What is the output of this program?

```
#include <iostream>

int main(int argc, const char * argv[]) {
  int x;
  std::cout << x << "\n";
  return 0;
}
```

A) `0`

B) `-272632584`

C) `Nobody expects the Spanish Inquisition!`

D) Any of the above, and more

What is the output of this program?

```
#include <iostream>

int main(int argc, const char * argv[]) {
  int x;
  std::cout << x << "\n";
  return 0;
}
```

A) `0`

B) `-272632584`

C) `Nobody expects the Spanish Inquisition!`

D) Any of the above, and more

Undefined Behavior

Using an uninitialized variable is ***undefined behavior***

Undefined behavior doesn't just mean that the number is unspecified

It means that *anything can happen*

Safe languages do not have this kind of "anything can happen" rule

Loop Indexing

How many times does this loop iterate?

```
int sum_handful(int *arr, int start) {
  int sum = 0, end = start + 5;
  for (int i = start; i < end; i++)
    sum += arr[i];
  return sum;
}
```

Loop Indexing

How many times does this loop iterate?

What if `start` is `INT_MAX`?

```
int sum_handful(int *arr, int start) {
  int sum = 0, end = start + 5;
  for (int i = start; i < end; i++)
    sum += arr[i];
  return sum;
}
```

Since integer overflow is undefined behavior, a compiler is allowed to generate code to always iterate 5 times

Or not

Loop Indexing

```
__Z11sum_handfulPhi:
100000af0:    55                pushq   %rbp
100000af1:    48 89 e5          movq    %rsp, %rbp
100000af4:    48 63 c6          movslq  %esi, %rax
100000af7:    0f b6 0c 07       movzbl  (%rdi,%rax), %ecx
100000afb:    0f b6 54 07 01    movzbl  1(%rdi,%rax), %edx
100000b00:    01 ca   addl      %ecx, %edx
100000b02:    0f b6 4c 07 02    movzbl  2(%rdi,%rax), %ecx
100000b07:    01 d1   addl      %edx, %ecx
100000b09:    0f b6 54 07 03    movzbl  3(%rdi,%rax), %edx
100000b0e:    01 ca   addl      %ecx, %edx
100000b10:    0f b6 44 07 04    movzbl  4(%rdi,%rax), %eax
100000b15:    01 d0   addl      %edx, %eax
100000b17:    5d      popq      %rbp
100000b18:    c3      retq
```

clang++ -O2

Loop Indexing

How many times does this loop iterate?

```
int sum_handful(int *arr, unsigned start) {
  int sum = 0; unsigned end = start + 5;
  for (unsigned i = start; i < end; i++)
    sum += arr[i];
  return sum;
}
```

Overflow for `unsigned` is **not** undefined behavior

Loop Indexing

```
__Z11sum_handfulPhj:
100000ae0:    55          pushq    %rbp
100000ae1:    48 89 e5             movq    %rsp, %rbp
100000ae4:    31 c0       xorl     %eax, %eax
100000ae6:    83 fe fa             cmpl    $-6, %esi
100000ae9:    77 2e       ja       46 <__Z11sum_handfulPhj+0x39>
100000aeb:    89 f1       movl     %esi, %ecx
100000aed:    83 c6 05             addl    $5, %esi
100000af0:    0f b6 04 0f          movzbl  (%rdi,%rcx), %eax
100000af4:    48 8d 51 01          leaq    1(%rcx), %rdx
100000af8:    48 39 f2             cmpq    %rsi, %rdx
100000afb:    73 1c       jae      28 <__Z11sum_handfulPhj+0x39>
100000afd:    0f b6 54 0f 01       movzbl  1(%rdi,%rcx), %edx
100000b02:    01 d0       addl     %edx, %eax
100000b04:    0f b6 54 0f 02       movzbl  2(%rdi,%rcx), %edx
100000b09:    01 c2       addl     %eax, %edx
100000b0b:    0f b6 74 0f 03       movzbl  3(%rdi,%rcx), %esi
100000b10:    01 d6       addl     %edx, %esi
100000b12:    0f b6 44 0f 04       movzbl  4(%rdi,%rcx), %eax
100000b17:    01 f0       addl     %esi, %eax
100000b19:    5d          popq     %rbp
100000b1a:    c3          retq
```

clang++ -O2

Why Undefined Behavior

Behavior is undefined in C/C++ because there's a cost to run-time checks

- Invalid array access: `arr[i]`
- Use after free:
  ```
  free(x); ....
  y = malloc(10); ....
  *x = 3
  ```
- Integer overflow: `x + y`
- Use before initialization:
  ```
  int x;
  if (complicated())
    x = 1;
  if (also_complicated())
    return x;
  ```
- ...

