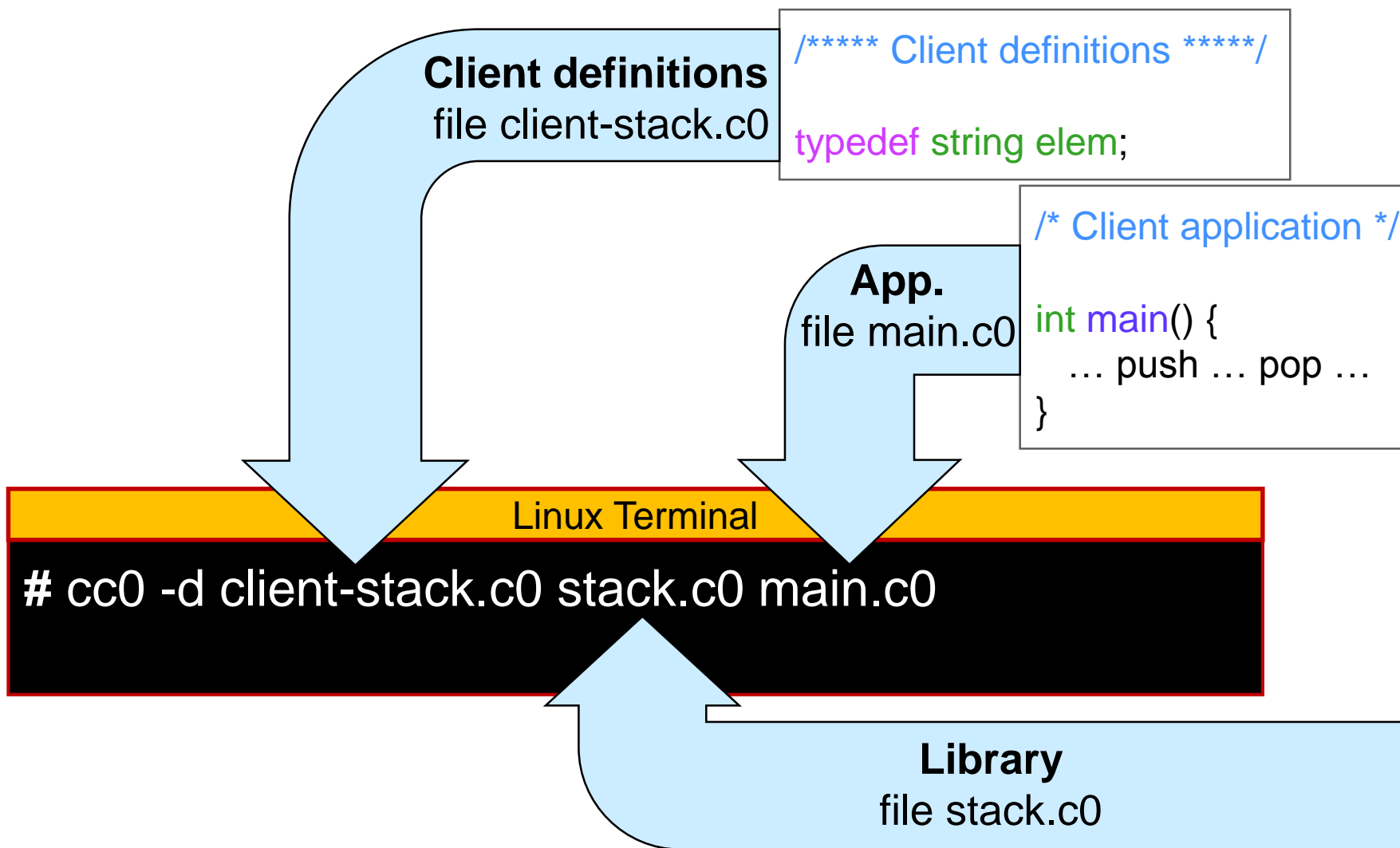
    ➢ the library needs elem to be defined
      ❑ This must occur **before** the library
    ➢ the client application needs the types and functions provided by the library to be defined
      ❑ This must occur **after** the library

This is mildly annoying

```
/*****     Client definitions *****/

typedef string elem;
```

```
/* Client application */

int main() {
   … push … pop …
}
```

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
 elem data;
  list* next;
};

typedef struct stack_header stack;
…

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Generic Stacks -- Take 1

Client definitions
file client-stack.c0

```
/***** Client definitions *****/

typedef string elem;
```

App.
file main.c0

```
/* Client application */

int main() {
    … push … pop …
}
```

Linux Terminal

 # cc0 -d client-stack.c0 stack.c0 main.c0

Library
file stack.c0

- This forces an **unnatural compilation pattern**

This is mildly annoying

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
    elem data;
    list* next;
};

typedef struct stack_header stack;
…

/*************** Interface ***************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Generic Stacks -- Take 1

Cons:

● Client application can contain at most **one type** of stacks

> This is a big deal

○ no way to have both a stack of strings and a stack of ints in the **same** application

➢ but we can have multiple stacks of ints

○ because there can be only **one definition** for elem

```
/***** Client definitions *****/

typedef string elem;
typedef int      elem;
```

?

The compiler won't know which elem to use when

Compilation error!

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
...

