[368] Building, Structs, Pointers

Tyler Caraza-Harter

What will you learn today?

Learning objectives

- organize code into multiple implementation (.cpp) and header (.h) files
- write Makefiles to incrementally build code
- create a shared library
- organize related data into structs (structures)
- manipulate data indirectly via pointers
- differentiate between lvalues and rvalues

What will you learn today?

Learning objectives

- organize code into multiple implementation (.cpp) and header (.h) files
- write Makefiles to incrementally build code
- create a shared library
- organize related data into structs (structures)
- manipulate data indirectly via pointers

demos...

Outline

Project Organization Demos

Structs

Pointers

lvalues vs. rvalues

Stucts with Pointers

Structures: Motivation

```
int main() {
  // x,y coords for point a
  int ax{0};
  int ay{0};

  // x,y coords for point b
  int bx{0};
  int by{0};
}
```

Common scenario: logical entitities (for example, coordinates) have multiple pieces of associated data (for example, longitude, latitude, altitude).

Problem: this can lead to messy code and a proliferation of variables.

Stucture Syntax

```
struct Loc {
  int x = 0;
  int y = 0;
};

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};
}
```

Solution: create new types using structs. Each variable (a and b) has its own members (x and y).

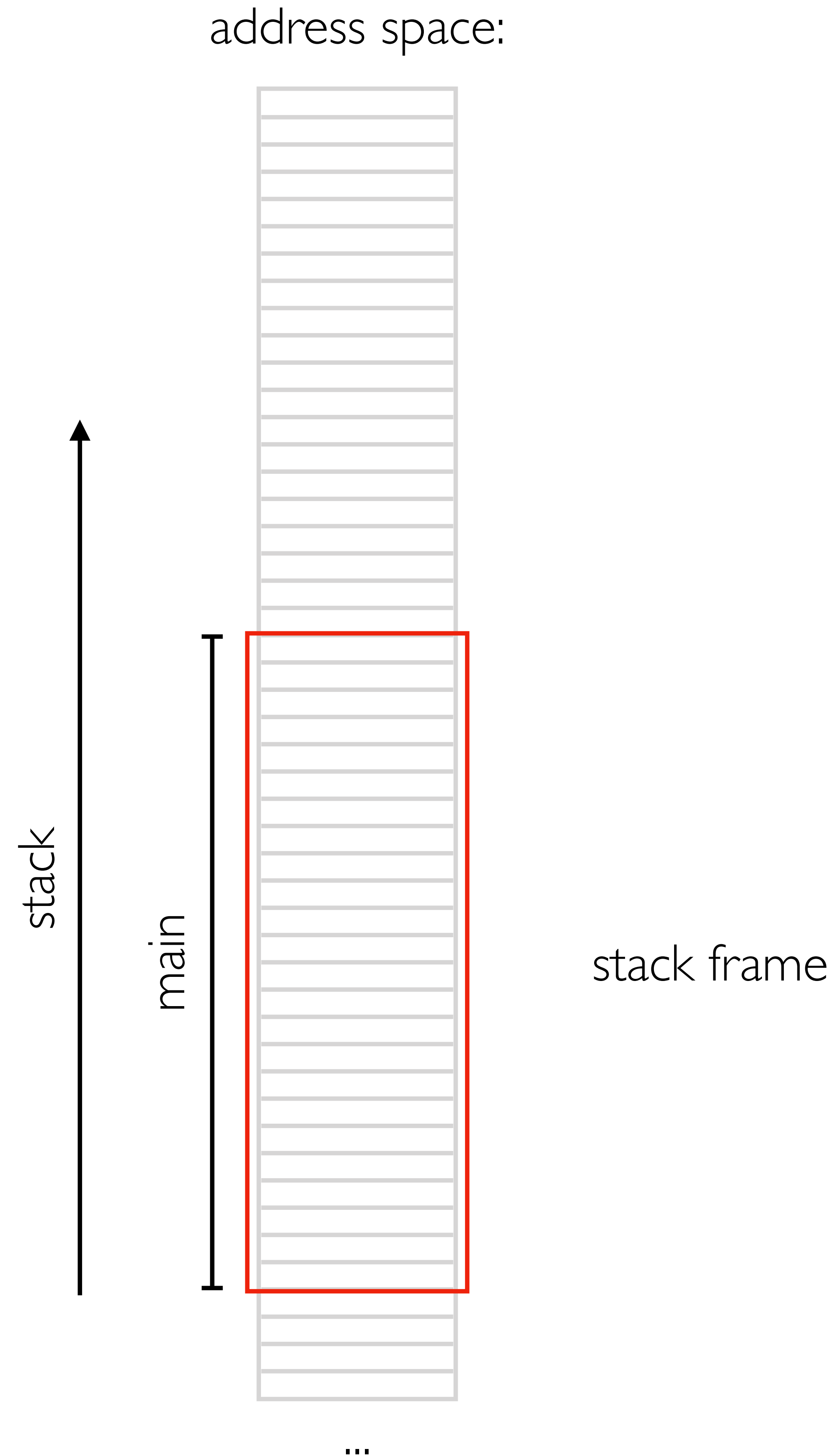
Structure Reference Operator (.)

```
struct Loc {
  int x = 0;
  int y = 0;
};

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};

  std::cout << a.x << "\n";
  std::cout << b.x << "\n";
}
```

get value from struct

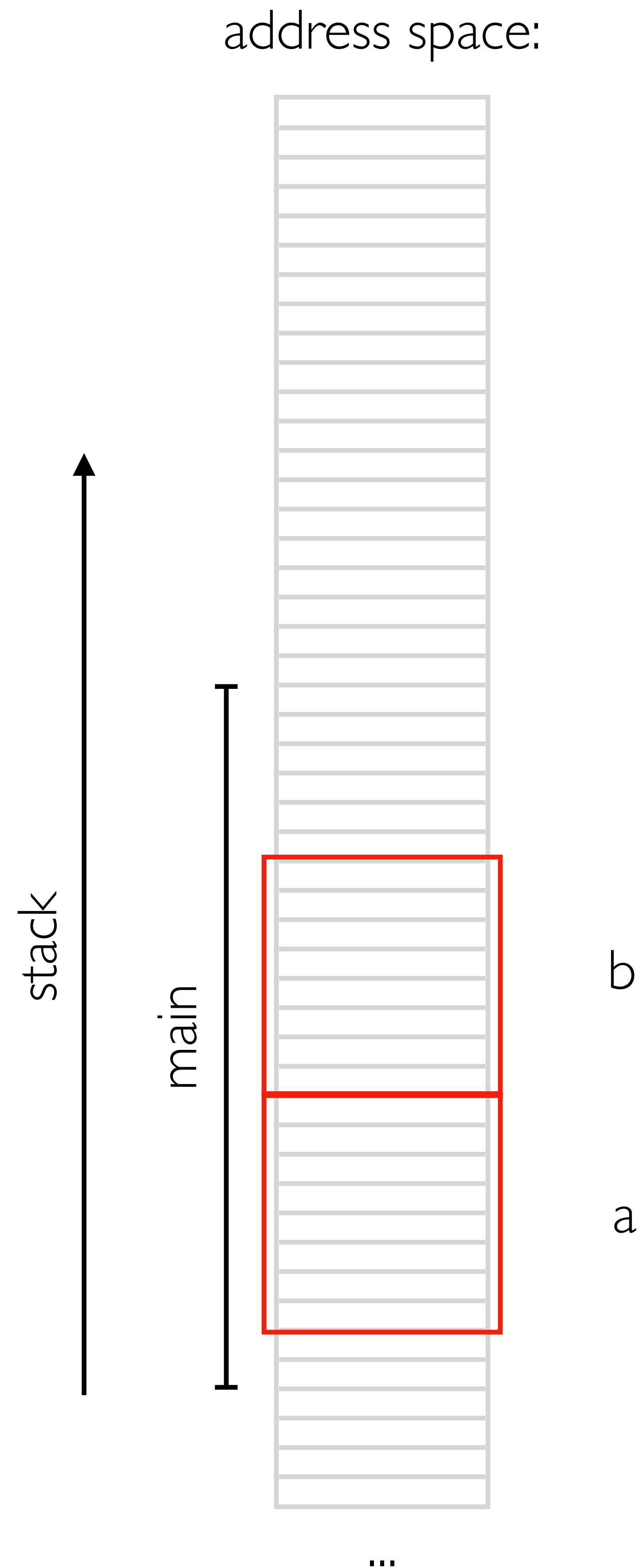
Pass by Value

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc c) {
 ...
}

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};