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

Ok:

```
copy_from("hello", 5);
```

as long as **`BUFSIZE`** is ≥ 5

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

Also ok:

```
copy_from("hello", 6);
```

as long as **BUFSIZE** is ≥ 6

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

Undefined:

```
copy_from("hello", -2);
```

because `memcpy` requires a non-negative length

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

Ok:

```
copy_from("hello", 0);
```

for any **BUFSIZE**

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

Undefined:

```
copy_from(NULL, 5);
```

because `memcpy` requires non-`NULL` arguments

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

???:

```
copy_from(NULL, 0);
```

undefined because `memcpy` requires non-`NULL` arguments

Library Function Constraints

```
char buffer[BUFSIZE];

int copy_from(const char *src, int len) {
  // len bytes from scr to buffer:
  memcpy(buffer, src, len);
  return (src == NULL);
}
```

???:

```
copy_from(NULL, 0);
```

might return 0!

undefined because `memcpy` requires non-`NULL` arguments

Library Function Constraints

```
0000000000000000 <copy_from>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 63 d6                movslq %esi,%rdx
   7:   b9 10 00 00 00          mov    $0x10,%ecx
   c:   48 89 fe                mov    %rdi,%rsi
   f:   bf 00 00 00 00          mov    $0x0,%edi
  14:   e8 00 00 00 00          callq  19 <copy_from+0x19>
  19:   31 c0                   xor    %eax,%eax
  1b:   48 83 c4 08             add    $0x8,%rsp
  1f:   c3                      retq
```

gcc -O2

Undefined Behavior and Functions

Your functions inherit undefined behavior from primitives and library functions

```
void rows(unsigned count, unsigned columns) {
  return (count+columns-1) % columns;
}
```

The `rows` function is defined only for non-zero `columns`

Undefined Behavior and Programs

```
$ msdscript
2147483647
2147483647

$ msdscript
2147483647+1
-2147483648
```

Undefined Behavior and Programs

```
$ msdscript
2147483647
2147483647

$ msdscript
2147483647+1
-2147483648
```

always?

Undefined Behavior and Programs

```
class NumVal : public Val {
  int rep;
  ....
};
```