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Generic Stacks -- Take 1

Summary

- **Pros:**
  - A single library for any kind of stacks

- **Cons:**
  - Client application is split into **two** files
    - Unnatural compilation pattern
  - Client application can contain at most **one type** of stacks

> This is mildly annoying

> This is a big deal

```
/*********** Client Interface **********/
// typedef _____ elem;

/*********** Implementation **********/
typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
...

/*************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Can we do Better?

- Not in C0 …

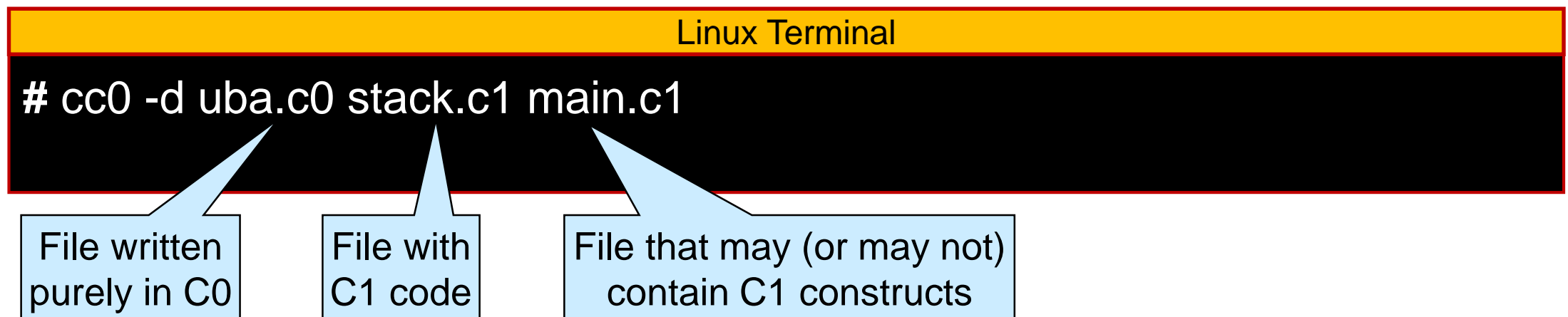- … but the language C1 extends C0 with a mechanism to address these issues

C1

The language C1

- C1 is an **extension** of C0
  - Every C0 program is a C1 program

- C1 provides two additional mechanisms
  - Generic pointers
  - Function pointers

  Both help with genericity

- Right now, we will only examine generic pointers

Running C1 Programs

- C1 programs are compiled with cc0
  - but C1-only constructs are only allowed in files with a **.c1** extension
    - C0-only code can appear in files with either a .c0 or a .c1 extension

- Example

Linux Terminal
cc0 -d uba.c0 stack.c1 main.c1

File written purely in C0