}
```



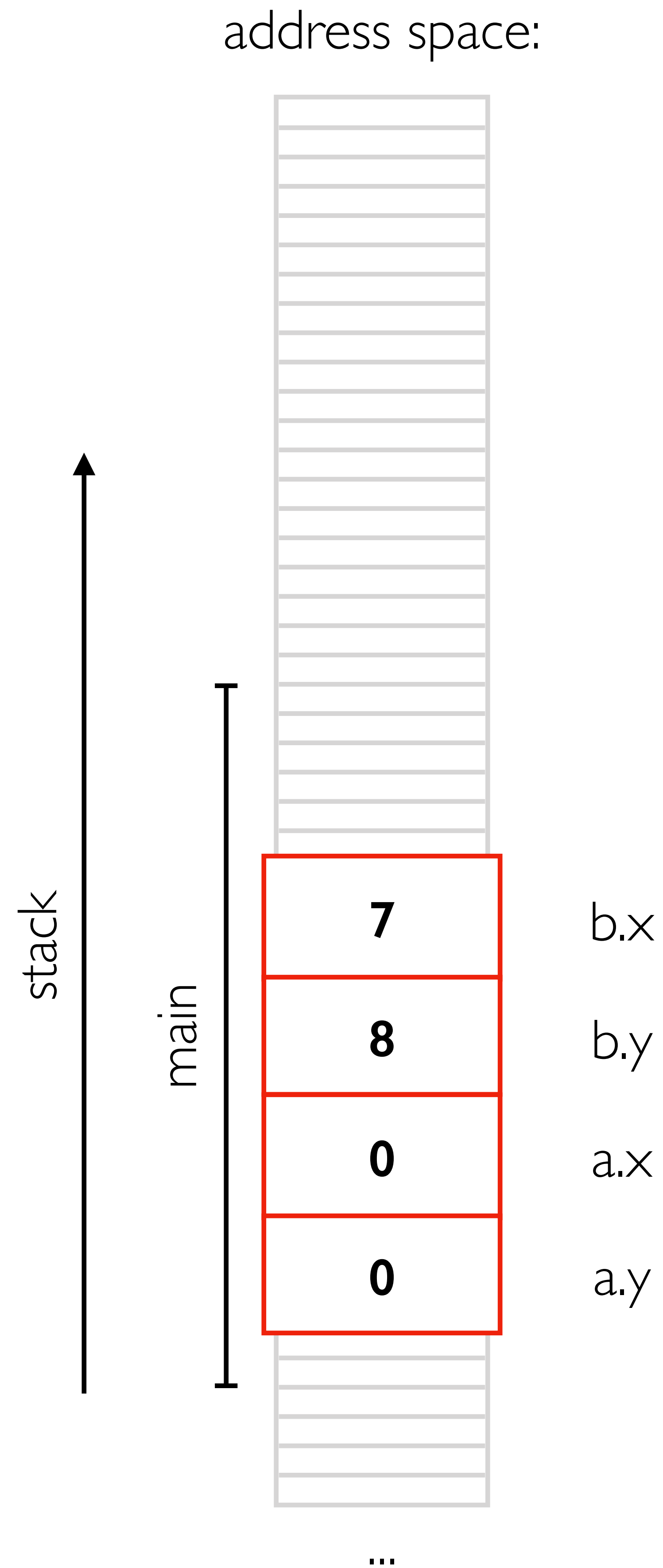
Pass by Value

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc c) {
 ...
}

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};

}
```



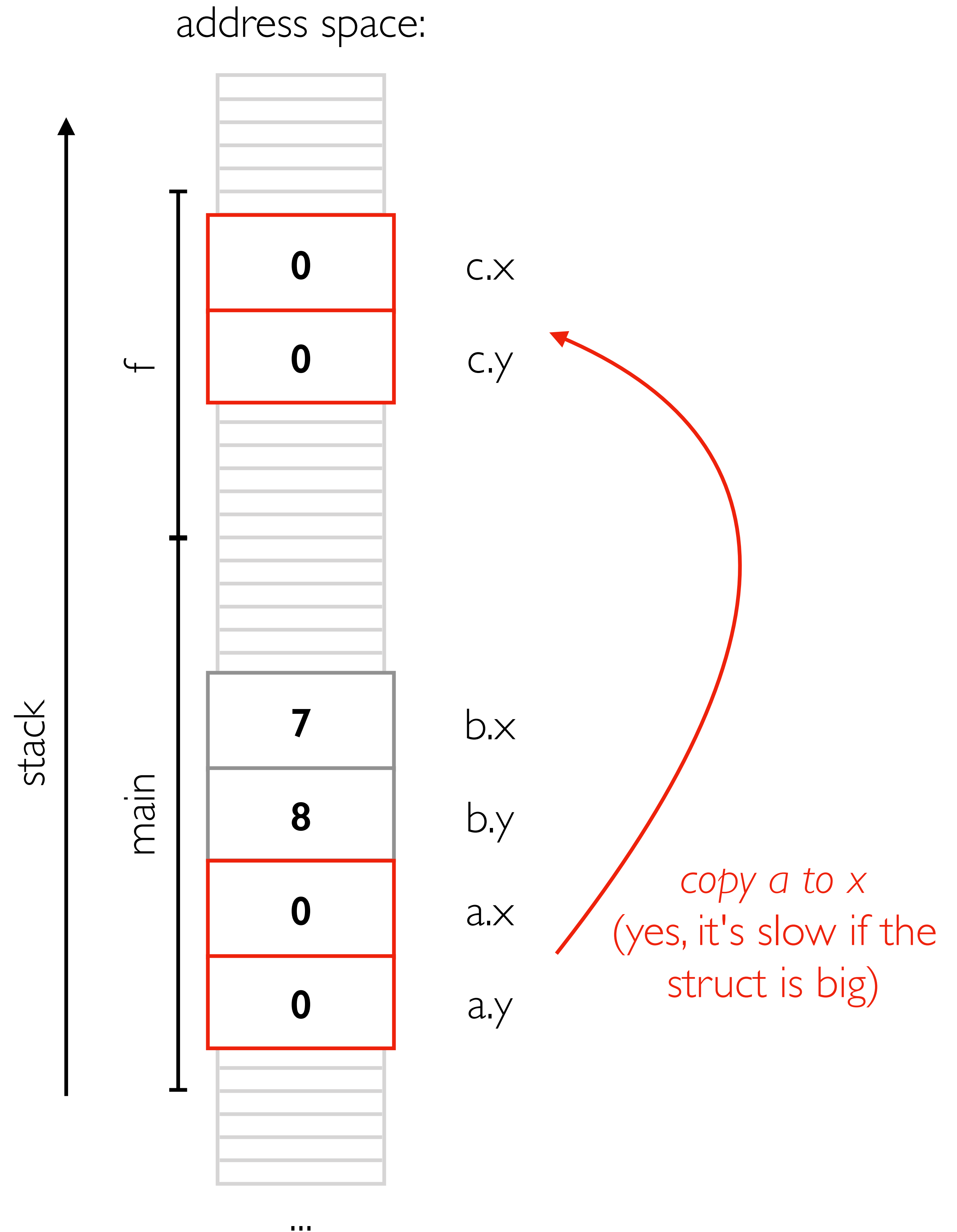
Pass by Value

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc c) {
 ...
}

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};

}
```



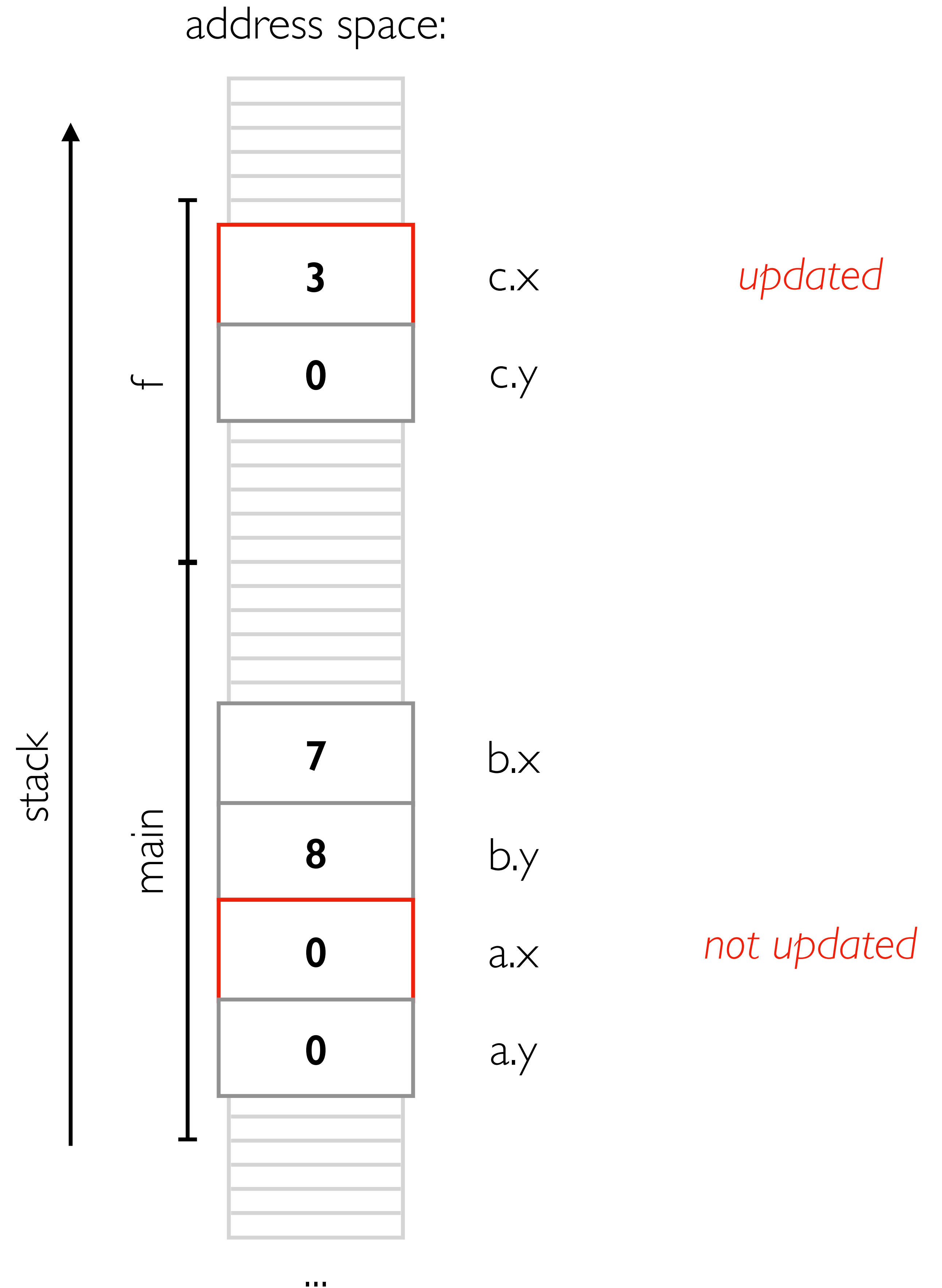
Pass by Value

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc c) {
 ...
}

int main() {
  Loc a{};
  Loc b{.x=7, .y=8};
  f(a);   call!
}
```



