CSc 352

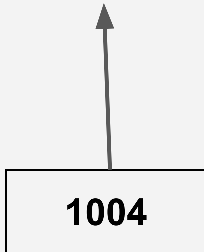
C Programming Pointers

Benjamin Dicken

Pointers

A pointer is a numeric value representing the address of a location memory

```
char x = 190;
char * xp = &x;
```



Memory Address	Sequential Memory
....	....
1001	105
1002	17
1003	32
1004	190
1005	147
1006	0
1007	100
....	....

Pointers

A pointer is a numeric value representing the address of a location memory

```
char x = 190;
char * xp = &x;
printf("%d\n", x);
printf("%p\n", &x);
printf("%p\n", xp);
```

Memory Address	Sequential Memory
....	....
1001	105
1002	17
1003	32
1004	190
1005	147
1006	0
1007	100
....	....

C Pointer

- An address to a location memory
- Access to the numeric value
- Can do *math* with that number
- For static / hard-coded values: compiler assigns pointer
- For dynamic values: malloc assigns values

Java / Py Reference

A reference is sorta like a pointer, however:

- No math on the pointer
- Less control over pointer address, dereference, etc.

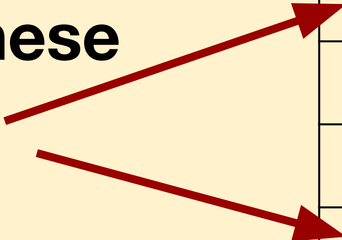
Pointers

Addresses	Memory
. . . .	. . . .
0x100000000001	0
0x100000000002	0
. . . .	0
0x10000000000A	0
. . . .	0
0x100000000013	0
0x100000000014	0
. . . .	. . . .
. . . .	. . . .

Activity

Pointers

How big are these number?



(8 bits? 16 bits? 32 bits? 48 bits? 64 bits?)

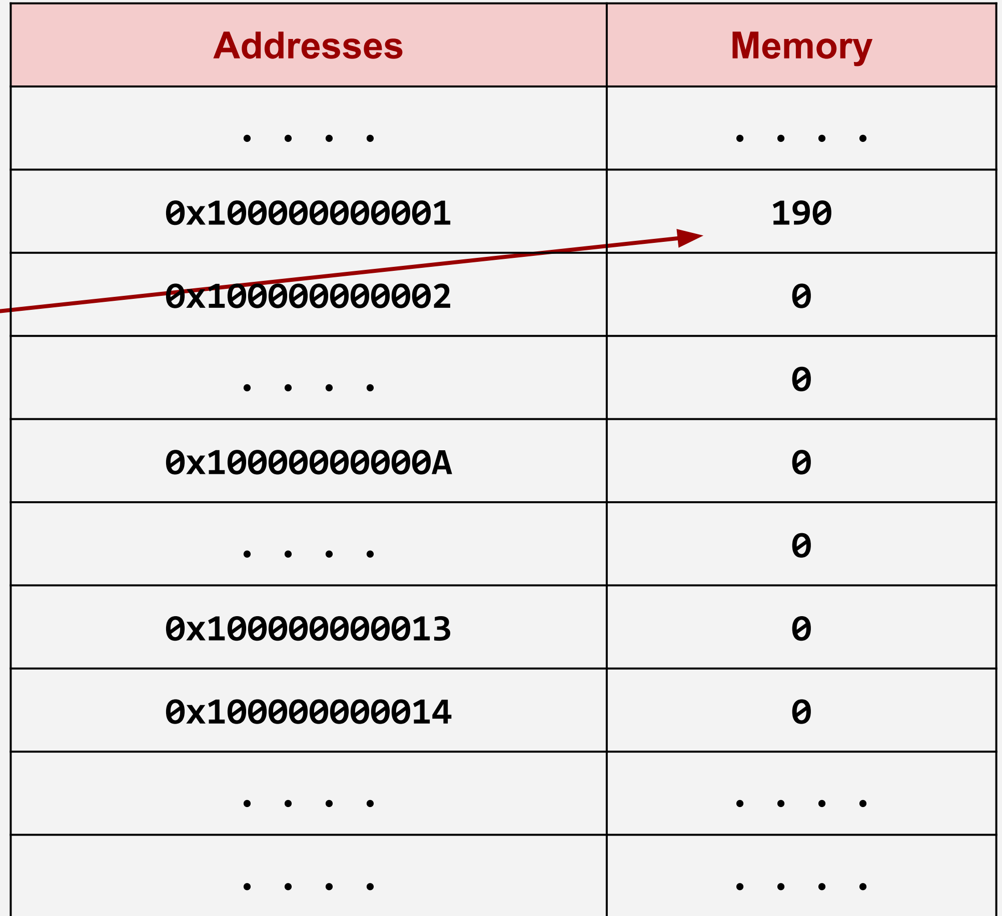
Why is it that size?