File with C1 code

File that may (or may not) contain C1 constructs

- The coin interpreter does not currently support C1 constructs
  - no way to experiment with them in coin

Generic Pointers

void*

This is **not** a pointer to void: void is not a type

Blame C for the confusing name!

- C1 provides a new **pointer type**: void*

  void* q; → q is a variable of type void*

  ○ a value of this type is a **generic pointer**

- Any pointer can be turned into a void* using a **cast**

  int* p = alloc(int);

  p = 7; Cast p to void

  q = (void*)p; → q still has type void*, but contains the same address of p

  ○ and later back to its original type

  int* r = (int*)q;

  Cast q to int*

  q still has type void*, but r contains the same address as p

16

void*

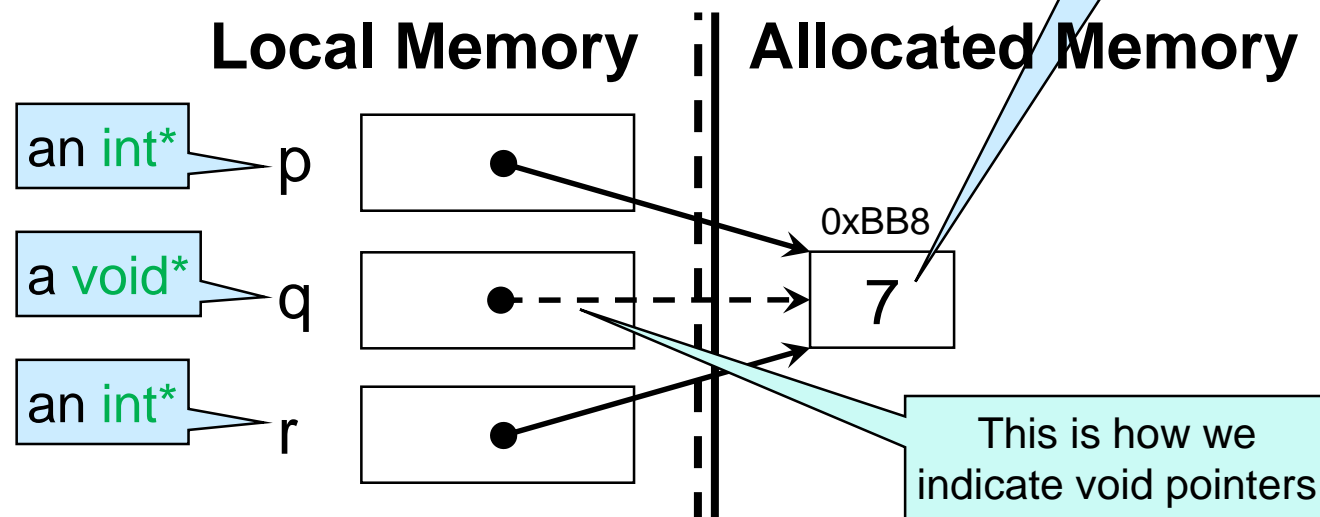- p, q and r contain the same address
  - they are aliases
- but
  - p and r have type int*
  - q has type void*

- With casting, we can pretend that a **specific** pointer (e.g., an int*) is a **generic** pointer (void*)
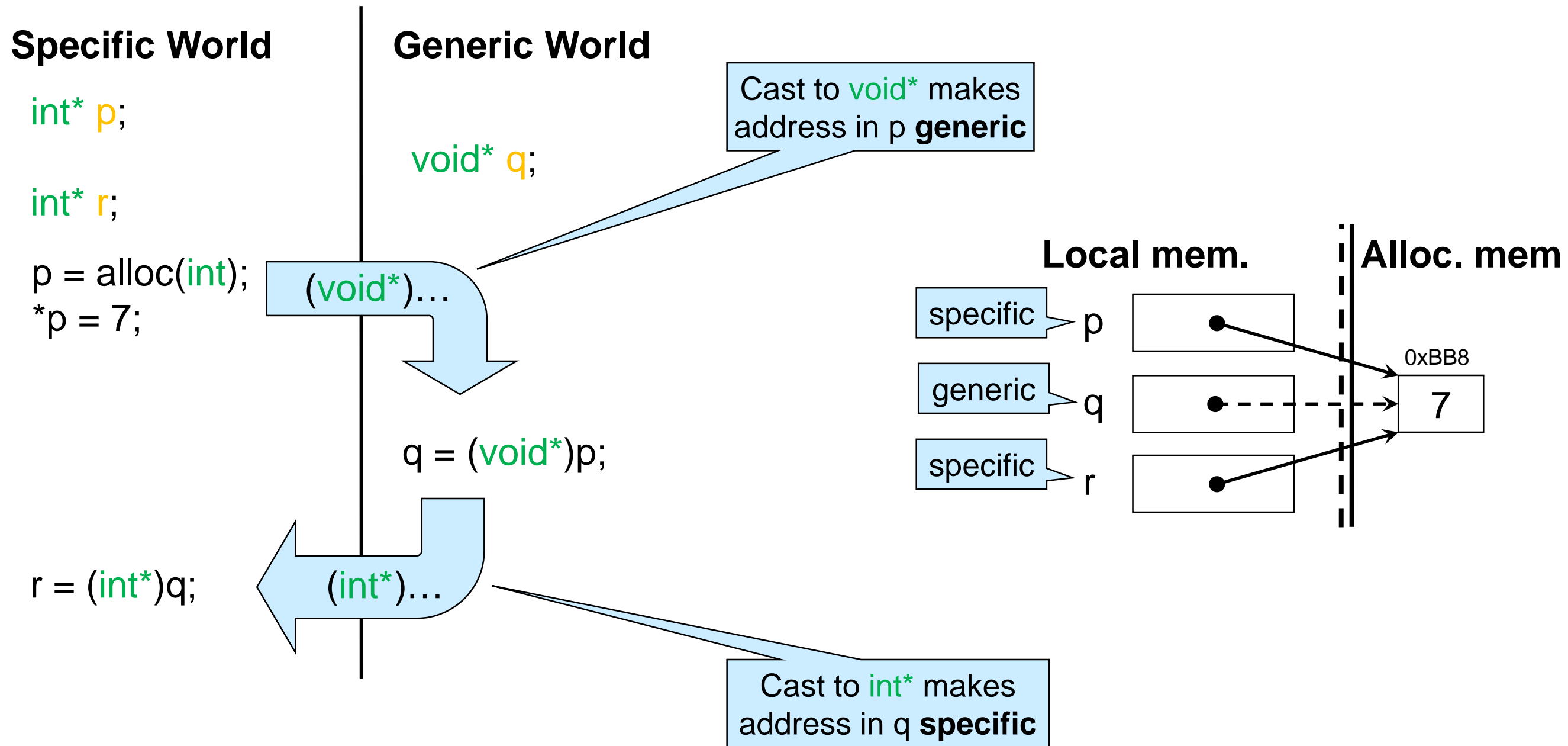 - a controlled way for a pointer to have two types
  - *only for pointers*

Memory view using addresses for pointers

Local Memory | **Allocated Memory**

an int* → p | 0xBB8

a void* → q | 0xBB8

an int* → r | 0xBB8

0xBB8
7

