The Heap, Value Semantics, References

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

pollev.com/tqm

- How are you? How is the start of the semester going for you?

Administrivia

- First Assignment (HW00 cowsay)
  - "Due" Yesterday
  - Extended to be due Tuesday the 9th (course selection period ends)
  - Mostly a C refresher
  - Don't put it off, HW01 is posted and you have to work on that too ☺
- Pre semester Survey
  - Anonymous
  - Due Friday the 12th
- HW01 Posted tonight
  - Due Tuesday the 9th
  - More rigorous C refresher ☺

Lecture Outline

- **Pointers & Const**
- The Heap
- Value Semantics
- References

const

- `const`: this cannot be changed/mutated
  - Used *much* more in C++ than in C
  - Signal of intent to compiler; meaningless at hardware level
    - Results in compile-time errors

```
void BrokenPrintSquare(const int* i) {
  *i = (*i)*(*i);  // compiler error here!
  std::cout << *i << std::endl;
}

int main(int argc, char** argv) {
  int j = 2;
  BrokenPrintSquare(&j);
  return EXIT_SUCCESS;
}
```

`const` and Pointers

- Pointers can change data in two different contexts:
  1) You can change the value of the pointer
  2) You can change the thing the pointer points to (via dereference)

```
int x;
int *p = &x;
```

- `const` can be used to prevent either/both of these behaviors!
  - `const` next to pointer name means you can't change the value of the pointer
  - `const` next to data type pointed to means you can't use this pointer to change the thing being pointed to
  - Tip: read variable declaration from *right-to-left*

```
int * const p;
```

```
const int * p;
```

`const` and Pointers

- The syntax with pointers is confusing:


```
int main(int argc, char** argv) {
  int x = 5;                  // int
  const int y = 6;            // (const int)
y++;

  const int *z = &y;          // pointer to a (const int)
*z += 1;
z++;

  int *const w = &x;          // (const pointer) to a (variable int)
*w += 1;
w++;

  const int *const v = &x; // (const pointer) to a (const int)
*v += 1;
v++;

  return EXIT_SUCCESS;
}
```

`const` Parameters

- A `const` parameter *cannot* be mutated inside the function
  - Therefore it does not matter if the argument can be mutated or not
- A non-`const` parameter *may* be mutated inside the function
  - Compiler won't let you pass in const parameters

Make parameters const when you can

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}

int main(int argc, char** argv) {
  const int a = 10;
  int b = 20;

  foo(&a);   // OK
  foo(&b);   // OK
  bar(&a);   // not OK - error
  bar(&b);   // OK

  return EXIT_SUCCESS;
}
```

Lecture Outline

- Pointers & Const
- **The Heap**
- Value Semantics
- References

Types of Memory

- There are three* main ways in which memory is allocated.

```
int counter = 0;    // global var

int main(int argc, char** argv) {
  counter++;
  cout << "count = " << counter;
  cout << endl;
  return 0;
}
```

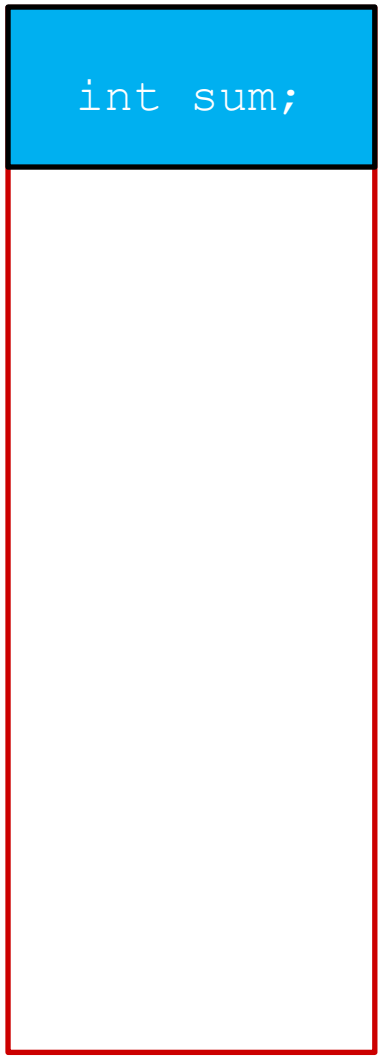
- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits
- Compiler knows exactly how many instances of it will exist across program's life.

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  cout << "y = " << y << endl;
  return 0;
}
```

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns
- Compiler knows how many there will be per function invocation and there lifetime is limited to scope of function.

Stack Example:

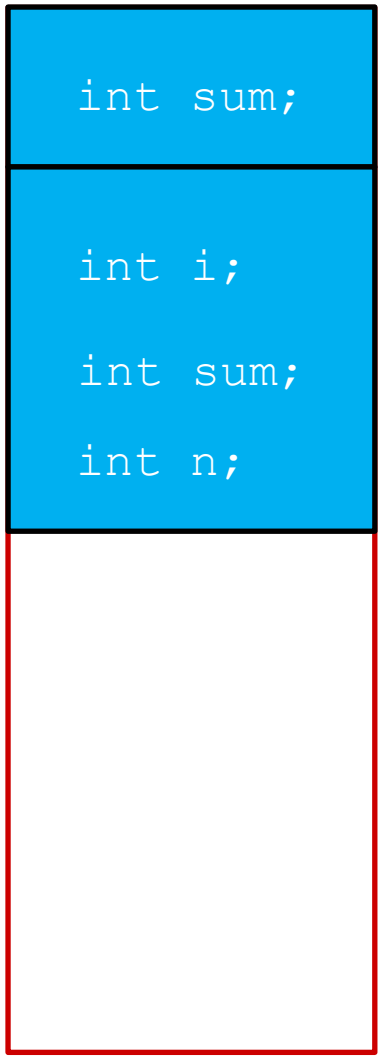
```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```

Stack frame for `main()`

Stack Example:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```

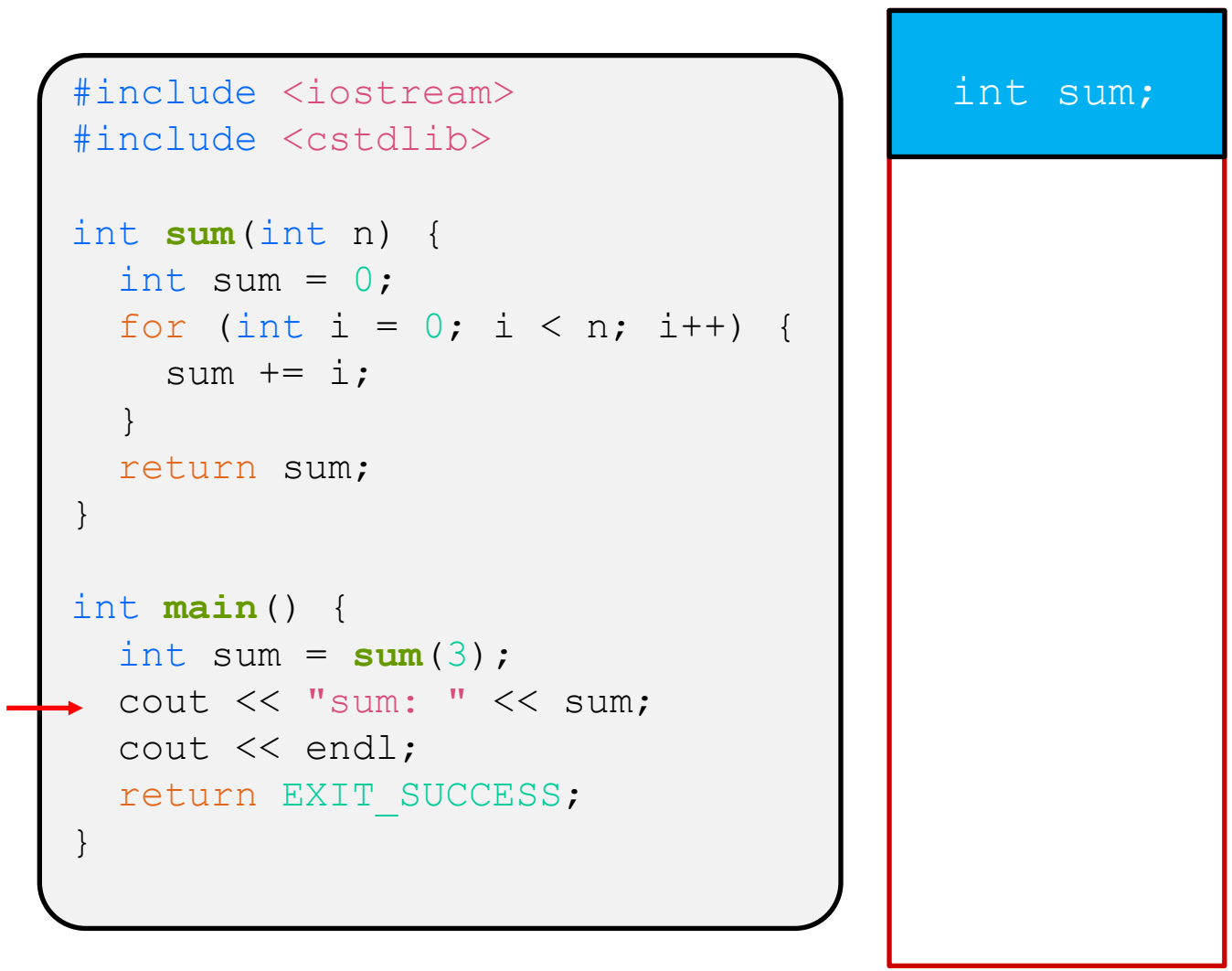


Stack frame for `main()`

Stack frame for `sum()`

Stack Example:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```



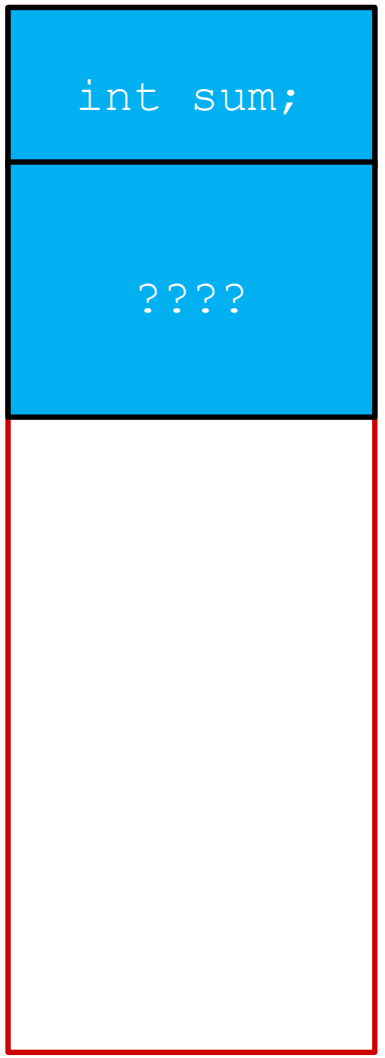
Stack frame for `main()`

`sum()`'s stack frame goes away after **`sum()`** returns.

`main()`'s stack frame is now top of the stack and we keep executing **`main()`**

Stack Example:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `cout << string`

Stack

- Grows, but has a static max size
  - Can find the default size limit with the command `ulimit -all`
    (May be a different command in different shells and/or linux versions. Works in bash on Ubuntu though)
  - Can also be found at runtime with `getrlimit(3)`

- Max Size of a stack can be changed
  - at run time with `setrlimit(3)`
  - At compilation time for some systems (not on Linux it seems)
  - (or at the creation of a thread)

Types of Memory

- There are three* main ways in which memory is allocated.

```
int* alloc_nums(size_t len) {
    int* arr = new int[len];
    for (size_t i = 0; i < len; i++) {
        arr[i] = 0;
    }
    return arr;
}

int main() {
    size_t len;
    cout << "please give a number: " << endl;
    cin >> len;  // reads an integer from stdin

    int* my_arr = alloc_nums(len);
    // ... do something with the array

    delete[] my_arr;
}
```

- `*arr`, `arr[i]`, etc are ***dynamically***-allocated.
  - Allocated "explicitly" by the program "at run time"
  - Deallocated when the program "explicitly" deallocates it.
- Compiler doesn't necessarily know how much memory will be allocated. It will be decided while running.
- Java, etc. uses dynamic allocation. It's just "hidden" and safer.
- Note: arr is a pointer that is on the stack with automatic allocation. The data that it is pointing at is on the heap.

How to mess it up

- Memory is easy to mess up, here are some ways how ☺
- In general:
  - Accessing memory that you are not supposed to access.
- Automatic Allocation:
  - Making a pointer to some thing on the stack and the pointer outlives the thing it is pointing at. Trying to dereference the pointer will get you Undefined Behaviour (UB)
- Dynamic Allocation:
  - Delete-ing a pointer that was not returned by new.
  - Using the wrong delete (delete vs delete[]).
  - Not deleting memory.
  - Deleting the same memory more than once.
  - Re-using memory after it has been deleted.
  - New doesn’t always initialize the memory allocated, you may need to set it yourself!

Ed Discussion

- What will happen when we try to compile and run?

A. Output “(2, 4, 0)"
B. Output “(2, 4, 3)"
C. Compiler error about arguments to foo (in main)
D. Compiler error about body of foo
E. We’re lost…

```
void foo(int* const x,
         int* y, int z) {
  *x += 1;
  *y *= 2;
   z -= 3;
}

int main(int argc, char** argv) {
  const int a = 1;
  int b = 2, c = 3;

  foo(&a, &b, c);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

Ed Discussion

- What will happen when we try to compile and run?