Pass by Value

```
struct Loc {
   int x = 0;
   int y = 0;
};

void f(Loc c) {
   c.x = 3    modify
}

int main() {
   Loc a{};
   Loc b{.x=7, .y=8};
   f(a);
}
```



not always our desired behavior.
sometimes we don't want a copy.
sometimes we want to modify the argument.

Outline

Project Organization Demos

Structs

Pointers

lvalues vs. rvalues

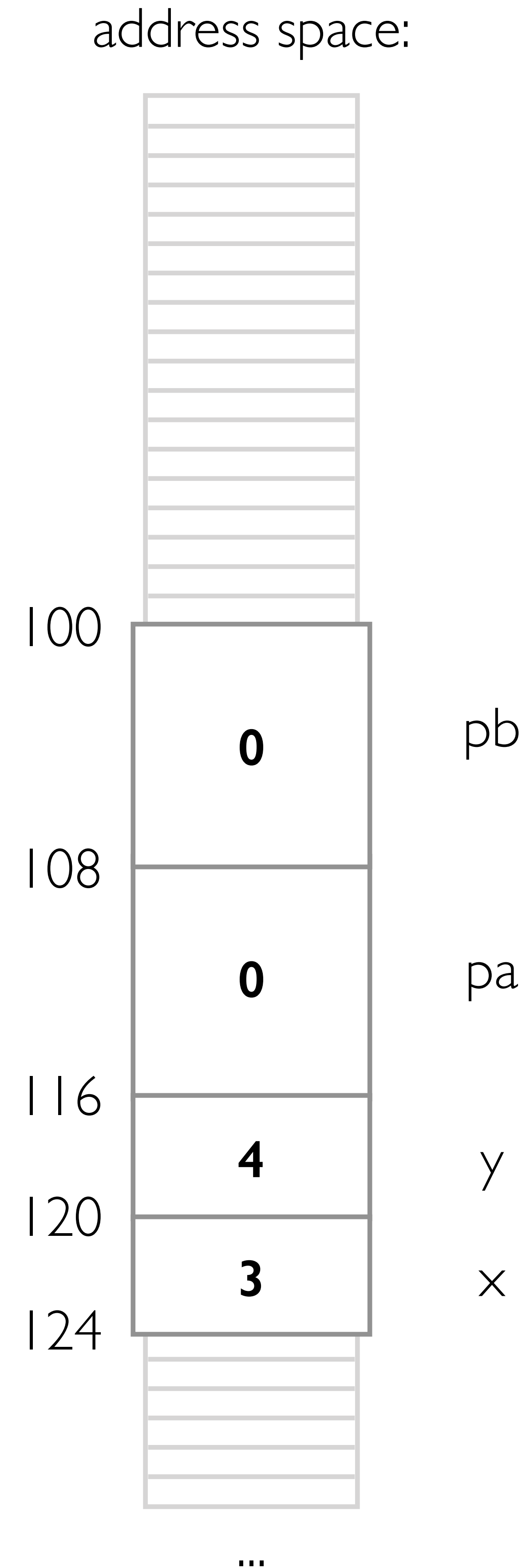
Stucts with Pointers

Address and Pointers

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
}
```

"int*" and "int *" are the same
(former leads to better intuition)

addresses



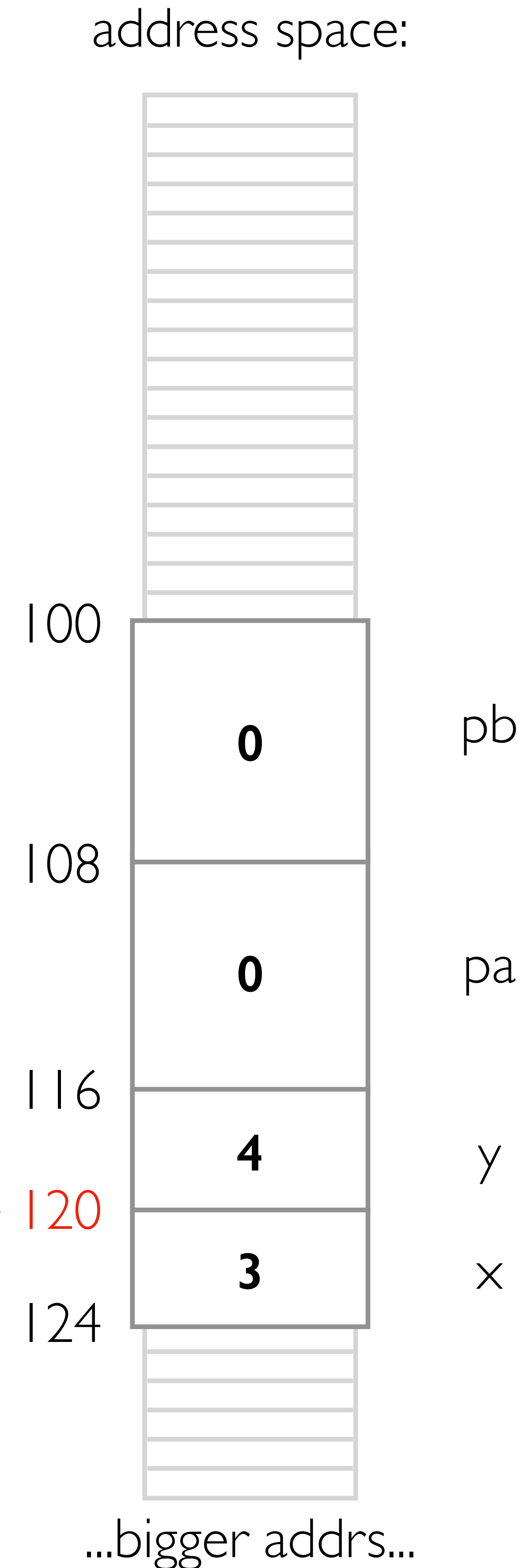
TYPE ????; variable containing value of TYPE
TYPE* ????; pointer variable containing addressof value of TYPE

Address-Of Operator

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0

  cout << &x << "\n";
}
```

& operator gives as an address.
will print 120 (in hexidecimal)