Cell contains an int
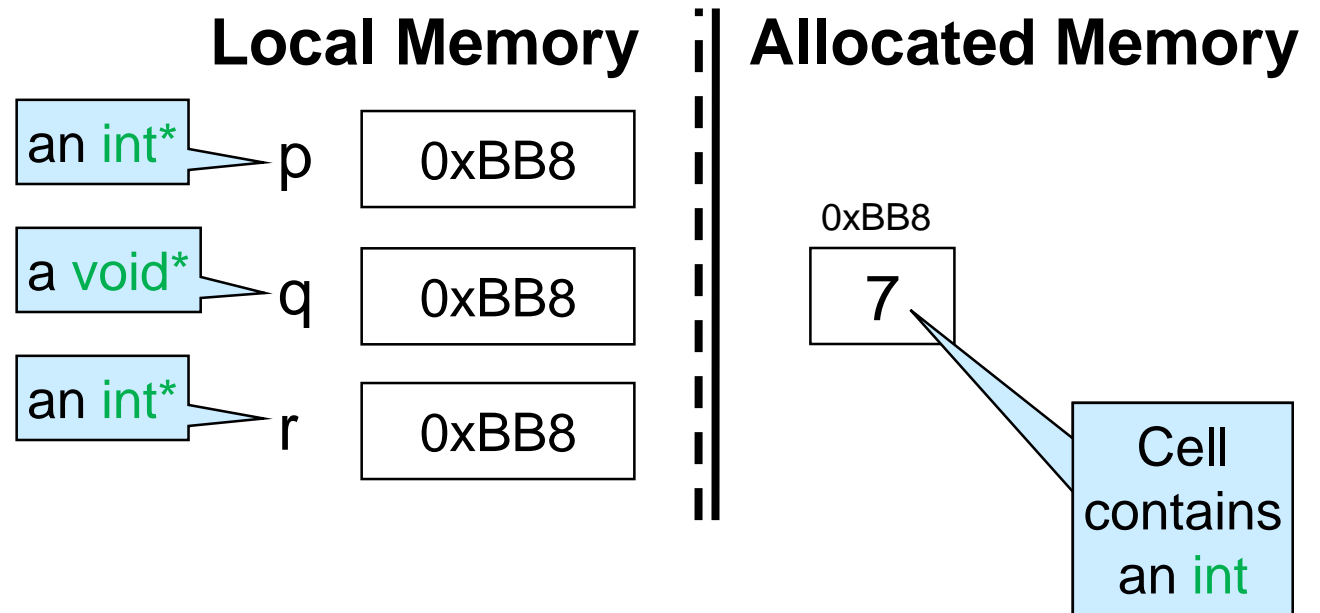
Memory view using arrows for pointers

Local Memory | **Allocated Memory**

an int* → p

a void* → q

an int* → r

0xBB8
7

This is how we indicate void pointers

17

The Specific/Generic Divide

Specific World | **Generic World**

int* p;

void* q;

int* r;

p = alloc(int);
*p = 7;

(void*)…

Cast to void* makes address in p **generic**

q = (void*)p;

r = (int*)q;

(int*)…

Cast to int* makes address in q **specific**

Local mem. | **Alloc. mem**

specific → p

0xBB8

generic → q

7

specific → r

18

What can we do with void* Pointers?

Allowed	**Not Allowed**

Allowed

- Cast to original type
  ```
  int* p = alloc(int);
  void* q1 = (void*)p;
  int* r = (int*)q1;        ✓
  ```

- Compare for equality
  ```
  void* q2 = (void*)alloc(int);
  if (q1 == q2) printf("same");   ✓
  ```

- Assign to a void* variable
  ```
  ➢ void* q3 = q1;    ✓
  ```

Not Allowed

- Dereference
  ```
  void x = *q1;    ✗
  ```
  ○ void is **not** a type in C0/C1/C

- Allocate
  ```
  void* q4 = alloc(void);    ✗
  ```
  ○ void is not a type

- Cast to type other than original
  ```
  printf("%s\n", (string*)q1);    ✗
  ```
  ○ *(see next)*

19

Safety of Generic Pointers
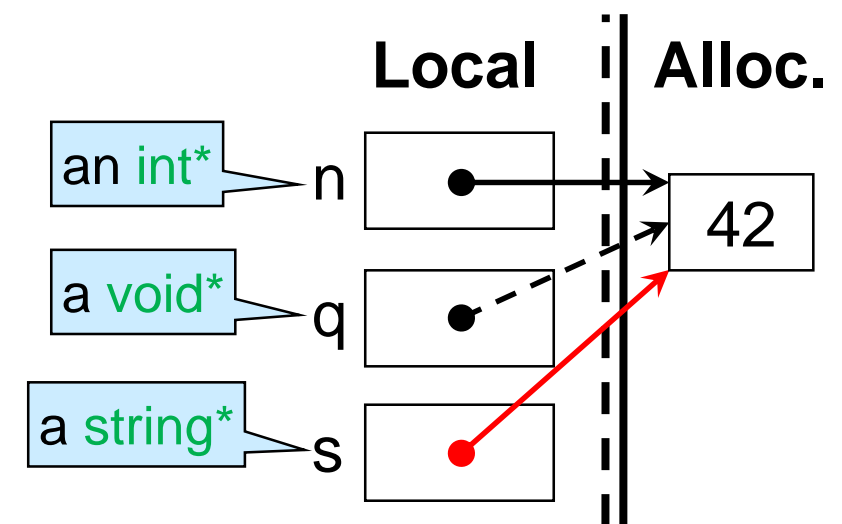
Casting back to the Wrong Type