A. Output "(2, 4, 0)"

B. Output "(2, 4, 3)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

Can't modify the x, but can modify *x (dereference)

```
void foo(int* const x,
         int* y, int z) {
  *x += 1;
  *y *= 2;
   z -= 3;
}

int main(int argc, char** argv) {
  const int a = 1;
  int b = 2, c = 3;

  foo(&a, &b, c);
  std::cout << "(" << a << ", " << b
   << ", " << c << ")" << std::endl;

  return EXIT_SUCCESS;
}
```

Int ptr → int*

Allowed

Copy of int value

Allowed, but change doesn't persist out

Const mismatch

Ed Discussion

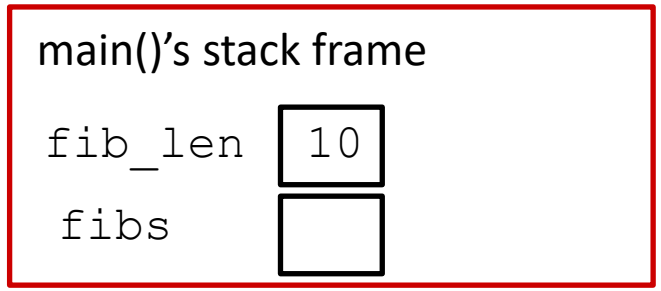
- Debug this function, what is wrong with it? how do we fix it?

```
int* find_fibs(size_t upper) {
  int res[upper];
  if (upper > 0) {
    res[0] = 0;
    res[1] = 1;
  }
  for(size_t i = 1; i < upper; ++i) {
    res[i] = res[i - 1] + res[i - 2];
  }
  return res;
}
```

```
int* find_fibs(size_t upper);

int main() {
  size_t fib_len = 10;
  int* fibs = find_fibs(fib_len);

  for (size_t i = 0; i <= fib_len; ++i) {
    cout << "fibs[i]: " << fibs[i] << endl;
  }

  return EXIT_SUCCESS;
}
```

Ed Discussion

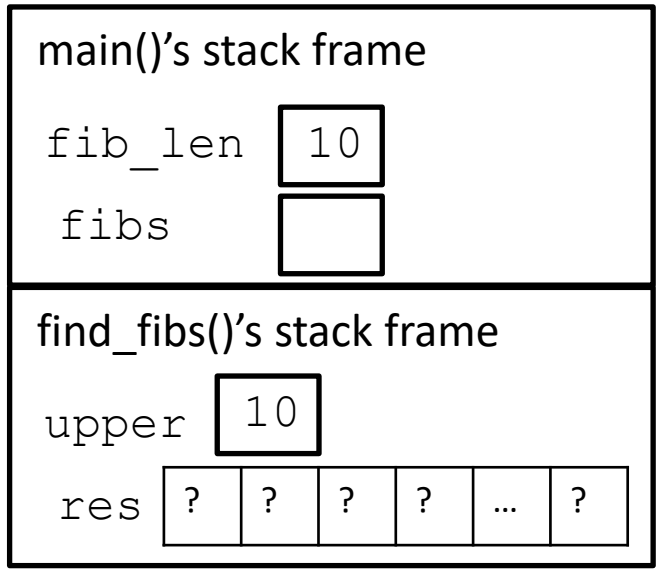
- Does this function work as intended?




```
int main() {
  size_t fib_len = 10;
  int* fibs = find_fibs(fib_len);

  for (size_t i = 0; i <= fib_len; ++i) {
    cout << "fibs[i]: " << fibs[i] << endl;
  }

  return EXIT_SUCCESS;
}
```

Ed Discussion

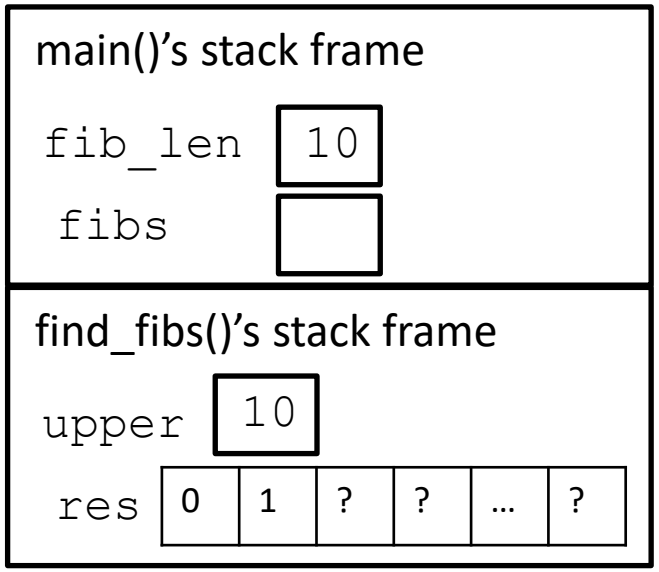
- Does this function work as intended?




```
int* find_fibs(size_t upper) {
  int res[upper];
  if (upper > 0) {
    res[0] = 0;
    res[1] = 1;
  }
  for(size_t i = 1; i < upper; ++i) {
    res[i] = res[i - 1] + res[i - 2];
  }
  return res;
}
```

Ed Discussion

- Does this function work as intended?

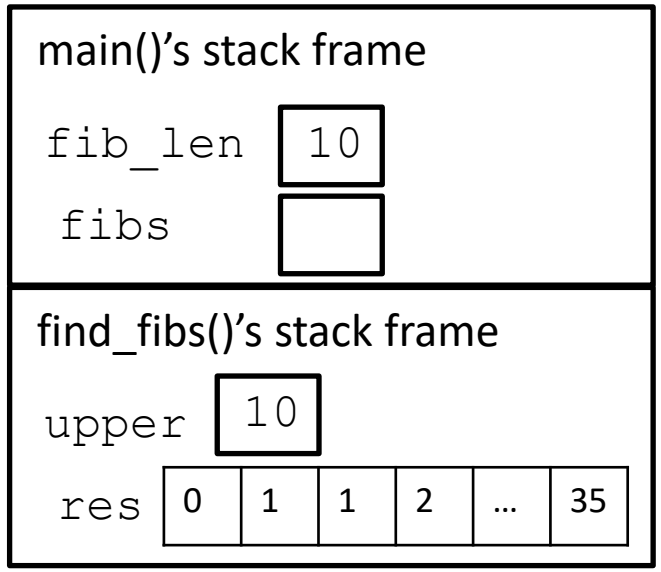



```
int* find_fibs(size_t upper) {
  int res[upper];
  if (upper > 0) {
    res[0] = 0;
    res[1] = 1;
  }
  for(size_t i = 1; i < upper; ++i) {
    res[i] = res[i - 1] + res[i - 2];
  }
  return res;
}
```

Off by one error: if upper == 1
then we go out of bounds of the array

Ed Discussion

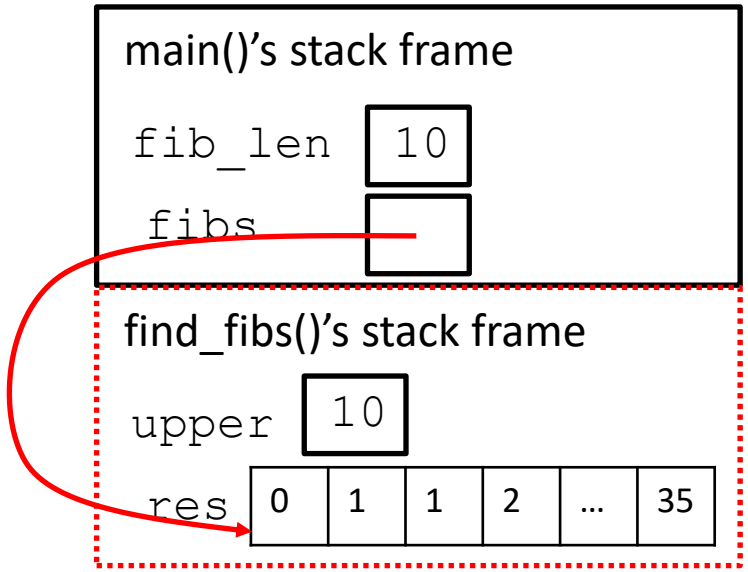
- Does this function work as intended?




```
int* find_fibs(size_t upper) {
  int res[upper];
  if (upper > 0) {
    res[0] = 0;
    res[1] = 1;
  }
  for(size_t i = 1; i < upper; ++i) {
    res[i] = res[i - 1] + res[i - 2];
  }
  return res;
}
```

Ed Discussion

- Does this function work as intended?

