Reasoning About Memory Safety

• How can we have **confidence** that our code executes in a memory-safe (and correct, ideally) fashion?

• Approach: build up confidence on a function-by-function / module-by-module basis

• Modularity provides boundaries for our reasoning:
  • **Preconditions**: what must hold for function to operate correctly
  • **Postconditions**: what holds after function completes

• These basically describe a contract for using the module

• Notions also apply to individual statements (what must hold for correctness; what holds after execution)
  • Stmt #1’s postcondition should logically imply Stmt #2’s precondition
  • Invariants: conditions that always hold at a given point in a function (this particularly matters for loops)
int deref(int *p) {
    return *p;
}

Precondition?
/* requires: p != NULL 
(and p a valid pointer) */

int deref(int *p) {
    return *p;
}

**Precondition**: what needs to hold for function to operate correctly.

Needs to be expressed in a way that a person writing code to call the function knows how to evaluate.
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); } 
    return p;
}

**Postcondition?**
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}

Postcondition: what the function promises will hold upon its return.

Likewise, expressed in a way that a person using the call in their code knows how to make use of.
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}

_Precondition_?
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access?
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* ?? */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires?
(3) Propagate requirement up to beginning of function
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                    0 <= i && i < size(a) */
        total += a[i];
    return total;
}

size(X) = number of elements allocated for region pointed to by X
size(NULL) = 0

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

This is an abstract notion, not something built into C (like sizeof).
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                     0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
                     0 <= i && i < size(a) */
        total += a[i];
    return total;
}

Let’s simplify, given that a never changes.
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```
The $0 \leq i$ part is clear, so let’s focus for now on the rest.

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

How to prove our candidate invariant?

\( n \leq \text{size}(a) \) is straightforward because \( n \) never changes.
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}

What about i < n ?
What about $i < n$? That follows from the loop condition.
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

At this point we know the proposed invariant will always hold...
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

... and we’re done!
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL &&
           0 <= i && i < n && n <= size(a) */
        total += a[i];
    return total;
}

A more complicated loop might need us to use induction:
   Base case: first entrance into loop.
   Induction: show that postcondition of last statement of
             loop, plus loop test condition, implies invariant.
int sumderefer(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
/* requires: a != NULL &&
   size(a) >= n &&
   ???                        */

int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
/* requires: a != NULL && size(a) >= n && for all j in 0..n-1, a[j] != NULL */

int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}

This may still be memory **safe** but it can still have undefined behavior!
char *tbl[N]; /* N > 0, has type int */

int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()? (a) 0 <= retval < N, (b) 0 <= retval, (c) retval < N, (d) none of the above. Discuss with a partner.
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()? 
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above. 
Discuss with a partner.
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()? (a) 0 <= retval < N, (b) 0 <= retval, (c) retval < N, (d) none of the above. Discuss with a partner.
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()? (a) \( 0 \leq \text{retval} < N \), (b) \( 0 \leq \text{retval} \), (c) \( \text{retval} < N \), (d) none of the above. Discuss with a partner.
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()? (a) 0 <= retval < N, (b) 0 <= retval, (c) retval < N, (d) none of the above. Discuss with a partner.
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17; /* 0 <= h */
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
    int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
unsigned int hash(char *s) {
    unsigned int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
    unsigned int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
Memory safe languages

• You can spare yourself this work by using a memory-safe language
  • Turns "undefined" memory references into an immediate exception or program termination
  • Now you simply don't have to worry about buffer overflows and similar vulnerabilities

• Plenty to choose from:
  • Python, Java, Go, Rust, Swift, C#, …  Pretty much everything other than C/C++/Objective C