```
int main() {

  int* n = alloc(int);
  *n = 42;

  void* q = (void*)n;

  string* s = (string*)q;

  print(*s);

  return 0;
}
```

n is an int*

q is a void* that secretly points to an int*

this turns an int* into a string* ??

What ????

Local   **Alloc.**

an int* → n

a void* → q

a string* → s

42
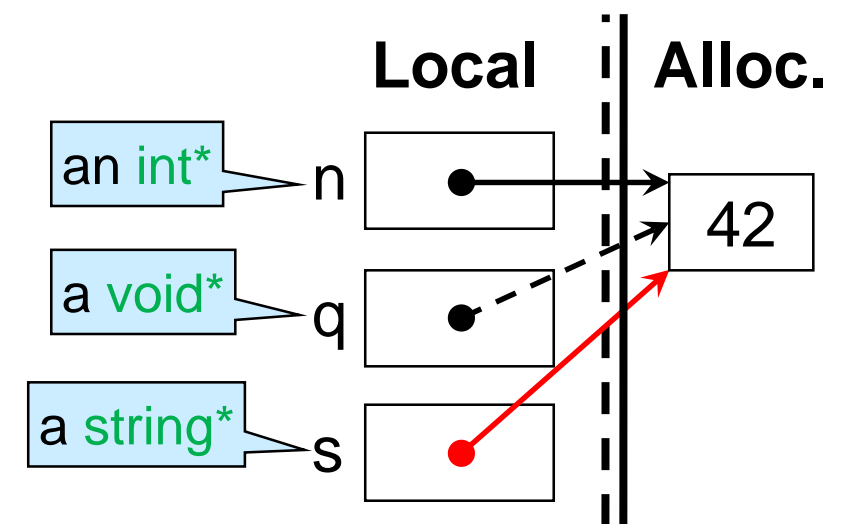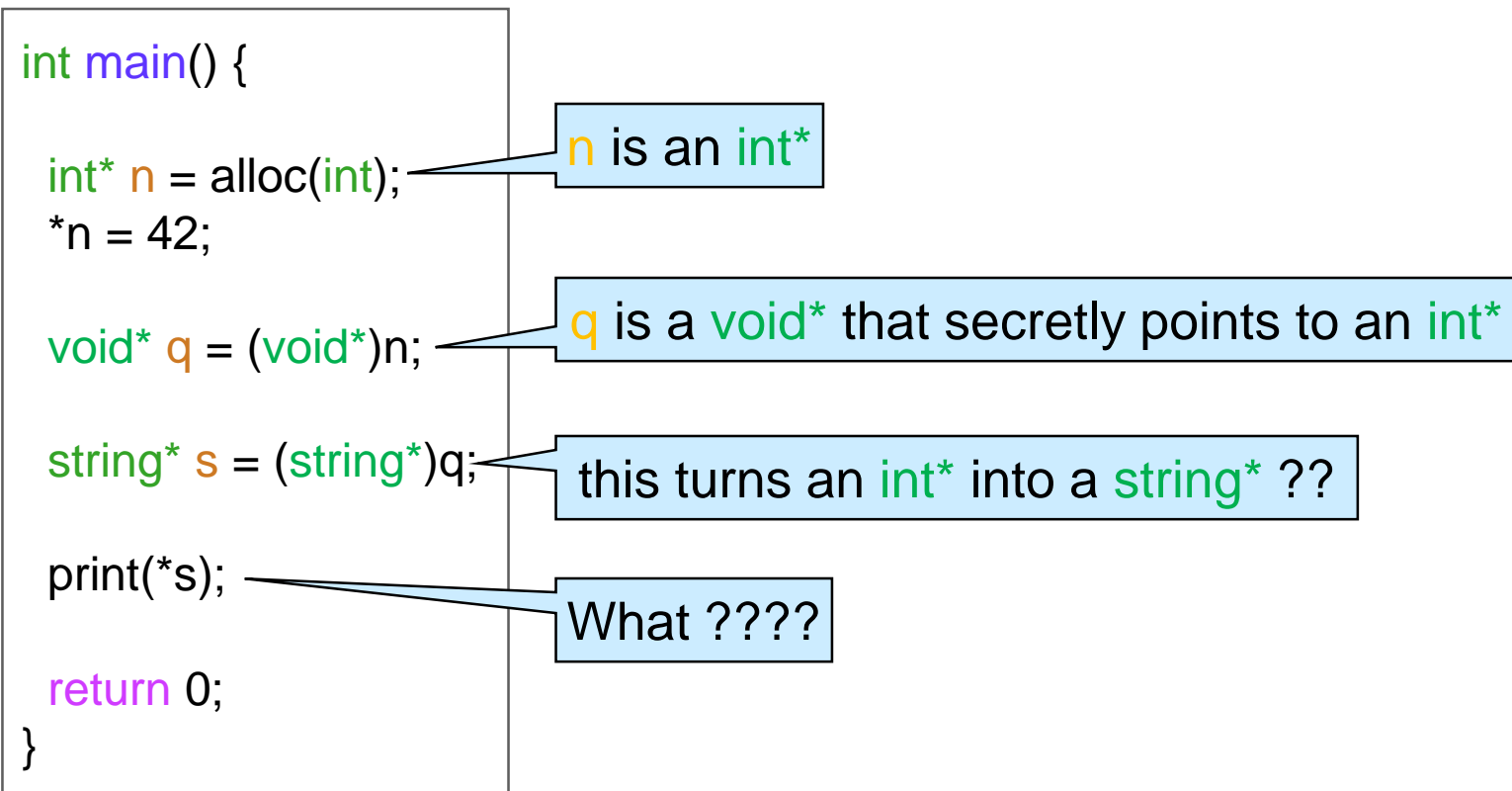
● This makes no sense!!!

○ dereferencing s, we get to an int (42)

○ but print expects a string

This doesn't feel right

➢ a safety violation maybe?

21

Casting back to the Wrong Type

```
int main() {

    int* n = alloc(int);
    *n = 42;

    void* q = (void*)n;

    string* s = (string*)q;

    print(*s);

    return 0;
}
```

n is an int*

q is a void* that secretly points to an int*

this turns an int* into a string* ??

What ????

Local   **Alloc.**

an int*   n

a void*   q

a string*   s

42

- Let's run it

Linux Terminal

```
# cc0 -d bad-casting.c1
# ./a.out
untagging pointer failed
Segmentation fault (core dumped)
```

- We get a memory error
- This **is** a safety violation

Tags

```
int main() {

  int* n = alloc(int);         ┤ n is an int*
  *n = 42;

  void* q = (void*)n;          ┤ q is a void* that secretly point...

  string* s = (string*)q;      ┤ this turns an int* into a string* ...

  print(*s);                   ┤ What ????