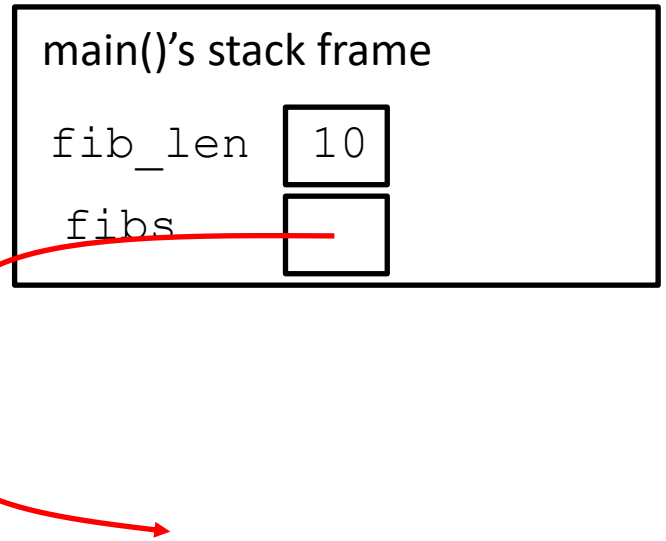


Stack local variable goes away

```
int* find_fibs(size_t upper) {
  int res[upper];
  if (upper > 0) {
    res[0] = 0;
    res[1] = 1;
  }
  for(size_t i = 1; i < upper; ++i) {
    res[i] = res[i - 1] + res[i - 2];
  }
  return res;
}
```

Ed Discussion

- Does this function work as intended?




```
int main() {
  size_t fib_len = 10;
  int* fibs = find_fibs(fib_len);

  for (size_t i = 0; i <= fib_len; ++i) {
    cout << "fibs[i]: " << fibs[i] << endl;
  }

  return EXIT_SUCCESS;
}
```

Stack local variable goes away

We try to print memory we don't have access to!

(and we go out of bounds in the for loop too)

Ed Discussion

- Consider our `SimpleString` "object" from HW00, could we implement it without a pointer to the heap?

```
struct SimpleString {
  char* data;
  size_t len;
};
```

Ed Discussion

- Consider our `SimpleString` "object" from HW00, could we implement it without a pointer to the heap?

```
struct SimpleString {
  char* data;
  size_t len;
};
```

- Does this work?

```
struct SimpleString {
  char data[10];
  size_t len;
};
```

Ed Discussion

- Consider our `SimpleString` "object" from HW00, could we implement it without a pointer to the heap?

```
struct SimpleString {
  char* data;
  size_t len;
};
```

- We have to use a pointer, we don't know how big the string is!

```
struct SimpleString {
  char data[10];
  size_t len;
};
```

Ed Discussion

- Consider our `SimpleString` "object" from HW00, could we implement it without a pointer to the heap?

```
struct SimpleString {
  char* data;
  size_t len;
};
```

- Does this work?

```
SimpleString SimpleString_From(const char* cstring) {
  SimpleString ret;
  char arr[strlen(cstring) + 1];
  for (size_t i = 0; i <= strlen(cstring); i++) {
    arr[i] = cstring[i]
  }
  ret.data = arr;
  ret.len = strlen(cstring);

  return ret;
}
```

Ed Discussion

- Consider our `SimpleString` "object" from HW00, could we implement it without a pointer to the heap?

```
struct SimpleString {
  char* data;
  size_t len;
};
```

- We have to use a pointer we don't know how big the string is!
- If we use a pointer, where is it pointing? Pointing to the stack won't work…

```
SimpleString SimpleString_From(const char* cstring) {
  SimpleString ret;
  char arr[strlen(cstring) + 1];
  for (size_t i = 0; i <= strlen(cstring); i++) {
    arr[i] = cstring[i]
  }
  ret.data = arr;
  ret.len = strlen(cstring);

  return ret;
}
```

Ed Discussion

```
int main() {
  // start as length 2 array {3, 0}
  int* arr = new int[2]{3, 0};
  size_t len = 2;

  PushNums(arr, len, 3);
  len += 1;
  PushNums(arr, len, 4);
  len += 1;

  for (size_t i = 0; i < len; ++i) {
    cout << arr[i] << endl;
    delete arr + i;
  }

  delete arr;
}
```

- What is wrong with this code? (Multiple bugs)
  You can assume this compiles.
- How do we fix this?

```
void PushNums(int* arr, size_t len,
              int new_num) {
  int* new_arr = new int[len + 1];
  for (size_t i = 0; i < len; ++i) {
    new_arr[i] = arr[i];
  }
  new_arr[len + 1] = new_num;
  arr = new_arr;
}
```

Ed Discussion

- Finish writing this function and program:

```
struct String {
  char* data;
  size_t len;
};

void StringAppend(String* str, char* new_data) {
    // finish this function
}

int main(int argc, char* argv[]) {
  // write a main that takes all the args, adds them to one string,
  // then prints them. Make sure you have no memory leaks or other errors.
  // you can assume you have access to String String_From(char* cstr)
  // which creates a String deep copying the passed in c-string
}
```

Will I Actually Use new?

- In “real” or “modern” C++ code, you would not explicitly use new or delete yourself.

- In most cases, a string, vector or other data structure can be used, and you never have to allocate memory yourself. (BUT it is still important to be conscious of all the memory allocations going on!)

- Whenever you are using objects from the C++ standard library (more later), those objects will do memory allocation.

- For now: we will handle memory ourselves.
(not for very long, after HW2 we will be more modern)

Raise Your Hands

- QUICK, which of these two is faster?

```
int fib(int n) {
  int arr[2] = {0, 1};
  int index = 0;
  int i = 0;

  while (i < n) {
    arr[index] = arr[0] + arr[1];
    i += 1;
    index = 1 - index;
  }

  return arr[1 - index];
}
```

```
int fib(int n) {
  int* arr = new int[2]{0, 1};

  for (int i = 0; i < n; ++i) {
    arr[i % 2] = i[0] + i[1];
  }

  delete[] arr;

  return arr[(i - 1) % 2];
}
```

Heap is SLOW

- QUICK, which of these two is faster?

```
int fib(int n) {
  int arr[2] = {0, 1};
  int index = 0;
  int i = 0;

  while (i < n) {
    arr[index] = arr[0] + arr[1];
    i += 1;
    index = 1 - index;
  }

  return arr[1 - index];
}
```

```
int fib(int n) {
  int* arr = new int[2]{0, 1};

  for (int i = 0; i < n; ++i) {
    arr[i % 2] = i[0] + i[1];
  }

  bool res = arr[(i - 1) % 2]
  delete[] arr;

  return res;
}
```

- There is overhead to the program maintaining the heap. Using the heap is almost always slower.
- If you can avoid using the heap, then you should ☺
- (so even if we don't call new explicitly in modern C++, we need to be conscious of when it is called for us)

Lecture Outline

- Pointers & Const
- The Heap
- **Value Semantics**
- References

Raise Your Hands

- What does this code print?

3
5

```
int modify_int(int x) {
  x = 5;
  return x;
}

int main() {
  int num = 3;
  int n = modify_int(num);
  cout << num << endl;
  cout << n << endl;
  return EXIT_SUCCESS;
}
```

Raise Your Hands

- What does this code print?