https://en.wikipedia.org/wiki/X86-64#Canonical_form_addresses

Addresses	Memory
. . . .	. . . .
0x100000000001	0
0x100000000002	0
. . . .	0
0x10000000000A	0
. . . .	0
0x100000000013	0
0x100000000014	0
. . . .	. . . .
. . . .	. . . .

Pointers

```
char x = 190;
```



Addresses	Memory
. . . .	. . . .
0x100000000001	190
0x100000000002	0
. . . .	0
0x10000000000A	0
. . . .	0
0x100000000013	0
0x100000000014	0
. . . .	. . . .
. . . .	. . . .

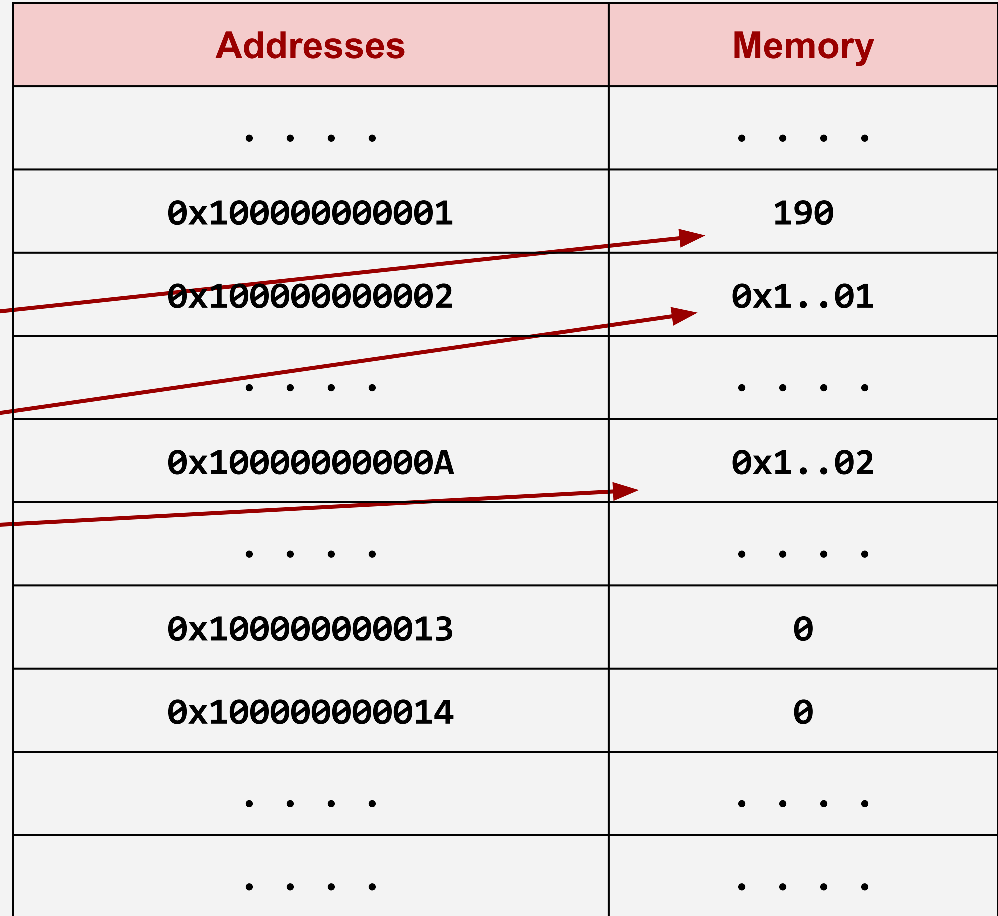
Pointers

```
char x = 190;
char * xp = &x;
```

Addresses	Memory
. . . .	. . . .
0x100000000001	190
0x100000000002	0x1..01
. . . .	. . . .
0x10000000000A	0
. . . .	0
0x100000000013	0
0x100000000014	0
. . . .	. . . .
. . . .	. . . .