  return 0;
}
```
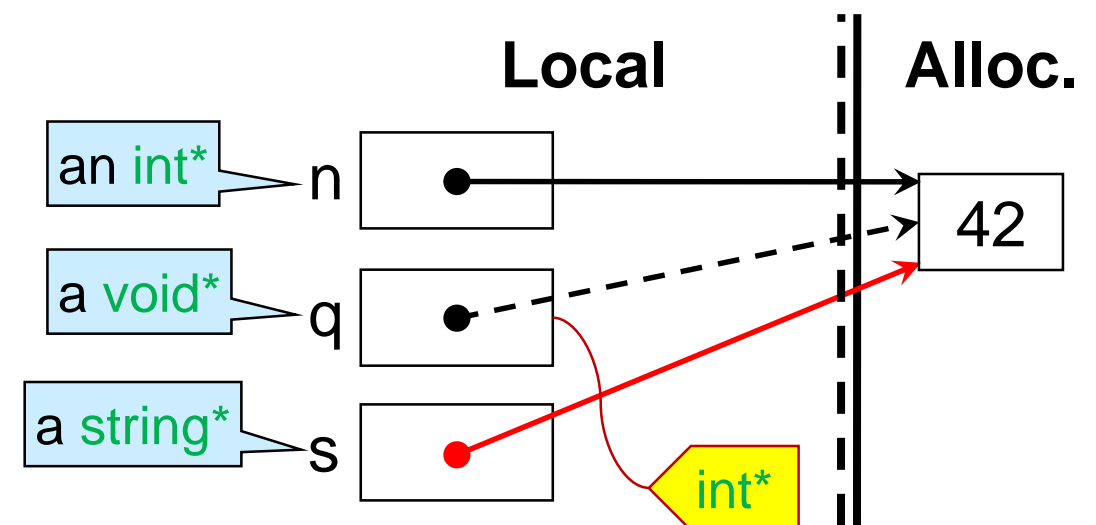
```
Linux Terminal
# cc0 -d bad-casting.c1
# ./a.out
untagging pointer failed
Segmentation fault (core dumped)
```

- *Untagging pointer failed?*
  - At run time, values of type void* carry a **tag** that records the original type of the pointer
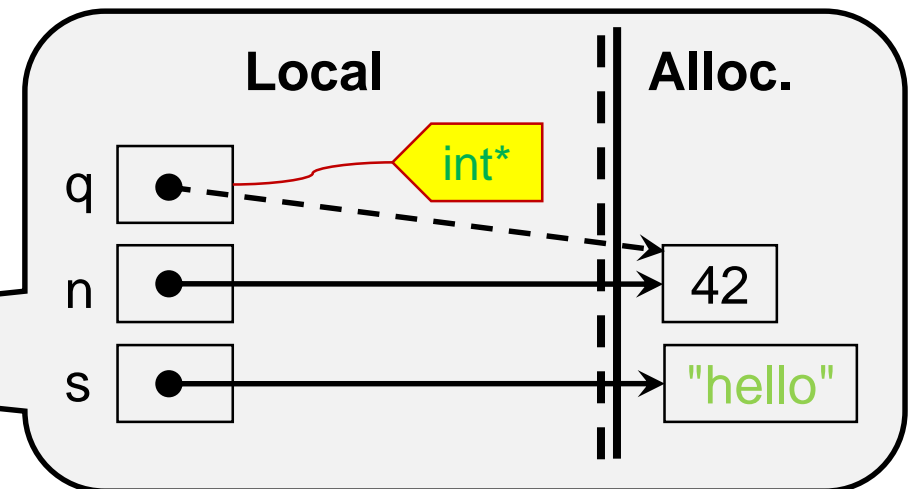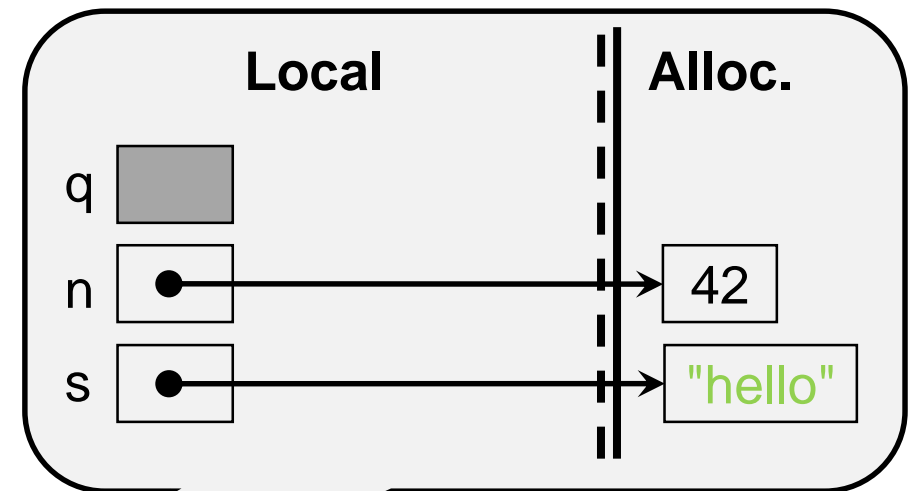 - C1 checks that the tag is correct before casting back

23

Tags

- The tag of a void* changes as execution proceeds

```
int main() {
  void* q;

  int* n = alloc(int);
  *n = 42;

  string* s = alloc(string);
  *s = "hello";

  q = (void*)n;
  printf("%d\n", *(int*)q);

  q = (void*)s;
  printf("%s\n", *(string*)q);

  return 0;
}
```
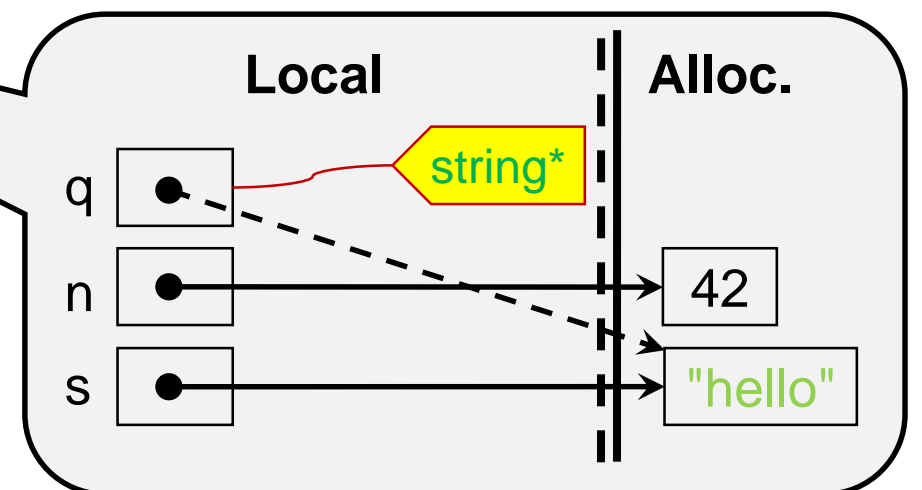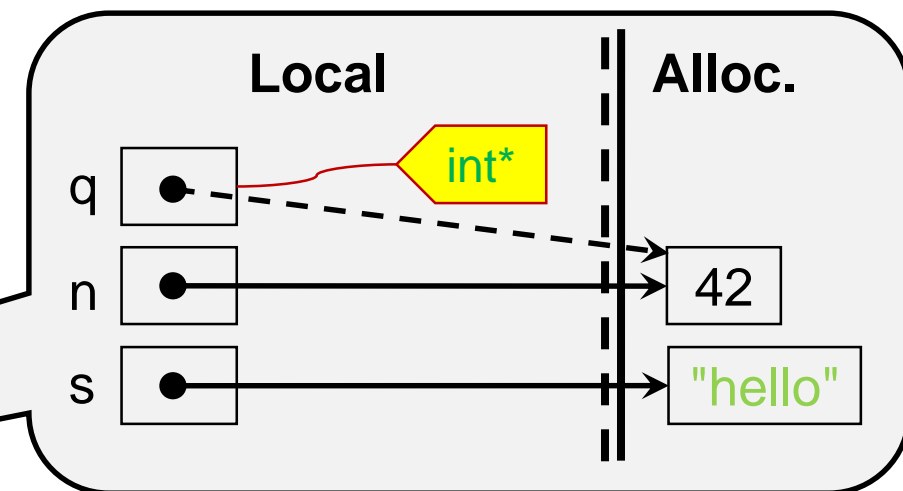
\hastag

- Annotation-only function \hastag(tp*, ptr) can be used to check that generic pointer ptr has type tp* *in debugging mode*
  - tp* cannot be void*