  1100
  3800

- How could we fix it?
  E.g. make modify_point actually modify the input?

```
struct Point {
  int x;
  int y;
};

Point modify_point(Point p) {
  p.x = 3800;
  p.y = 4710;
  return p;
}

int main() {
  Point pt0 = {1100, 2400};
  Point pt1 = modify_point(pt0);
  cout << pt0.x << endl;
  cout << pt1.x << endl;
  return EXIT_SUCCESS;
}
```

Value Semantics

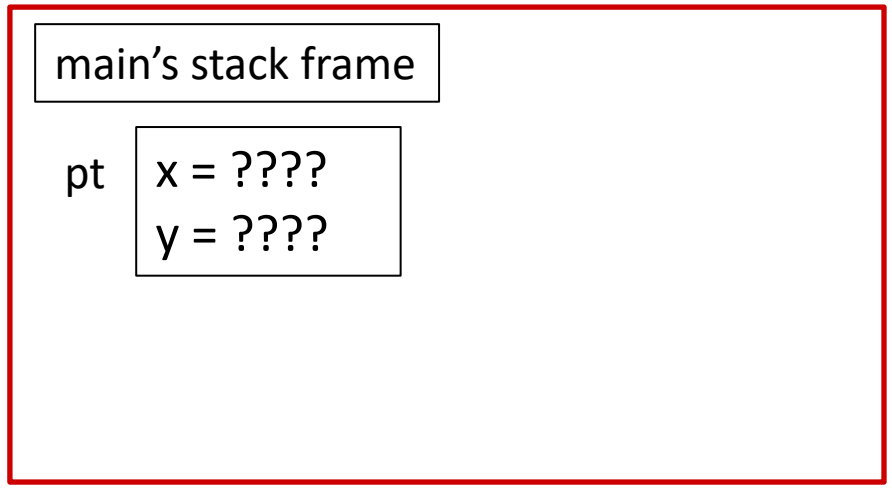
- C++ (and most "systems" programming languages) use **Value Semantics** *by default*.

- When we pass something to a function, we pass a ***copy*** of that thing.
- When we return a thing, we return a ***copy*** of that thing

- HOWEVER, we can pass a copy of a pointer (e.g. a reference to something) to mimic pass-by-reference.
  - (Or use something mentioned later in lecture ☺)

Structs: Copied not Referenced

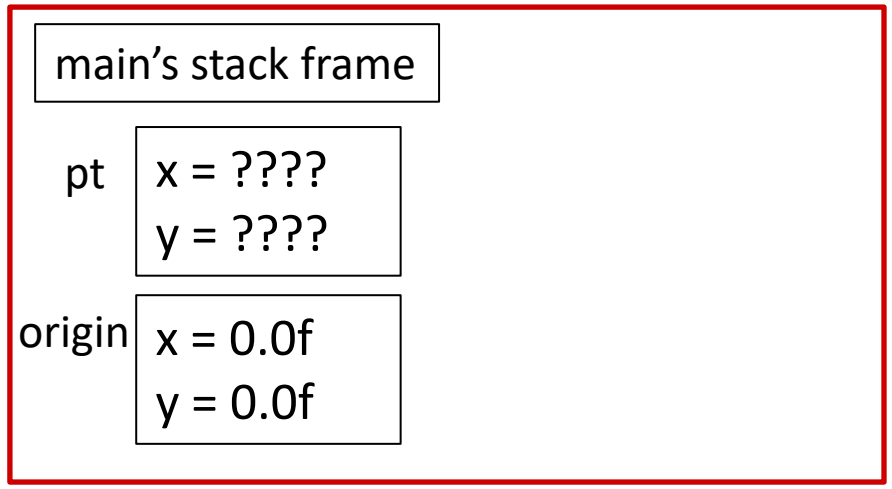
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```
struct Point {
  float x;
  float y;
};

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

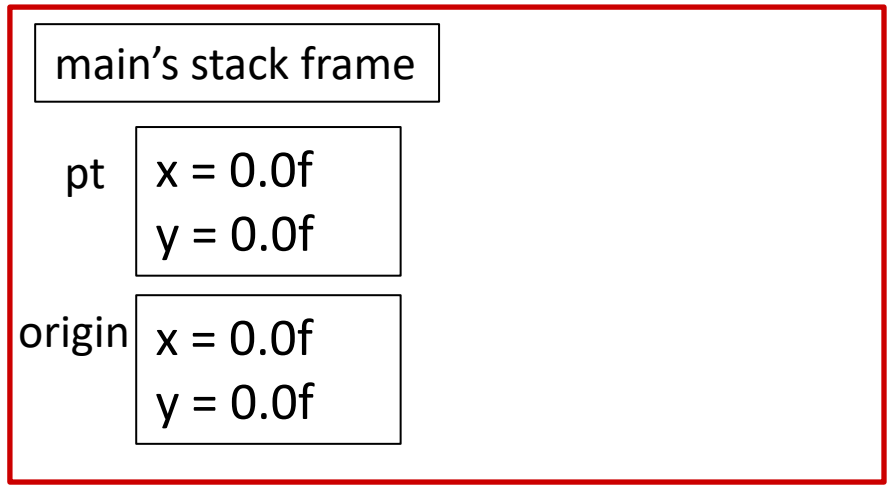
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```
struct Point {
  float x;
  float y;
};

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

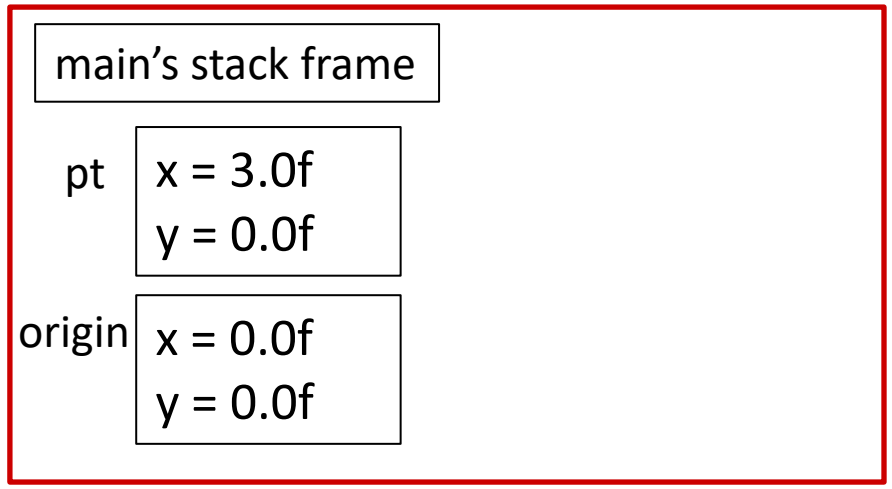
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```
struct Point {
  float x;
  float y;
};