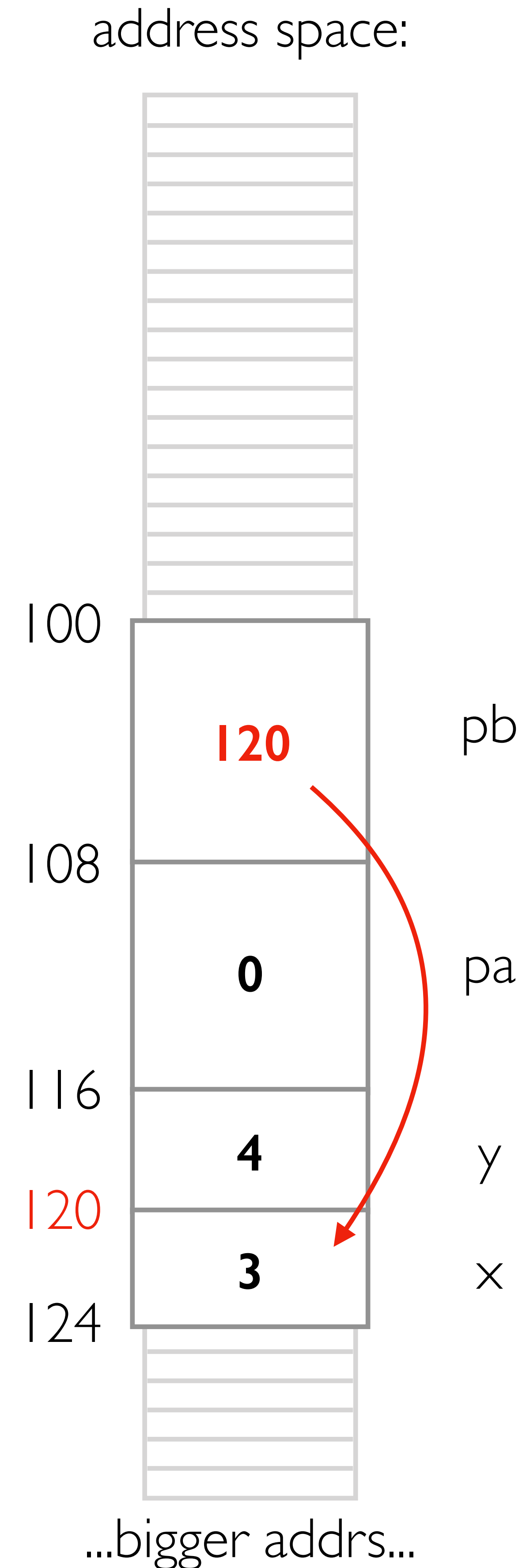
"address of" operator



Address-Of Operator

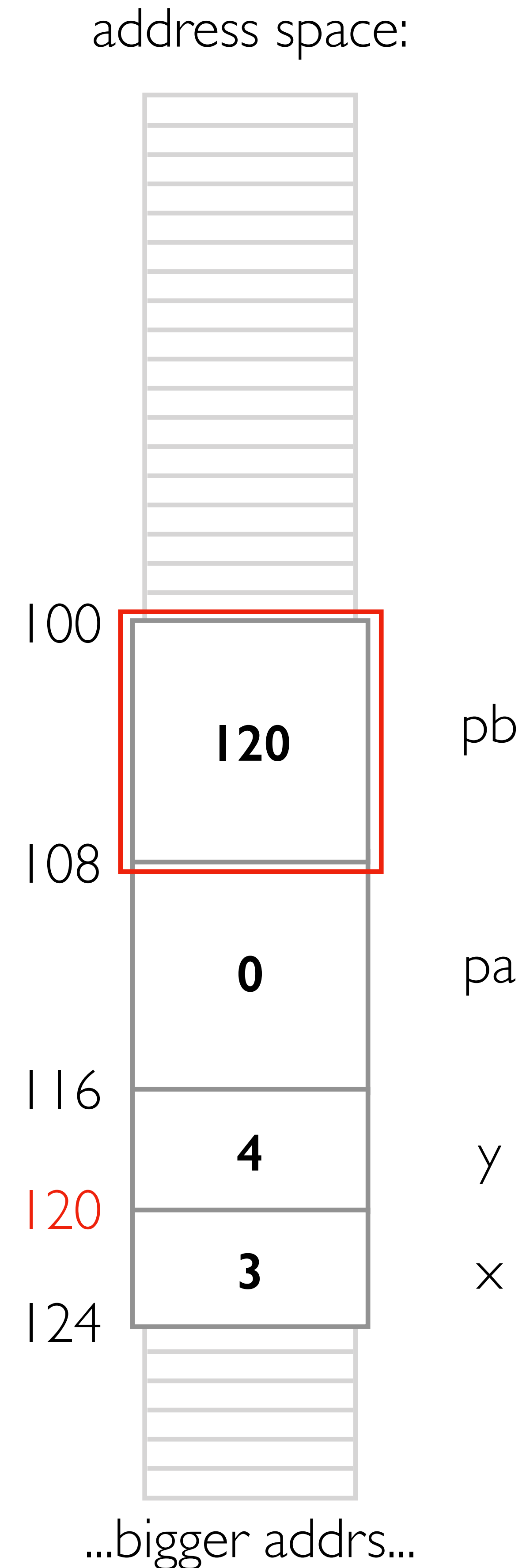
```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
}
```

store address of x value in pb variable



Expressions Involving Ptrs

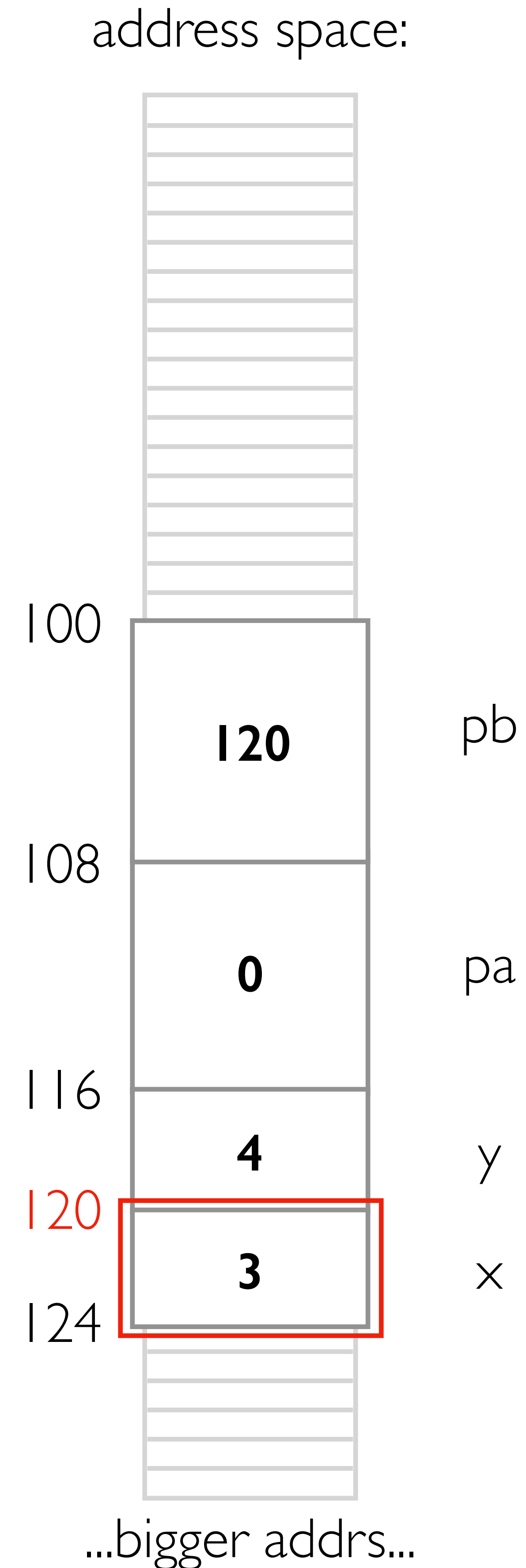
```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
  pb
}
```