```
int main() {
  void* q;

  int* n = alloc(int);
  *n = 42;

  string* s = alloc(string);
  *s = "hello";

  q = (void*)n;
  //@assert \hastag(int*, q);
  printf("%d\n", *(int*)q);

  q = (void*)s;
  //@assert \hastag(string*, q);
  //@assert !\hastag(int*, q);
  printf("%s\n", *(string*)q);

  return 0;
}
```
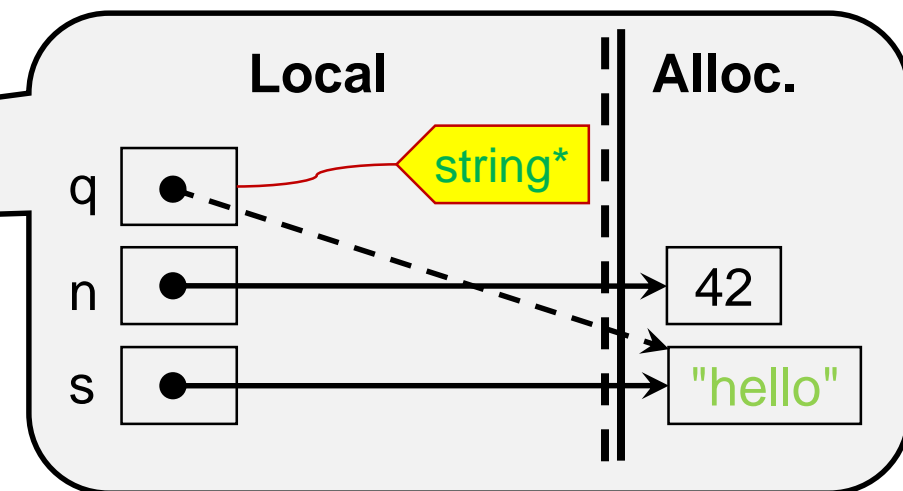
Checks that q has tag int*

Local **Alloc.**

q   int*

n   42

s   "hello"

Local **Alloc.**

q   string*

n   42

s   "hello"

Checks that q now has tag string*

Checks that q does not have tag int*

25

\hastag

- *Annotation-only function \hastag(tp*, ptr) can be used to check that generic pointer ptr has type tp* in debugging mode*

```
int main() {

  int* n = alloc(int);
  *n = 42;

  void* q = (void*)n;
  //@assert \hastag(int*, q);

   //@assert \hastag(string*, q);
  string* s = (string*)q;

  printf("%s\n", *s);

  return 0;
}
```
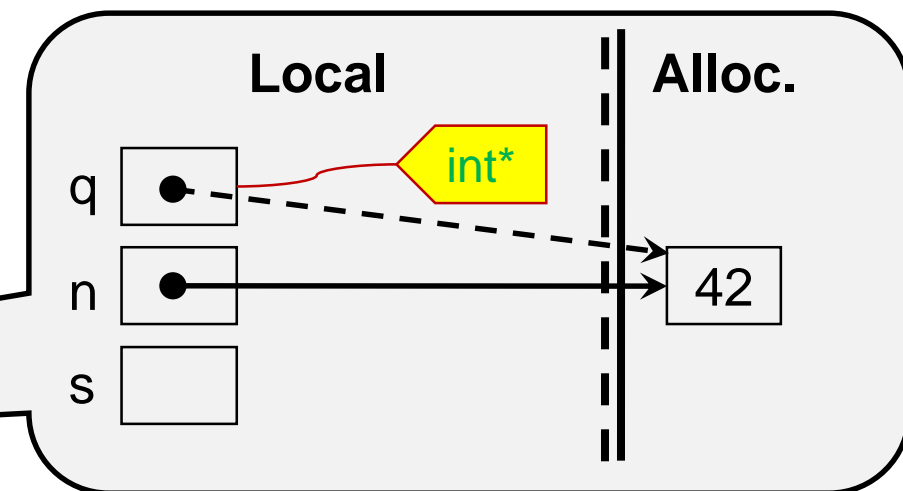
Local   **Alloc.**

q

int*

n   42

s

Checks that q has tag int* ✓

Checks that q has tag string* ✗

- Use \hastag before casting a void* back to a specific type

NULL

- NULL is a pointer of any type, including void*
  - We can cast NULL back and forth as we please

    int* p = NULL;

    void* q = (void*)p;

    string* r = (string*)q;　　　　This is legal because q is NULL

  - or do even wilder things

    void* q = NULL;

    void* r = (void*)(int*)(void*)(string*)q;　　This is legal because q is NULL

- A NULL variable of type void* has *every* tag

    void* v = NULL;

    //@assert \hastag(int*, v);　　This is legal because v is NULL

    //@assert \hastag(string*, v);

  - except void*

    //@assert \hastag(void*, v);　　This causes a compilation error

27

Contracts of Cast Operations

- Casts are **potentially unsafe** operations over pointer expressions
  - With \hastag, we can write contracts for them

- Casting from specific to generic type
  - (void*)x where x was declared of type tp*

    (void*)x
    //@ensures \hastag(tp*, \result);

- Casting from generic to specific type
  - (tp*)q where q was declared of type void*

    (tp*)q
    //@requires \hastag(tp*, q);

Generic Stacks in C1

Generic Stacks

Use void* as the type of the elements

- **Pros:**
  - Simple change to the library

    `typedef void* elem;` ← That's it!
  - A single library for any kind of stacks

- **Cons:**
  - Stack elements must be pointers
    - We cannot have a stack of ints
    - We need to turn them into int*
  - This is the best we will be able to do
    - genericity is limited to pointers
    - not just in C1, but also in C

```
/*********** Implementation **********/
typedef void* elem;        // Element type

typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
...