Pointers

```
char x = 190;
char * xp = &x;
char ** xpp = &xp;
```



Addresses	Memory
. . . .	. . . .
0x100000000001	190
0x100000000002	0x1..01
. . . .	. . . .
0x10000000000A	0x1..02
. . . .	. . . .
0x100000000013	0
0x100000000014	0
. . . .	. . . .
. . . .	. . . .

Pointer-related operators (unary, prefix)

`*` dereferences a pointer (gives the values that the pointer points to)

`&` gets the address of a value

If **`x`** is a pointer to an int, then **`*x`** is the int itself

If **`x`** is an integer, **`&x`** is the address of that integer in memory

One is the opposite of the other

`*(&p)` is equivalent to **`p`**

What will print?

```c
#include <stdio.h>

int main() {
  int x = 50;
  int * z = &x;
  printf("%d\n", *z);
  return 0;
}
```

Activity

What will print?

```c
#include <stdio.h>

int main() {
  int x = 50;
  int ** z = &&x;
  printf("%d\n", **z);
  return 0;
}
```

Activity

What will print?

```c
int* something(
     int a, int * b) {
  int c = 40;
  a = 20;
  *b = 30;
  int * d = &c;
  return d;
}
```

```c
int main() {
  int x = 100;
  int y = 200;
  int * z = something(y, &y);
  printf("%d\n", x);
  printf("%d\n", y);
  printf("%d\n", *z);
  return 0;
}
```

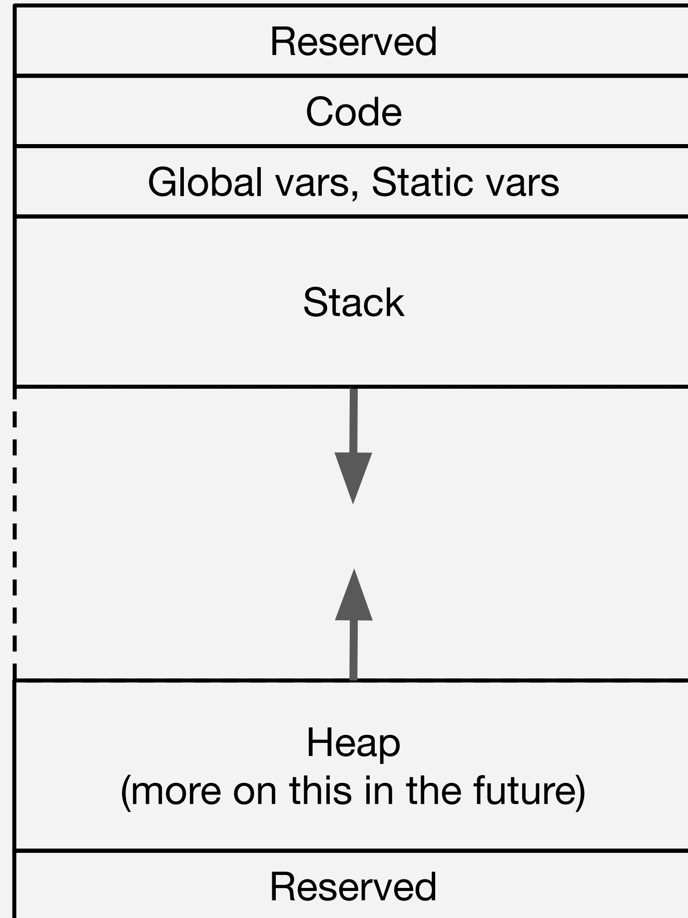
Activity

What will print?

```c
#include <stdio.h>
void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}
int * something() {
  int c = 100;
  int * d = &c;
  return d;
}
```

```c
int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```



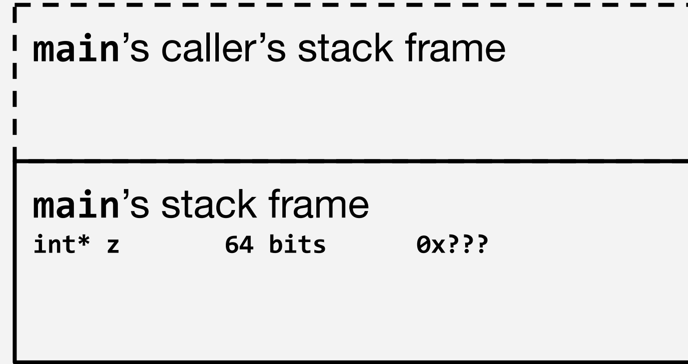
```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