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

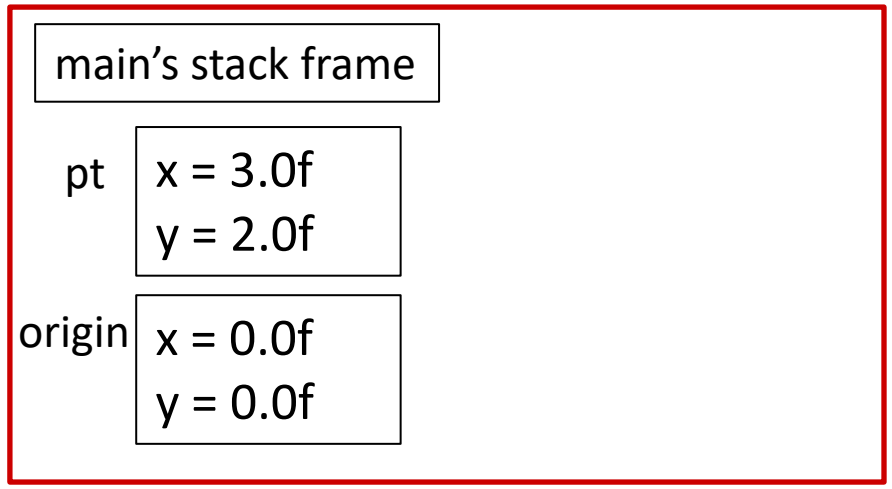
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```
struct Point {
  float x;
  float y;
};

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```
struct Point {
  float x;
  float y;
};

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Visualization: fixed pass by reference (Output Parameters)

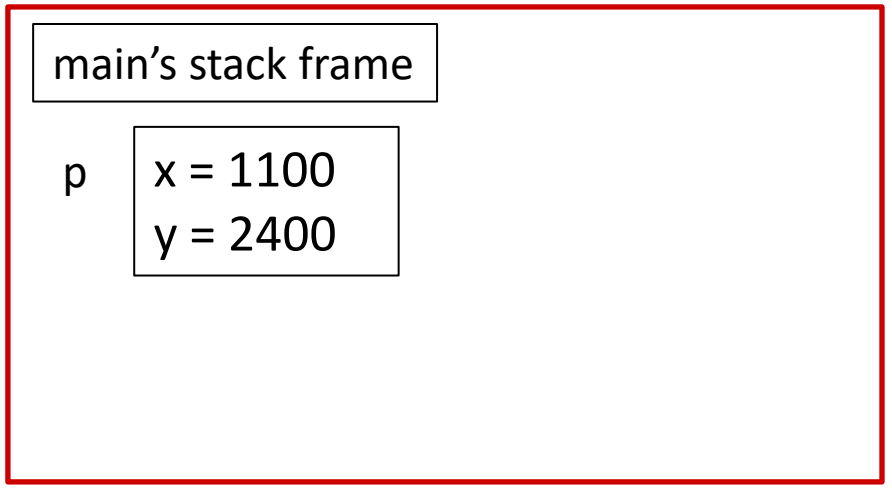
```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)

Buggy version would say:
`ptr = &new_point`

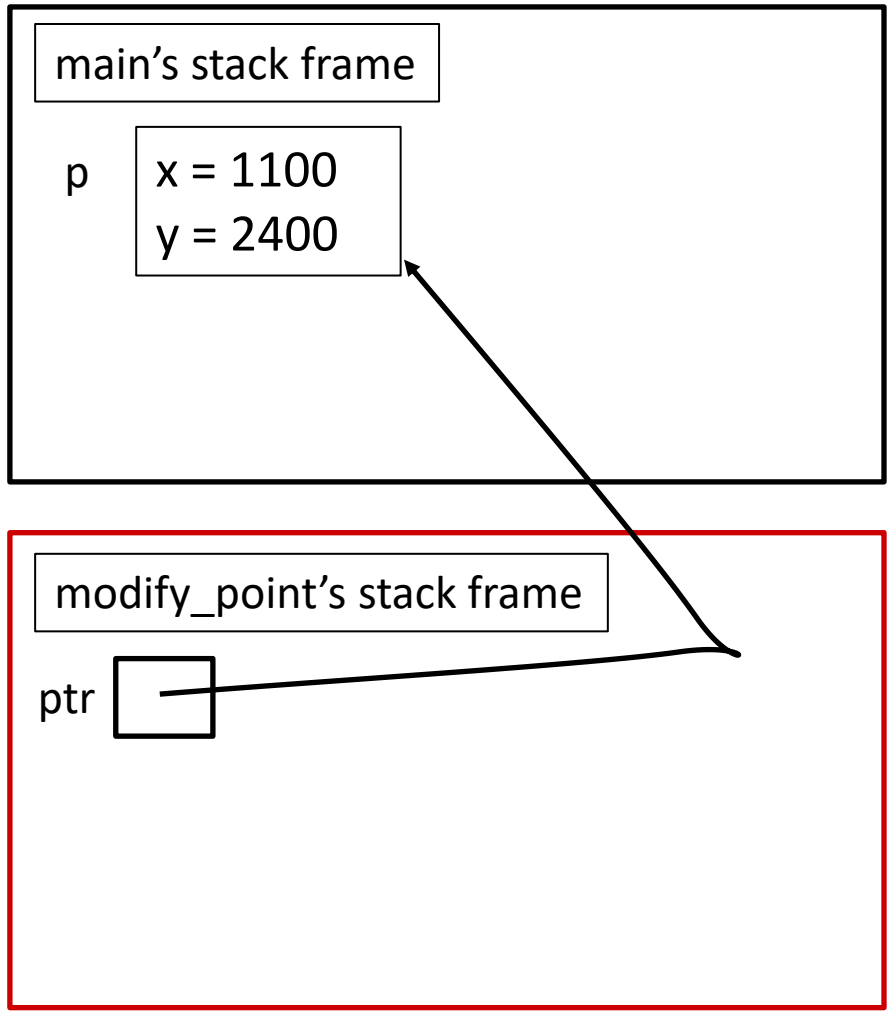
```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