/************** Interface **************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Converting an int* Stack to Generic

- Cast elements to void* when pushing
- Cast them back to int* when popping

```
int main() {
  stack_t l = stack_new();

  int* x = alloc(int);
  *x = 42;
  push(l, x);




  int* y = alloc(int);
  *y = 15122;
  push(l, y);



  int* z = pop(l);
  printf("%d\n", *z);

  printf("%d\n", *pop(l));

  return 0;
}
```

push now expects a void* ;
cast x to a temp variable of type void*

Same thing, done inline

pop returns a void*, but z has type int*

Same thing, inline

main.c1

```
int main() {
  stack_t l = stack_new();

  int* x = alloc(int);
  *x = 42;
  void* p = (void*)x;
  push(l, p);




  int* y = alloc(int);
  *y = 15122;
  push(l, (void*)y);



  int* z = (int*)pop(l);
  printf("%d\n", *z);

  printf("%d\n", *(int*)pop(l));

  return 0;
}
```

31

Compilation

Application
file main.c1

Linux Terminal

```
# cc0 -d stack.c1 main.c1
```

Library
file stack.c1

No need for a client-stack.c1 file!

```c
int main() {
  stack_t l = stack_new();

  int* x = alloc(int);
  *x = 42;
  void* p = (void*)x;
  push(l, p);

  int* y = alloc(int);
  *y = 15122;
  push(l, (void*)y);

  int* z = (int*)pop(l);
  printf("%d\n", *z);

  printf("%d\n", *(int*)pop(l));

  return 0;
}
```

```c
/*********** Implementation ***********/
typedef void* elem;      // Element type

typedef struct list_node list;
struct list_node {
  elem data;
  list* next;
};

typedef struct stack_header stack;
…

/************** Interface ***************/

// typedef _____* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Converting an int Stack to Generic

- ● No way to store an int into a generic stack
  - ○ We need to convert elements to int* first
    - ➤ And cast them to void* to use the stack

This is annoying
… but that's the best we can do

main.c0

```
int main() {
  stack_t l = stack_new();



  push(l, 42);



  int y = 15122;
  push(l, y);

  int z = pop(l);
  printf("%d\n", z);

  printf("%d\n", pop(l));

  return 0;
}
```

We must store 42 in allocated memory, and
cast its pointer to a temp variable of type void*

We can inline the cast, but not the allocation

pop returns a void*, but z has type int*

Same thing, inline

main.c1

```
int main() {
  stack_t l = stack_new();

  int* x = alloc(int);
  *x = 42;
  void* p = (void*)x;
  push(l, p);

  int* y = alloc(int);
  *y = 15122;
  push(l, (void*)y);

  int z = *(int*)pop(l);
  printf("%d\n", *z);

  printf("%d\n", *(int*)pop(l));

  return 0;
}
```

Using two Stacks of Different Type in C0

… in the same application

- ○ We need to have two copies of the stack library
  - ➤ *int_stack* for ints and *str_stack* for strings

main.c0

```
int main() {
  int_stack_t I = int_stack_new();    // a stack of ints
  int_push(I, 42);
  int y = 15122;
  int_push(I, y);
  int z = int_pop(I);
  printf("%d\n", z);
  printf("%s\n", int_pop(I));

  str_stack_t S = str_stack_new();  // a stack of strings
  str_push(S, "hello");
  string s = "world";
  str_push(S, s);
  string w = str_pop(S);
  printf("%s\n", w);
  printf("%s\n", str_pop(S));

  return 0;
}
```

Using two Stacks of Different Type in **C1**

… in the same application

- o The one generic stack library is enough
- o but we need to convert elements to be pointers

```
main.c1
```

```
int main() {
 stack_t I = stack_new();  // a stack for ints

 int* x = alloc(int);
 *x = 42;
 void* p = (void*)x;
 push(I, p);

 int* y = alloc(int);
 *y = 15122;
 push(I, (void*)y);

 int z = *(int*)pop(I);
 printf("%d\n", z);

 printf("%d\n", *(int*)pop(I));

 // continued to the right
```

```
// continued from left

 stack_t S = stack_new();  // a stack for strings

 string* s1 = alloc(string);
 *s1 = "hello";
 push(S, (void*)s1);

 string* s = alloc(string);
 *s = "world";
 push(S, (void*)s);

 string w = *(string*) pop(S);
 printf("%s\n", w);
 printf("%s\n", *(string*)pop(S));

 return 0;
}
```

Bad Uses of Generic Stacks

- Nothing prevents pushing elements of *different* type in the same generic stack
  - *but why would you want to do that???*

```
int main() {
  stack_t X = stack_new();    // one stack

  int* i = alloc(int);
  *i = 42;
  push(X, (void*)i);      // push an int onto X

  string* s = alloc(string);
  *s = "Ouch!";
  push(X, (void*)s);      // now push a string onto X!

  string w = *(string*)pop(X);
  printf("%s\n", w);      // pop the string and print it

  printf("%d\n", *(int*)pop(I)); // pop the int and print it

  return 0;
}
```

main.c1

Don't do it

- Extremely error-prone

In general, how do we remember this element will be a string?

… and this one an int?

- There is always a cleaner way to do this

36