Expressions Involving Ptrs

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
  *pb
}
```

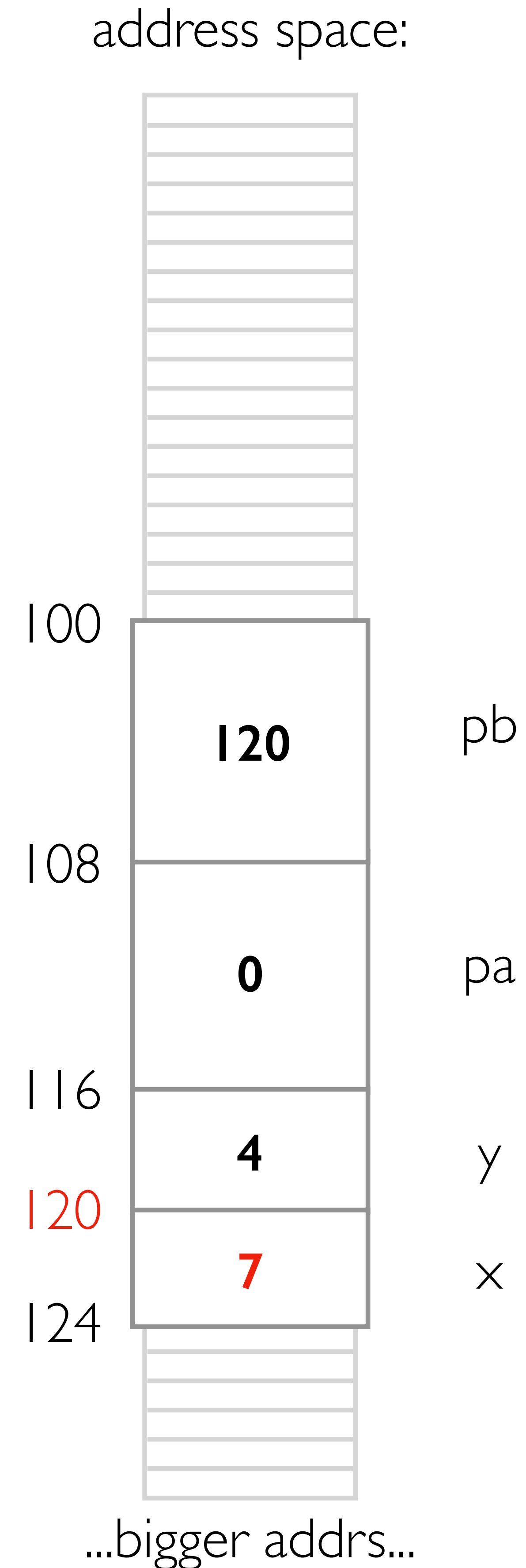
"*" is the indirection operator



Expressions Involving Ptrs

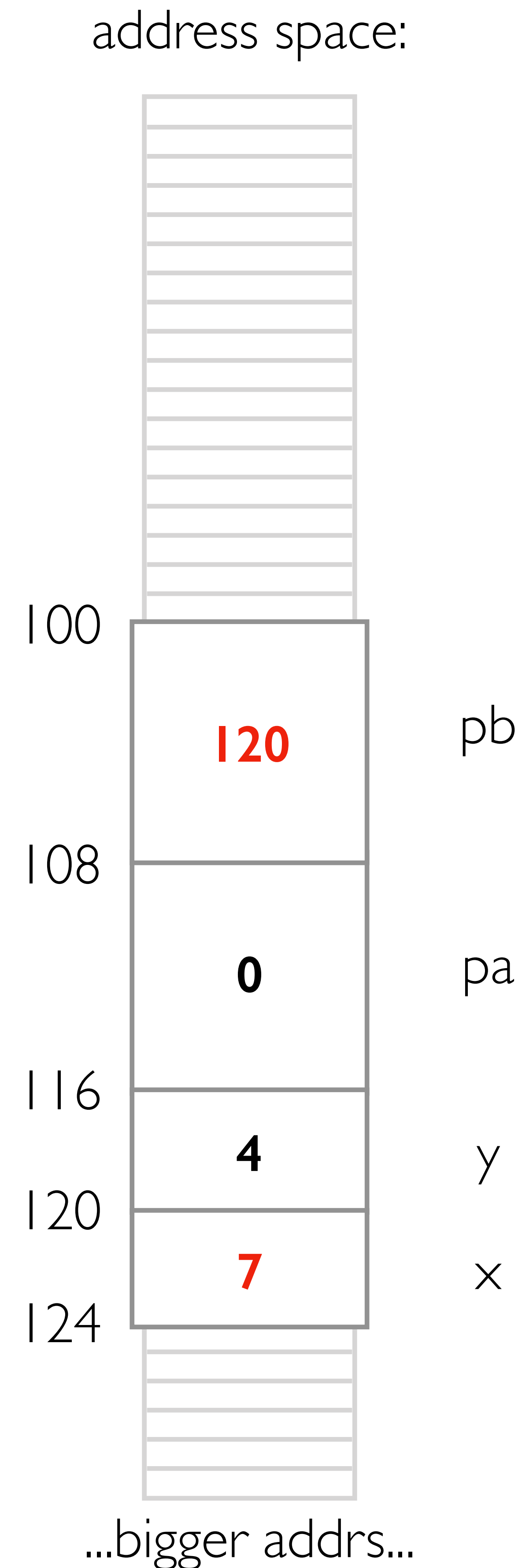
```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
  *pb = 7;
}
```

can use to update



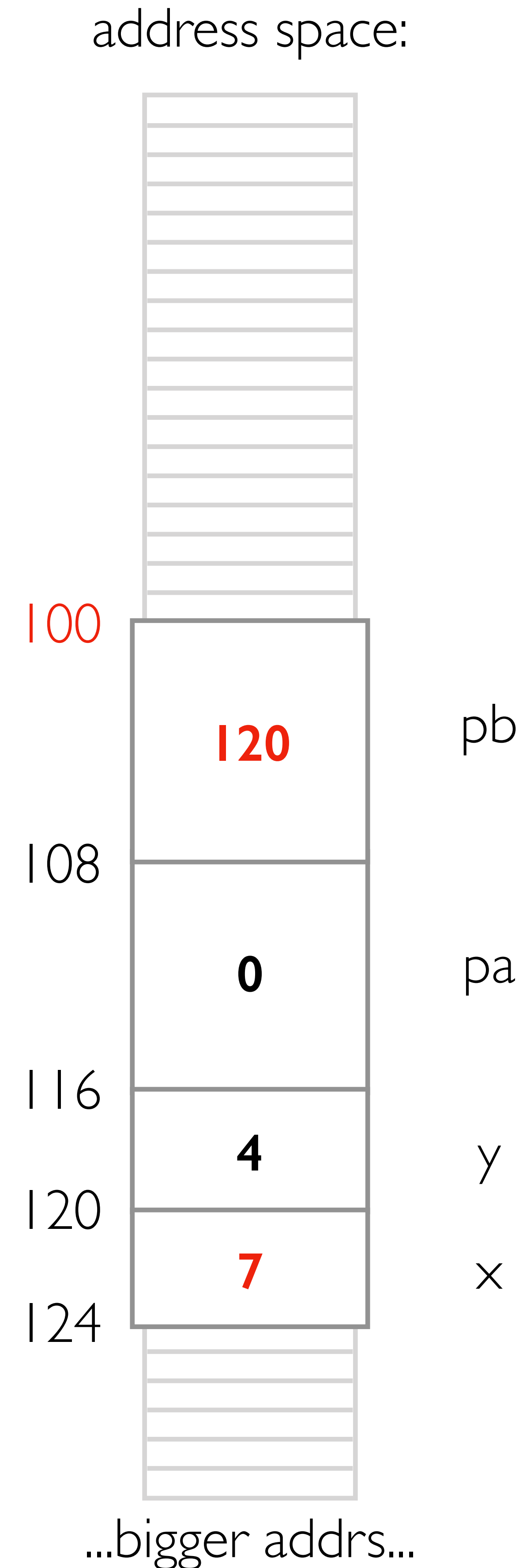
Expressions Involving Ptrs

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
  *pb = 7;
  cout << *pb << "\n";   7
  cout << pb  << "\n";  120
}
```



Addr of Pointer!

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr; // 0
  int *pb = nullptr; // 0
  pb = &x;
  *pb = 7;
  cout << *pb << "\n";    7
  cout << pb  << "\n";    120
  cout << &pb << "\n";    100
}
```

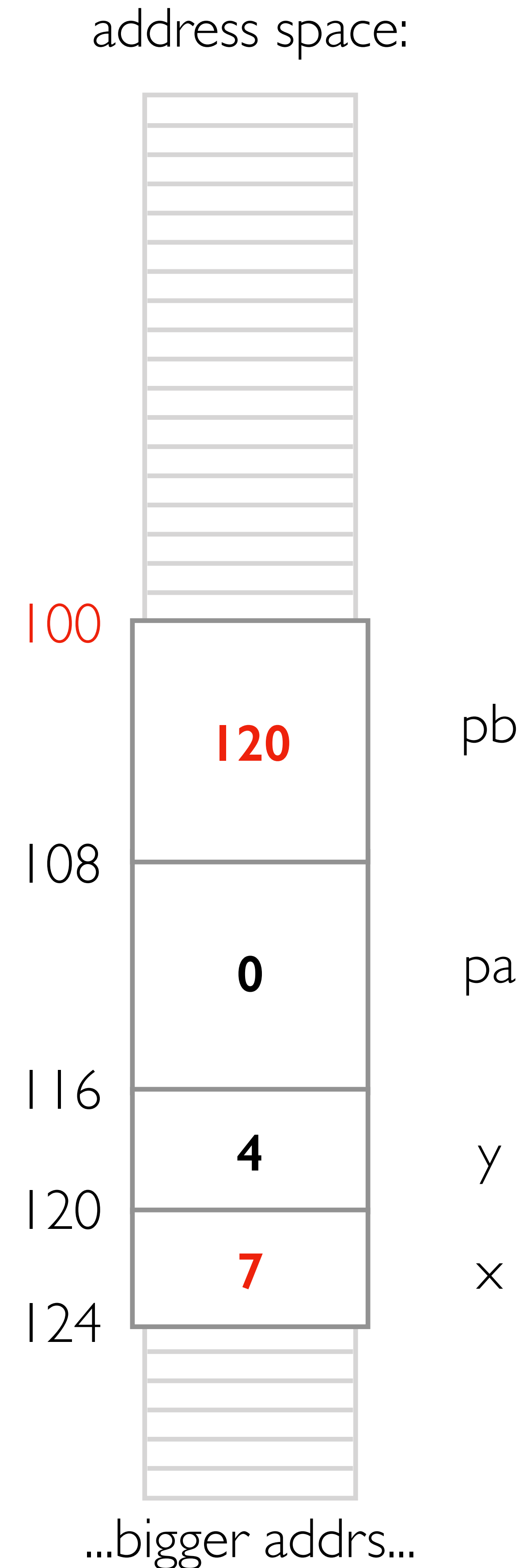


Watch Out!