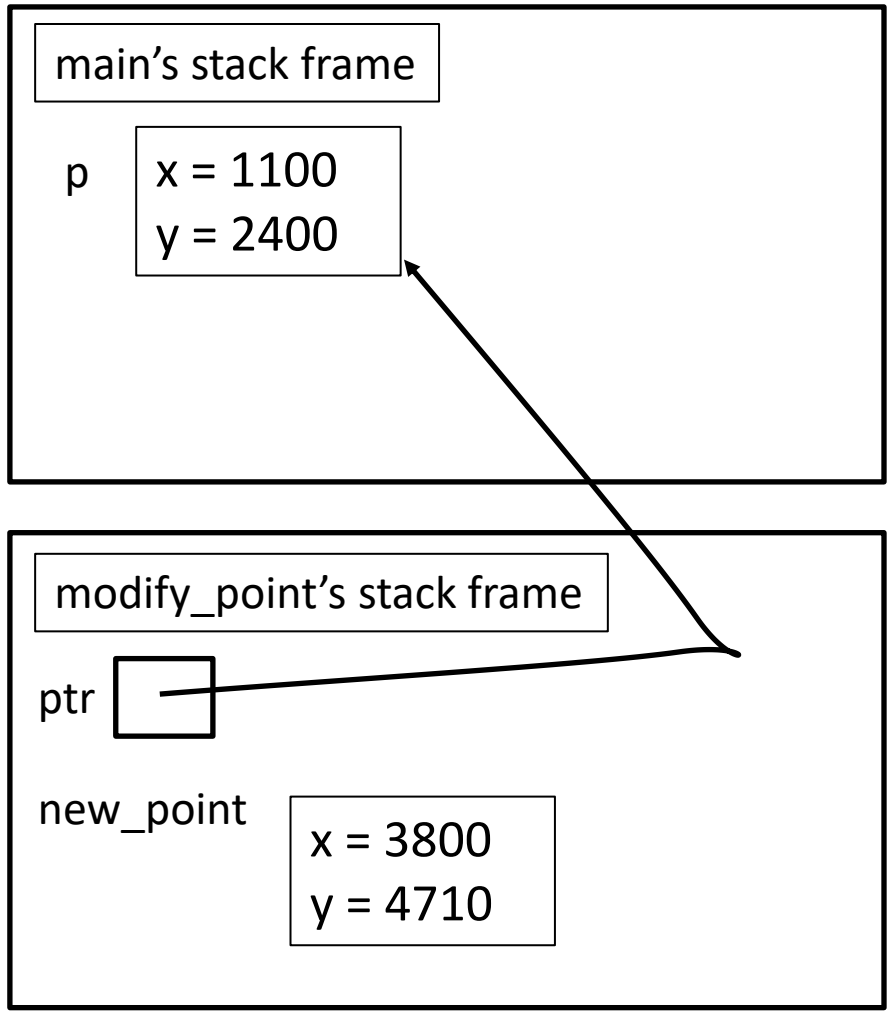

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



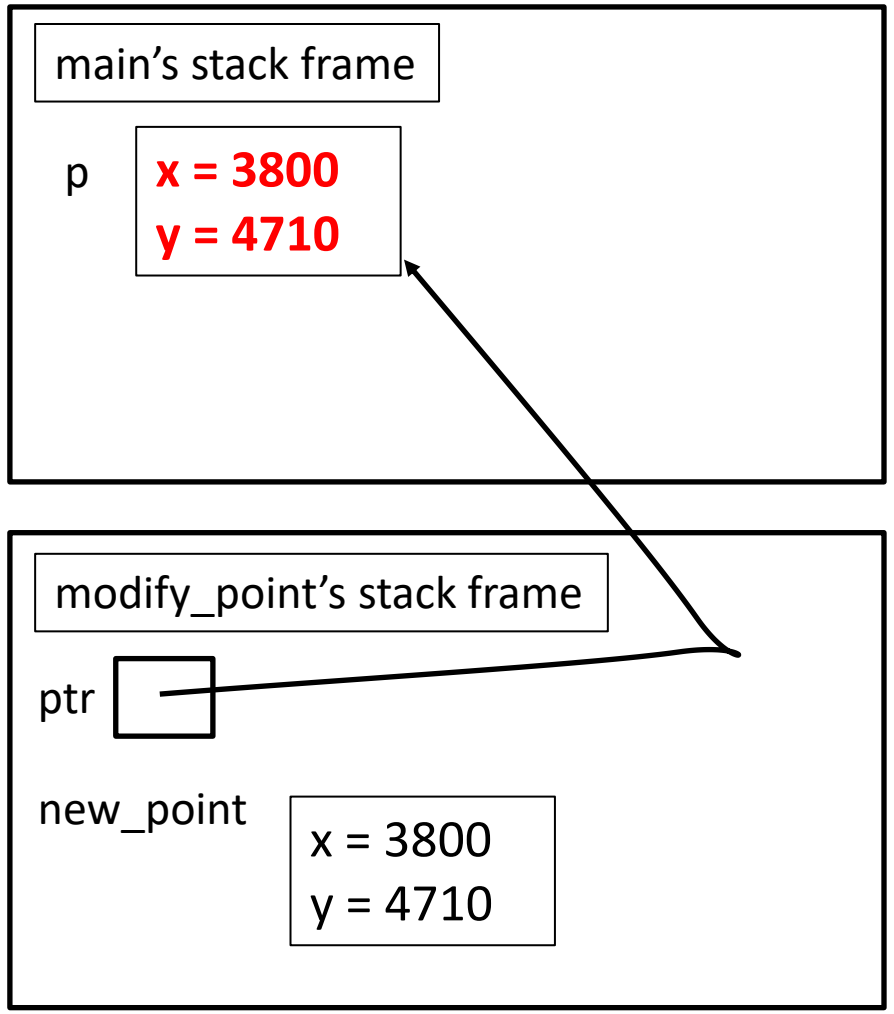

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



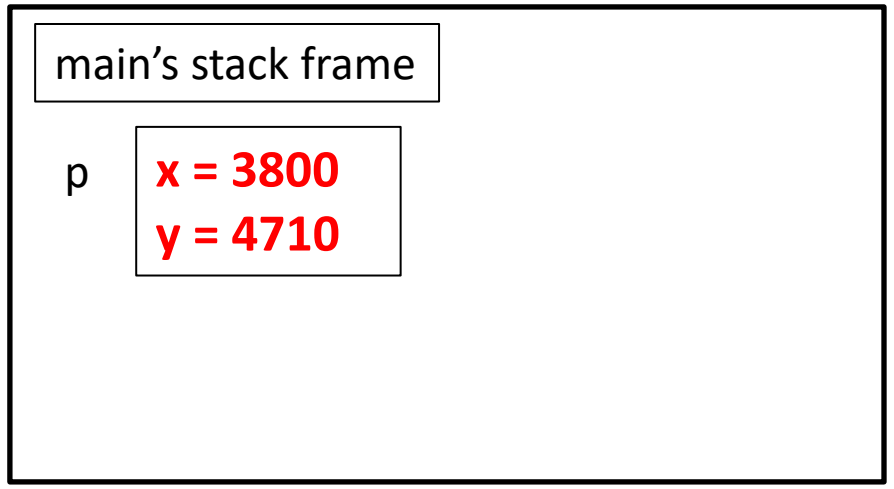

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Pointers & Const
- The Heap
- Value Semantics
- **References**

References

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in **C++** as part of the language

- References are mostly an alternative to pointers
  - They are implemented internally with a pointer
  - But references are a lot easier to use
  - Can't do everything with references, sometimes a pointer is needed.
  - References are used a lot more often than pointers are

References

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"

References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"

`main()`'s stack frame

x	5
y	10

References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"

`main()`'s stack frame

x, z	5
y	10

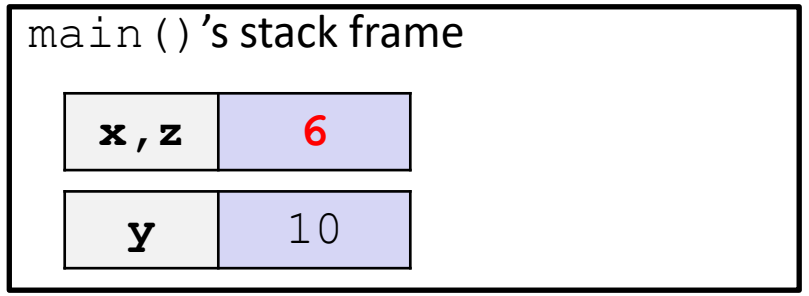
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



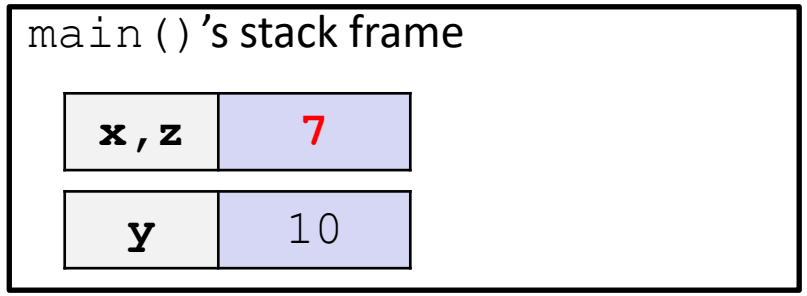
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