Memory
Addresses

0x0…100 sp
// other vars

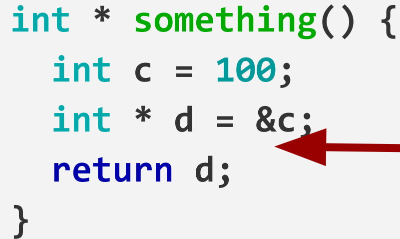
0x00…130 sp
0x00…138 z

Stack Example



Stack
growth
direction

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```



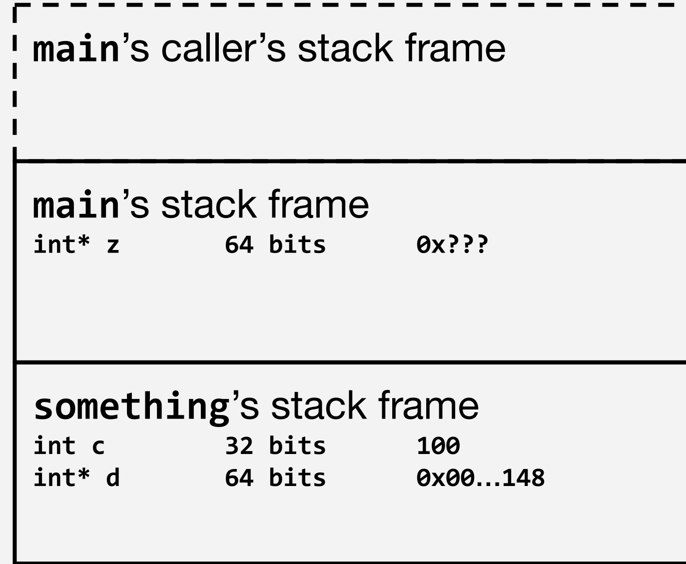
Memory
Addresses

0x0…100 sp
// other vars

0x00…130 sp
0x00…138 z

0x00…140 sp
0x00…148 c
0x00…14c d

Stack Example



Stack
growth
direction

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

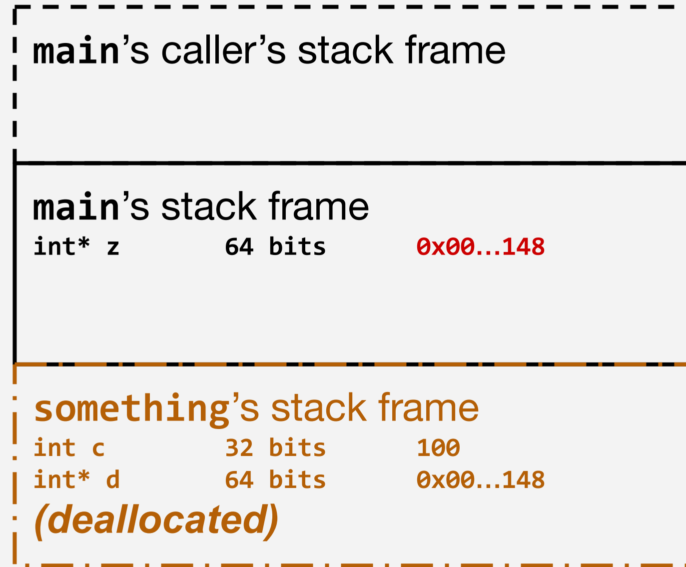
Memory
Addresses

0x0…100 sp
// other vars

0x00…130 sp
0x00…138 z

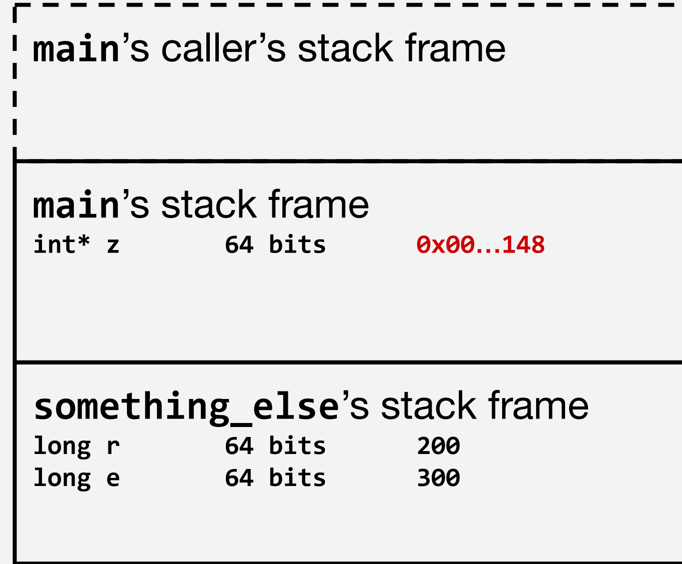
0x00…140 sp
0x00…148 c
0x00…14c d

Stack Example



Stack
growth
direction

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```



Stack growth direction

```c
#include <stdio.h>

void something_else() {
  long r = 200;
  long e = 300;
  printf("%ld, %ld\n", e, r);
}

int * something() {
  int c = 100;
  int * d = &c;
  return d;
}

int main() {
  int * z = something();
  something_else();
  printf("%d\n", *z);
  return 0;
}
```

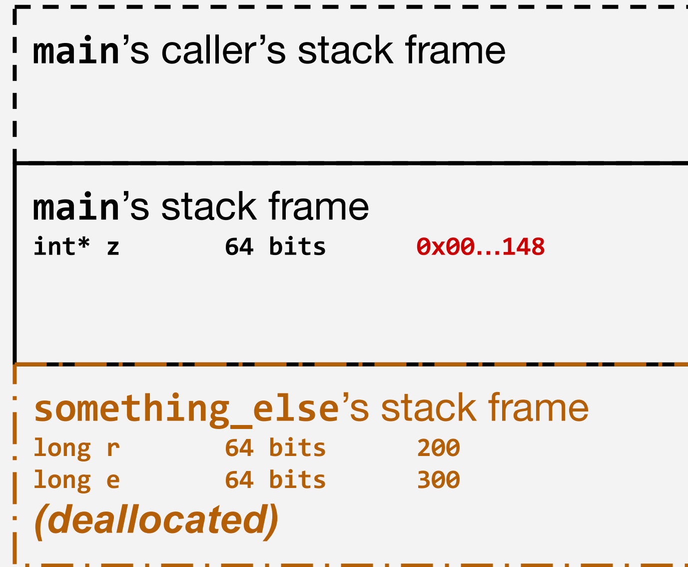
Memory Addresses

0x0…100 sp
// other vars

0x00…130 sp
0x00…138 z

0x00…140 sp
0x00…148 r
0x00…150 e

Stack Example



Stack growth direction

Uninitialized and Dangling Pointers

Uninitialized Pointer: A pointer that does not get assigned a value

- What happens when you look up a “random” address?

Dangling Pointer: Points to a location that is no longer valid

- Think: Points to a value that *was* on the stack but has been deallocated