```
int main() {
   int x{3};
   int y{4};
   int* pa = nullptr;
   int *pb = nullptr;
   pb = &x;
  *pb = 7;
}
```

NOT indirection op

indirection op



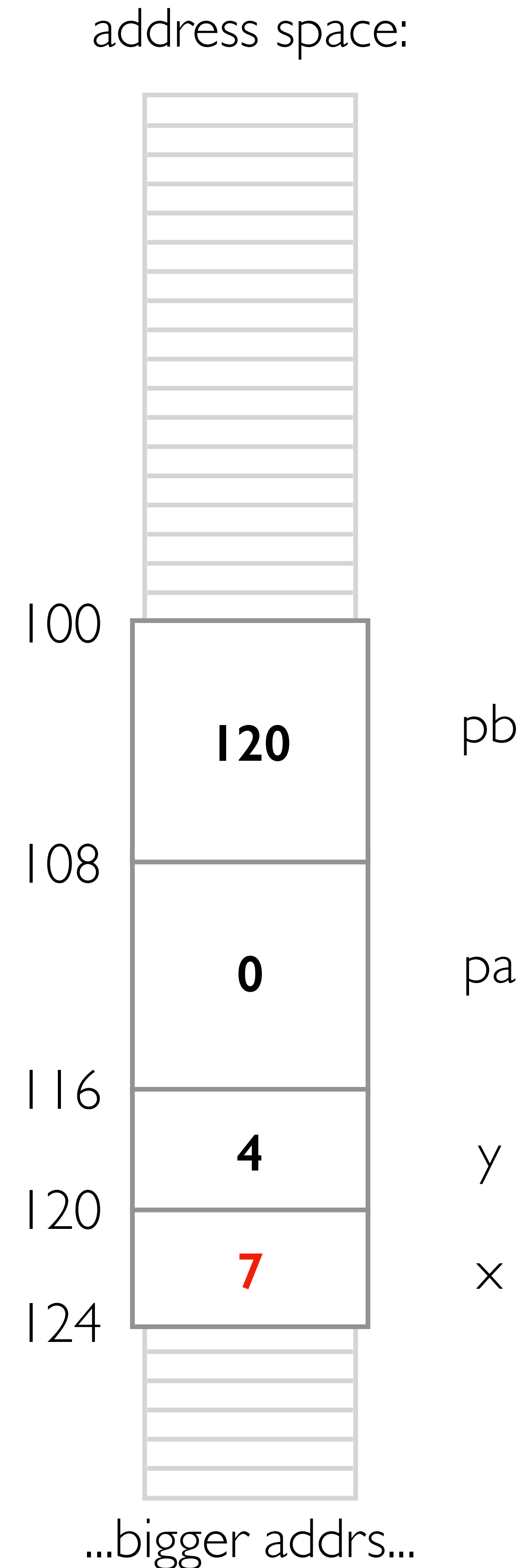
Pointer Init

```
int main() {
  int x{3};
  int y{4};
  int* pa = nullptr;
  int *pb = &x;
  *pb = 7;
}
```

make pb point to something

update value pb points to

"*" has different uses: creating a pointer type, and indirection

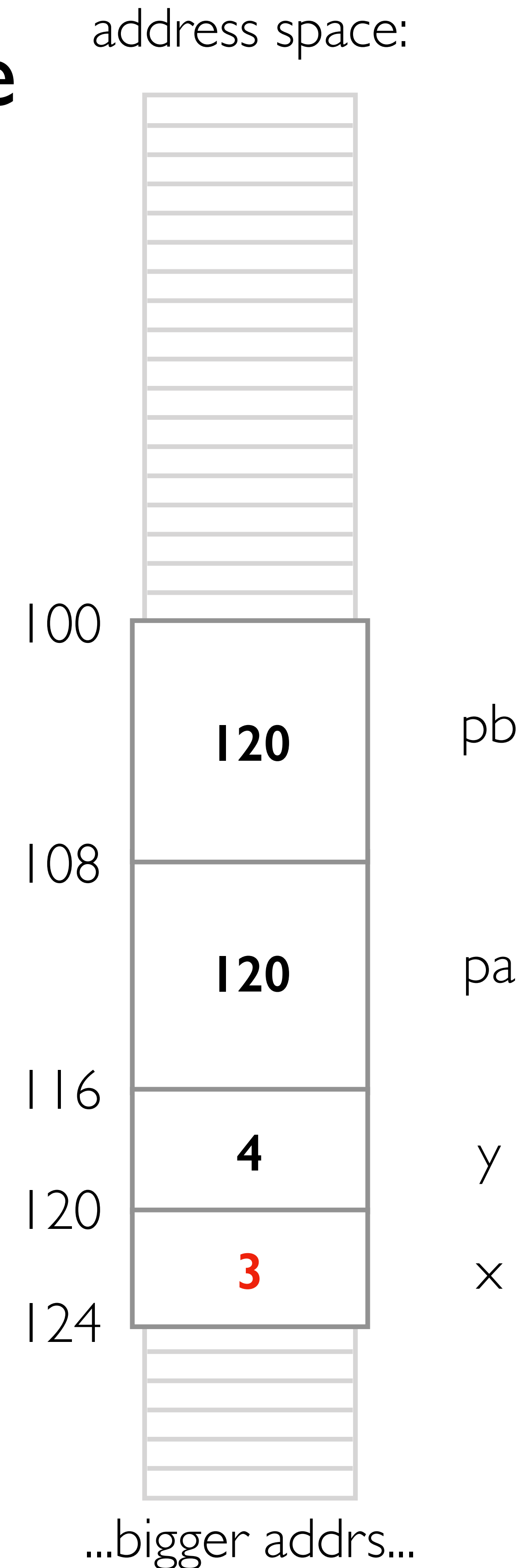


Many Ways to Get to a Value

```
int main() {
   int x{3};
   int y{4};
   int* pa = &x;
   int* pb = &x;
}
```

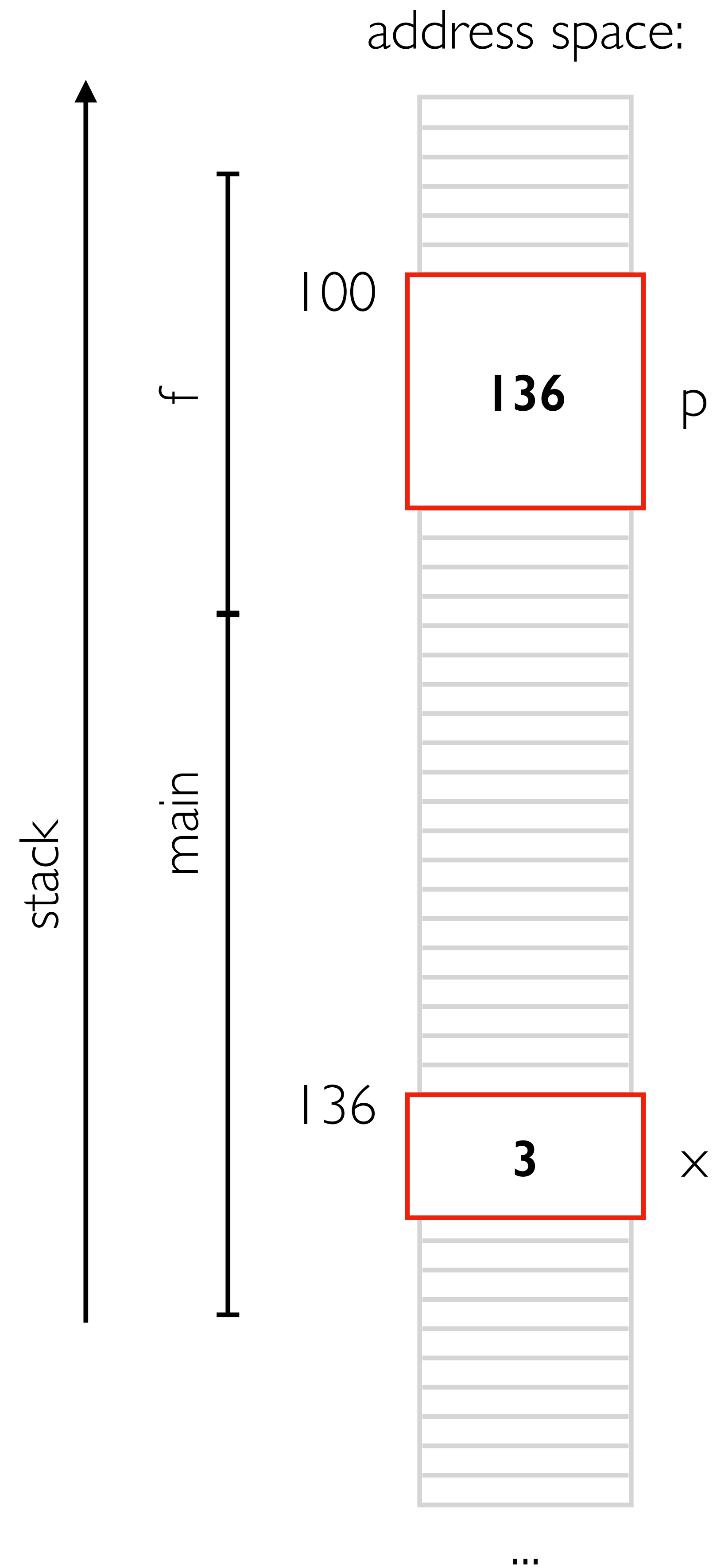
three ways to get to "3" value:

- x
- *pa
- *pb



Pass by Pointer

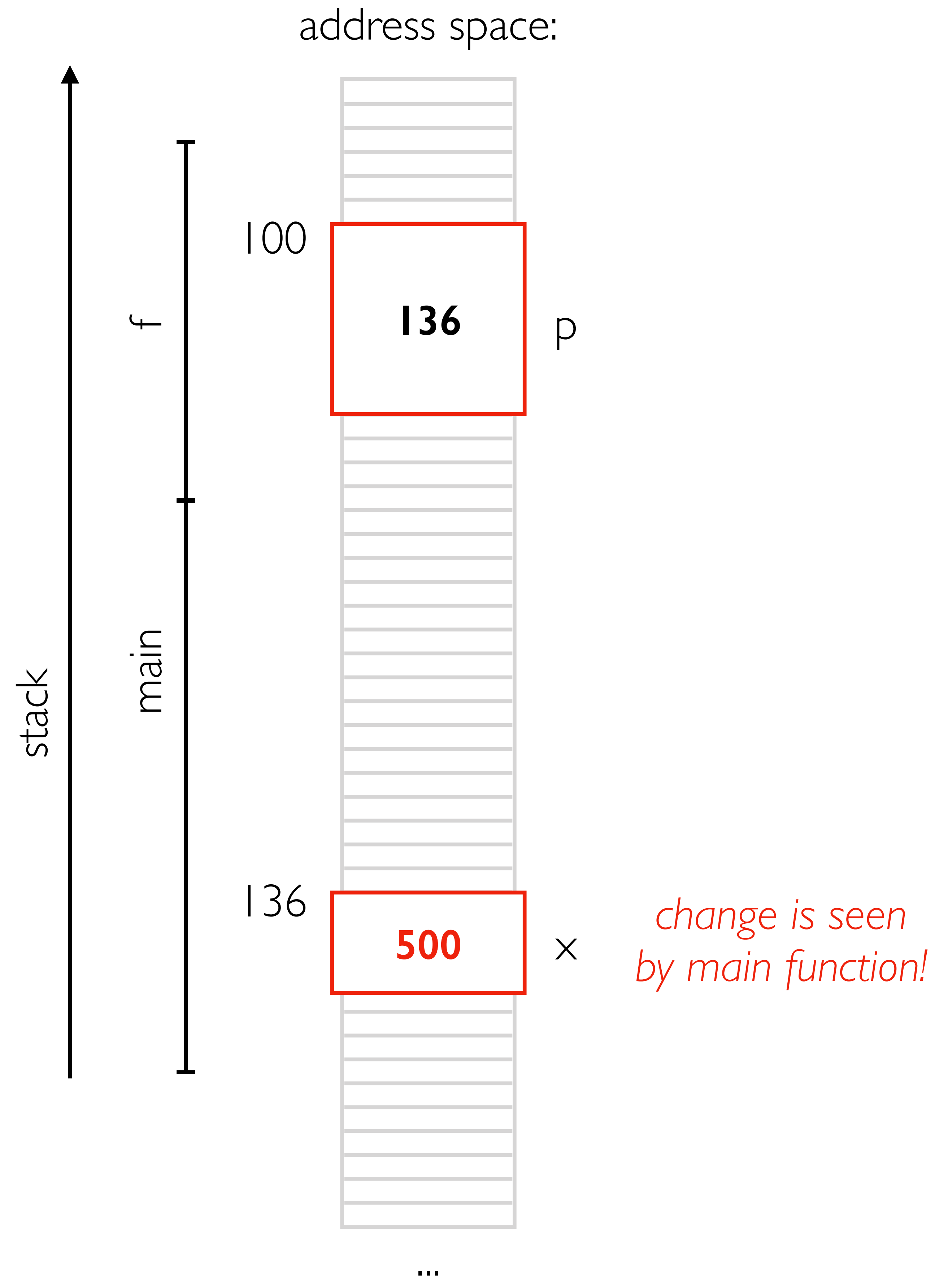
```
void f(int* p) {
  *p = 5 * 100;
}