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)(rep + other_num_val->rep);
}
```

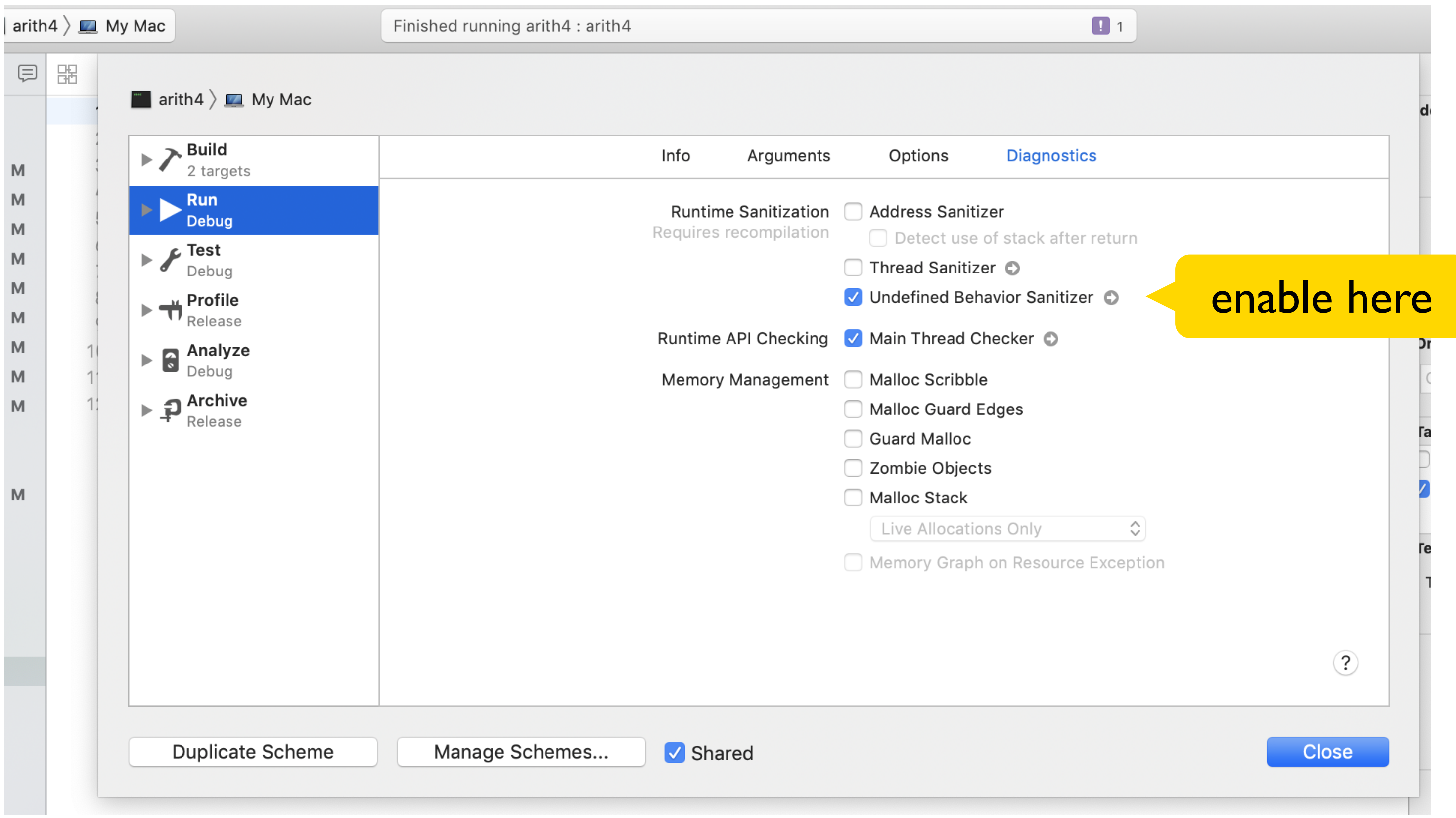
Undefined Behavior and Programs

```
class NumVal : public Val {
  int rep;
  ....
};
```

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)(rep + other_num_val->rep);
}
```

Undefined on overflow

Undefined Behavior Sanitizer in Xcode



Undefined Behavior Sanitizer in CLion / CMake

In **`CMakeLists.txt`**:

```
SET(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -fsanitize=undefined")
SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -fsanitize=undefined")
```

Undefined Behavior Sanitizer Results

```
$ msdscript
2147483647+1
..../value.cpp:26:28: runtime error: signed integer overflow:
     2147483647 + 1 cannot be represented in type 'int'
-2147483648
```

Arithmetic Solutions

Option 1: use floating-point

```
class NumVal : public Val {
  double rep;
  ....
};
```

Option 2: use defined overflow

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}
```

Arithmetic Solutions

Option 1: use floating-point

```
class NumVal : public Val {
  double rep;
  ....
};
```

Option 2: use defined overflow

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}
```

specified to wrap around for **unsigned**

Arithmetic Solutions

Option 1: use floating-point

```
class NumVal : public Val {
  double rep;
  ....
};
```

Option 2: use defined overflow

```
PTR(Val) NumVal::add_to(PTR(Val) other_val) {
  PTR(NumVal) other_num_val = CAST(NumVal)(other_val);
  if (other_num_val == nullptr)
    throw std::runtime_error("not a number");
  else
    return NEW(NumVal)((unsigned)rep + (unsigned)other_num_val->rep);
}
```

Option 3: implement bignums

Undefined Behavior Summary

Undefined behavior is

- tricky to understand
- frequently exposed by shifting compiler optimization levels
- never ok

Safe Languages

Safe languages do not have "anything at all" undefined behavior

- Bad operations tend to become exceptions

  Example: `1/0` raises divide-by-zero

- Some operations may still have unspecified effects

  Example: multiple threads writing to a variable

Safety in Java

In Java:

- Integer arithmetic is fully specified
  - `2147483647 + 1` reliably prroduces `-2147483648`
- Use after `free` is not possible
  - there is no `free`
  - garbage collection ensures memory safety
- Object and array references cannot be misused
  - `new` is the only make to make a reference
  - Casts are always checked
  - Array bounds are always checked

Exceptions versus Segmentation Faults

What's the difference beteween

`my_array[out_of_bound_index]`

triggering a seg fault versus *raising an exception?*

- Exception is raised reliably

  ⇒ unlikely to be a security risk

- Compiler cannot rearrange computation on the assumption that there will be no problem

Types and Safety

safe ≠ *typed*

safe: no "anything can happen" undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Java: Defintely an array of `Fish`

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Java: Defintely an array of `Fish`
... or `null`

Types and Safety

safe ≠ *typed*

safe: no "anything can happen" undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

JavaScript: Some value

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Java: Defintely an index integer

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Java: Defintely an index integer
... maybe in range

Types and Safety

safe ≠ *typed*

safe: no "anything can happen" undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

JavaScript: Some key... or not

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

Definitely returns a number
that can be added to other numbers

Types and Safety

safe ≠ *typed*

safe: no "anything can happen" undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

JavaScript: Returns some value

Types and Safety

safe ≠ *typed*

safe: no "anything can happen" undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

A type system can reduce the run-time cost of safety, but run-time checks can also provide safety

Types and Safety

safe ≠ *typed*

safe: no “anything can happen” undefined behavior

typed: static guarantees about run-time behavior

```
fishes[favorite].swim()
```

“Stronger” types can provide more guarantees

e.g., `Fish` means a `Fish` instance, never `null`

Memory Safety

Historically, safety required automatic memory management:

- garbage collection
- reference counting
- static and stack only

Memory Safety

Historically, safety required automatic memory management:

- garbage collection General, but prone to pauses
- reference counting
- static and stack only

Memory Safety

Historically, safety required automatic memory management:

- garbage collection
- reference counting — Disallows cycles, somewhat prone to pauses
- static and stack only

Memory Safety

Historically, safety required automatic memory management:

- garbage collection
- reference counting
- static and stack only limited

Memory Safety

Historically, safety required automatic memory management:

- garbage collection
- reference counting
- static and stack only

A type system can also guarantee proper memory management

- **MLKit** (1980s) – inference for allocating in a stack of regions
- **Cyclone** (2000s) – regions enforced by type system
- **Rust** (2010s) – ownership tracking in type checker

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  ...
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  let s2 = s1;
  .... // can't use s1 anymore
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

s2 takes ownership

```
{
  let s1 = String::from("hello");
  let s2 = s1;
  .... // can't use s1 anymore
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(s1);
  .... // can't use s1 anymore
}

fn greet(s2 : String) {
    ....
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(s1);
  .... // can't use s1 anymore
}
```

`greet` takes ownership

```
fn greet(s2 : String) {
   ....
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  let s2 = greet(s1);
  .... // can use s2 here
}

fn String greet(s2 : String) {
    ....
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&s1);
  .... // can still use s1 here
}

fn greet(s2 : &String) {
   ....
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&s1);
  .... // can still use s1 here
}

fn greet(s2 : &String) {
   ....
}
```

borrow

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&s1);
```

borrow

```
  .... // can still use s1 here
}

fn greet(s2 : &String) {
    ....
```

cannot stash away — enforced by the *borrow checker*

```
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&s1, &s1);
  .... // can still use s1 here
}

fn greet(s2 : &String,
         s3 : &String) {
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&s1, &s1);
  .... // can still use s1 here
}

fn greet(s2 : &String,
         s3 : &String) {
}
```

immutable

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&mut s1);
  .... // can still use s1 here
}

fn greet(s2 : &mut String) {
    ....
}
```

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&mut s1, &s1);
  .... // can still use s1 here
}

fn greet(s2 : &mut String,
         s3 : &String) {
}
```

not allowed

Ownership in Rust

- Every object has a single *owner* variable
- When the owner goes away, the object goes away

```
{
  let s1 = String::from("hello");
  greet(&mut s1, &mut s1);
  .... // can still use s1 here
}

fn greet(s2 : &mut String,
         s3 : &mut String) {
}
```

not allowed

Types

Types provide guarantees about run-time behavior

Some guarantees are useful for performance