- Think: Points to dynamically-allocated memory that has been freed

What do you think of this code?

```
int* get_id_number(char* prompt) {
  int id_number;
  int* id_number_address = &id_number;
  printf("%s", prompt);
  scanf("%d", &id_number);
  return id_number_address;
}

int main() {
  int* id = get_id_number("Enter your ID number:\n");
  printf("Your ID number is %d\n", *id);
  return 0;
}
```

L-values and R-values

- An L-value is a ***location*** and can be used on the left-hand side of an equals sign
  - Arithmetic-type variables, array elements, pointers
  - Also structs (later)
- An R-value is something that does not actually have a stored location in memory
  - Return value from function, a math expression, etc

Plus Plus

x++ yields x and increments x sometime before or at the completion of the statement it is within.

++x yields (x+1) and increments x sometime before or at the completion of the statement it is within.

Activity

```c
int x = 1;
int y = 2;
int r1 = 0;
int r2 = 0;

r1 = x++;                        // A
r2 = (x++)++;                    // B
x + y  = x + y;                  // C
*(&x) = (++y) + (r1++);          // D
*(&x) = (++y) + (x++);           // E
*(&x + y) = 10;                  // F
x++ = y++;                       // G
```

What is valid and what is not valid?

Activity

```
int x = 1;
int y = 2;
int* xp = &x;
int* yp = &y;

*(++xp) = 30;  // A
y = (*xp)++;   // B
y = *(xp++);   // C
y = *((&x)++); // D
```

What is valid and what is not valid?