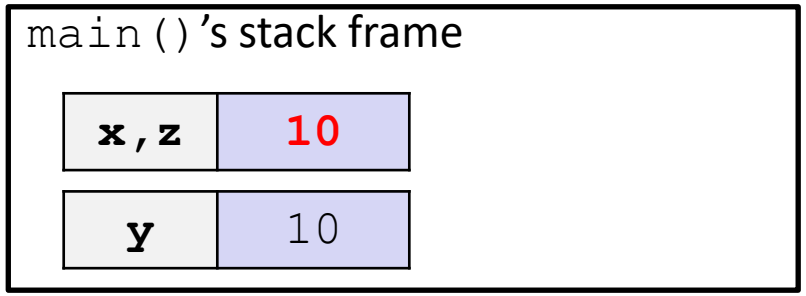
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



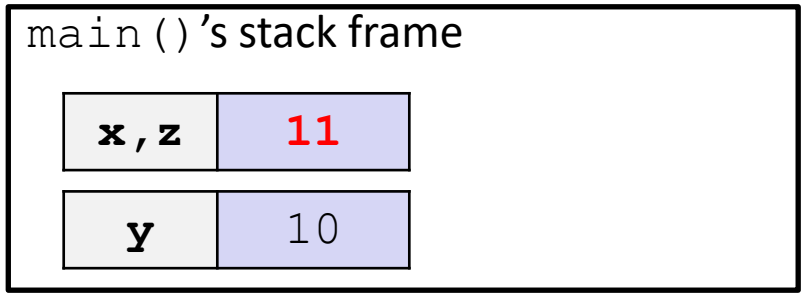
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



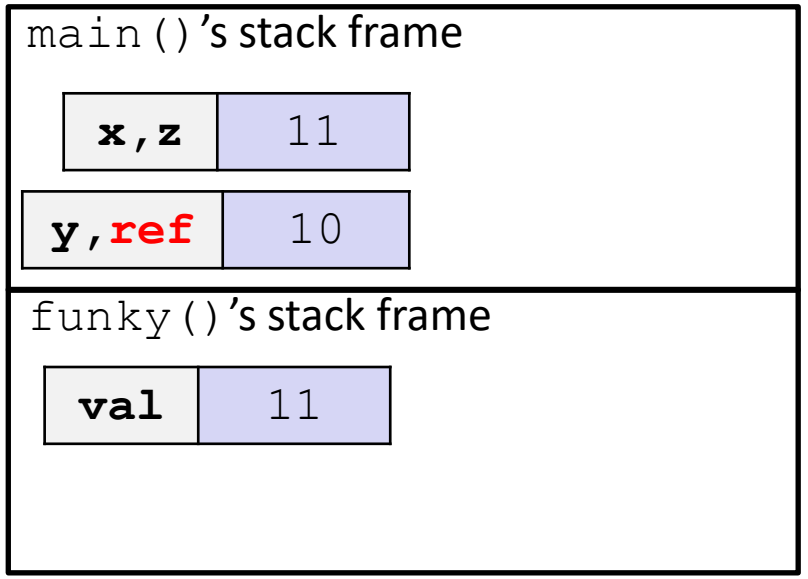
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



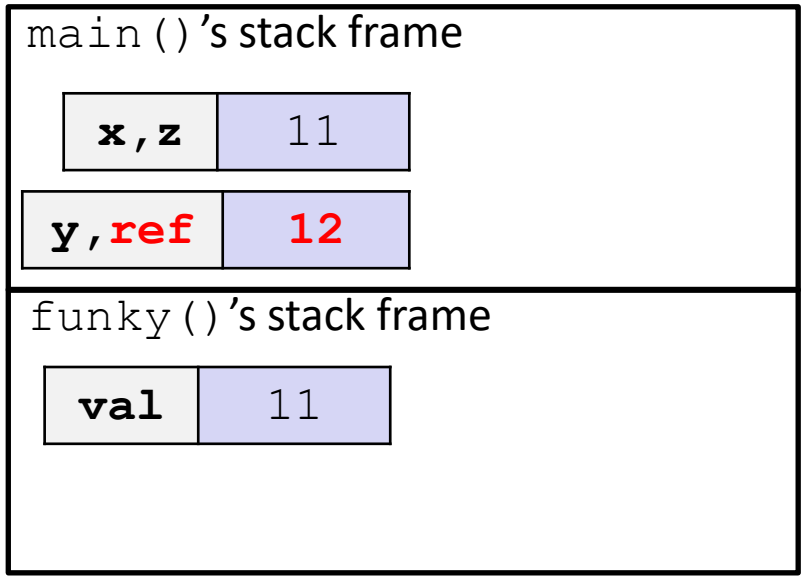
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



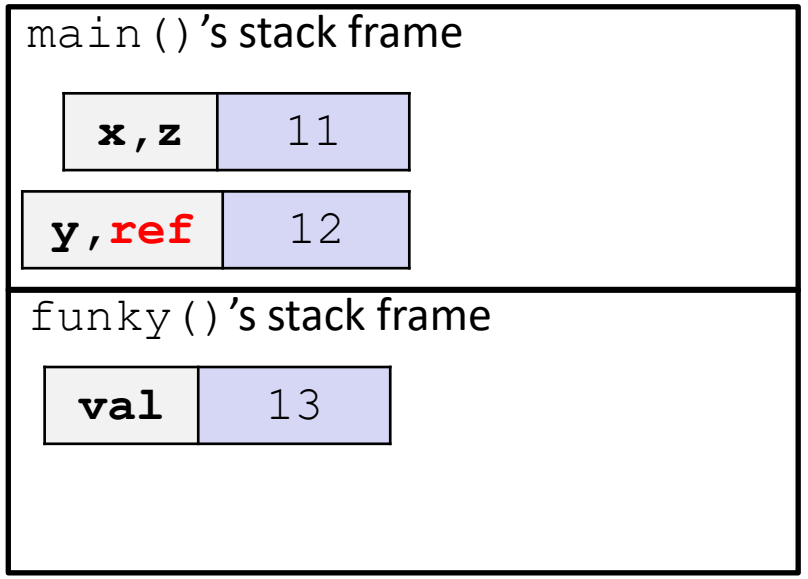
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



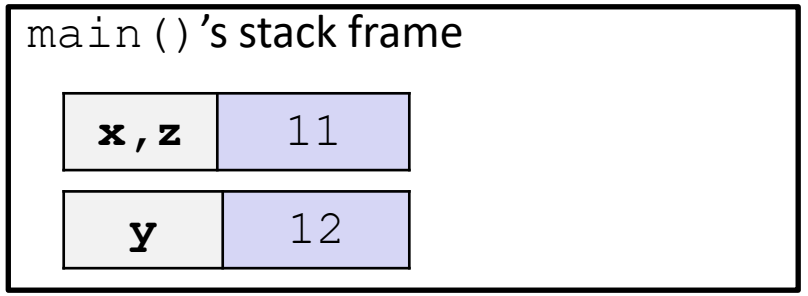
References

Note: Arrow points to *next* instruction.

```
void funky(int val, int& ref) {
  ref += 2;
  val += 2;
}

int main() {
  int x = 5;
  int y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z = y;
  z += 1;

  funky(x, y);
}
```

When we use '&' in a type declaration, it is a reference.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language
  - Lets us "pass things by reference"



That's all for now!

- Releasing tonight or tomorrow:
  - HW01
- Still out:
  - HW00
  - Pre-semester Survey
- Hopefully you are doing well ☺