int main() {
  int x{3};
  f(&x);
}
```



Pass by Pointer

```
void f(int* p) {
   *p = 5 * 100;
}

int main() {
   int x{3};
   f(&x);
}
```

modify



Outline

Project Organization Demos

Structs

Pointers

lvalues vs. rvalues

Stucts with Pointers

lvalue vs. rvalue

```
int main() {
  int *x = &3; // store address of 3 in pointer
  *x = 4;      // change 3 value
  std::cout << 3 << "\n";  // haha!
}
```

can we redefine 3?

lvalue vs. rvalue

```
int main() {
  int *x = &3; // store address of 3 in pointer
  *x = 4;      // change 3 value
  std::cout << 3 << "\n";  // haha!
}
```

can we redefine 3?

no

```
error: cannot take the address of an rvalue of type 'int'
```

related error:

```
expression is not assignable
```

lvalue vs. rvalue

```
int main() {
  int *x = &3; // store address of 3 in pointer
  *x = 4;      // change 3 value
  std::cout << 3 << "\n";  // haha!
}
```

can we redefine 3?

no

```
error: cannot take the address of an rvalue of type 'int'
```

lvalues are in memory somewhere, and you can get the address.

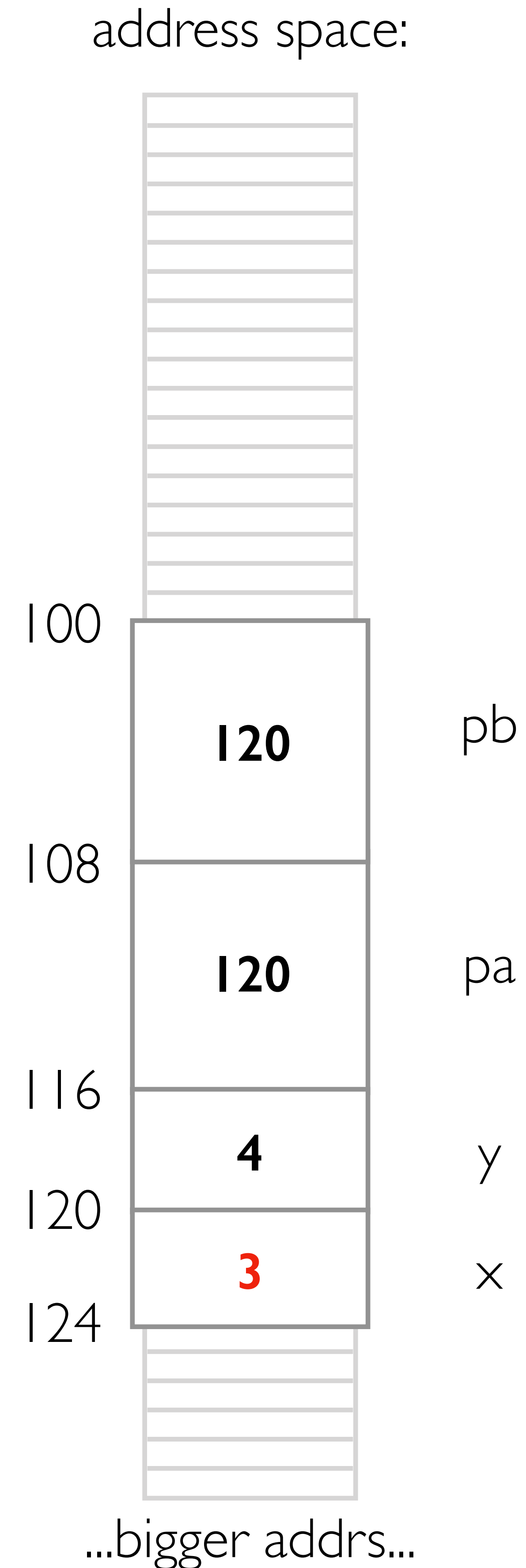
rvalues may or may not be in memory (could be in register, hardcoded, etc) -- you cannot get an address.

lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;
}
```

is pa the same as &*pa?

sort of...

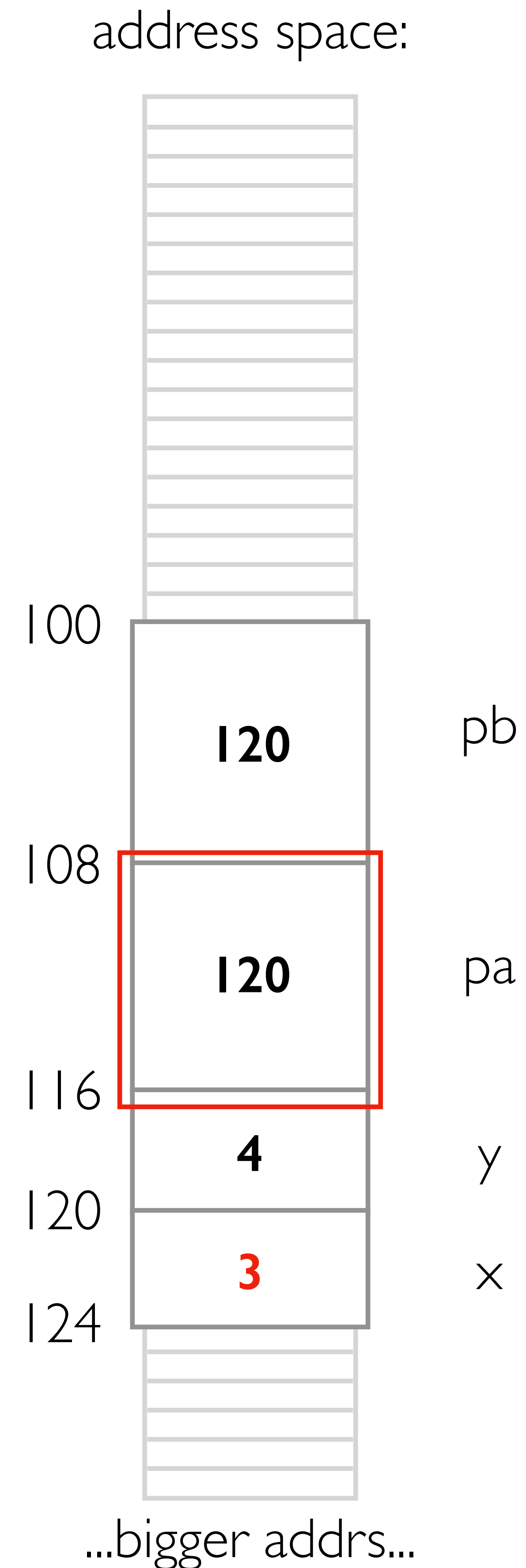


lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;
  pa
}
```

is pa the same as &*pa?

sort of...

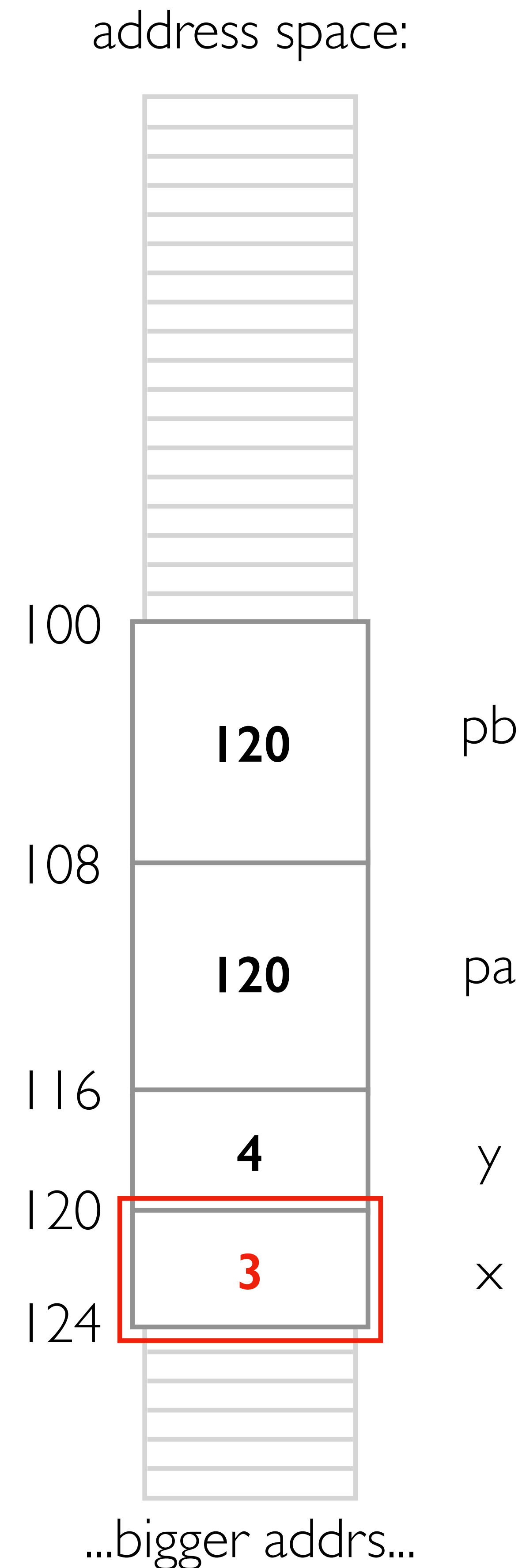


lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;
  *pa
}
```

is pa the same as &*pa?

sort of...

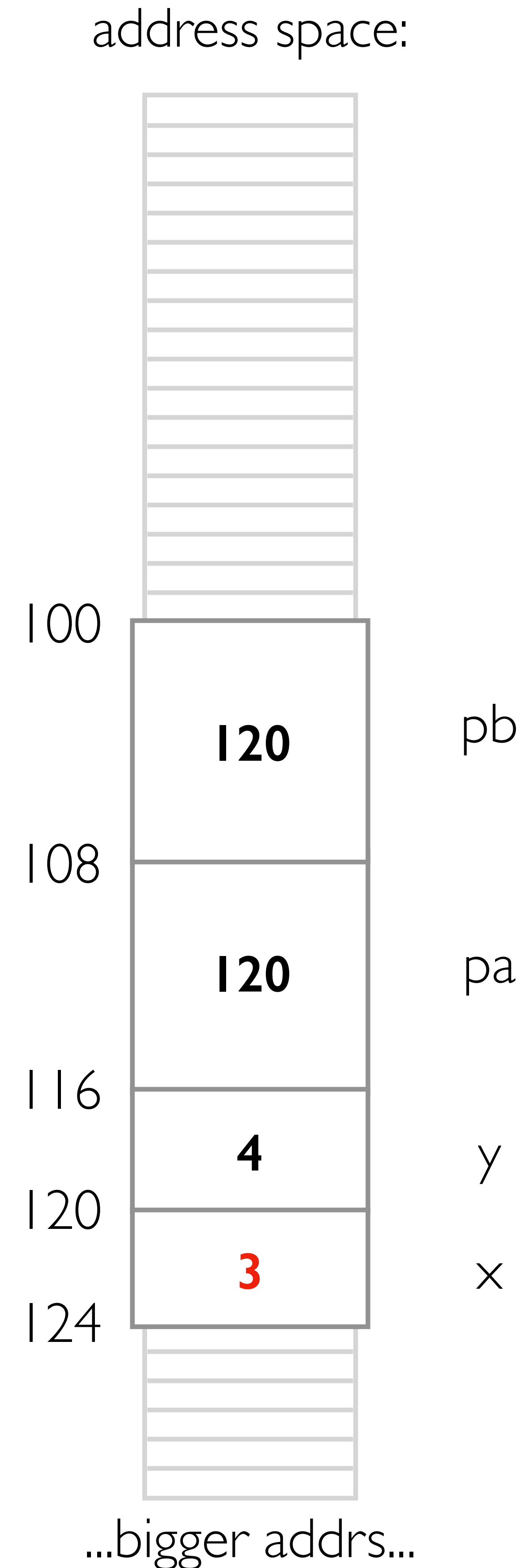


lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;
  &*pa
}
```

is pa the same as &*pa?

both are 120, but **pa is an lvalue**, and **&*pa is an rvalue**

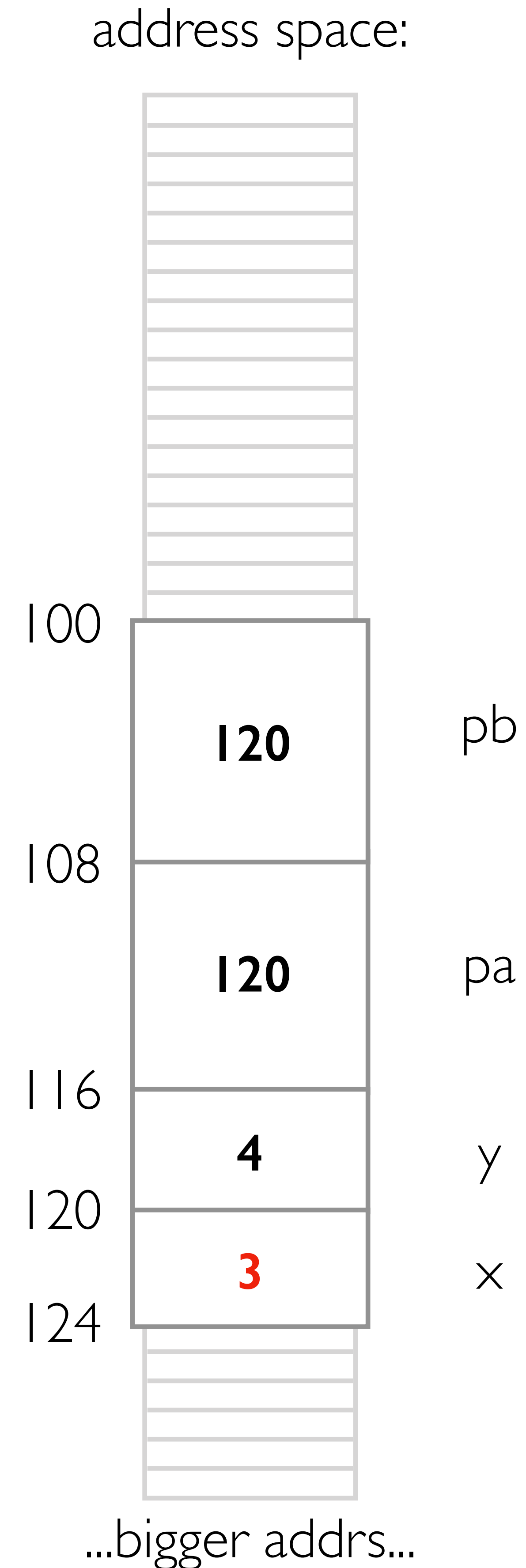


rvalue

120

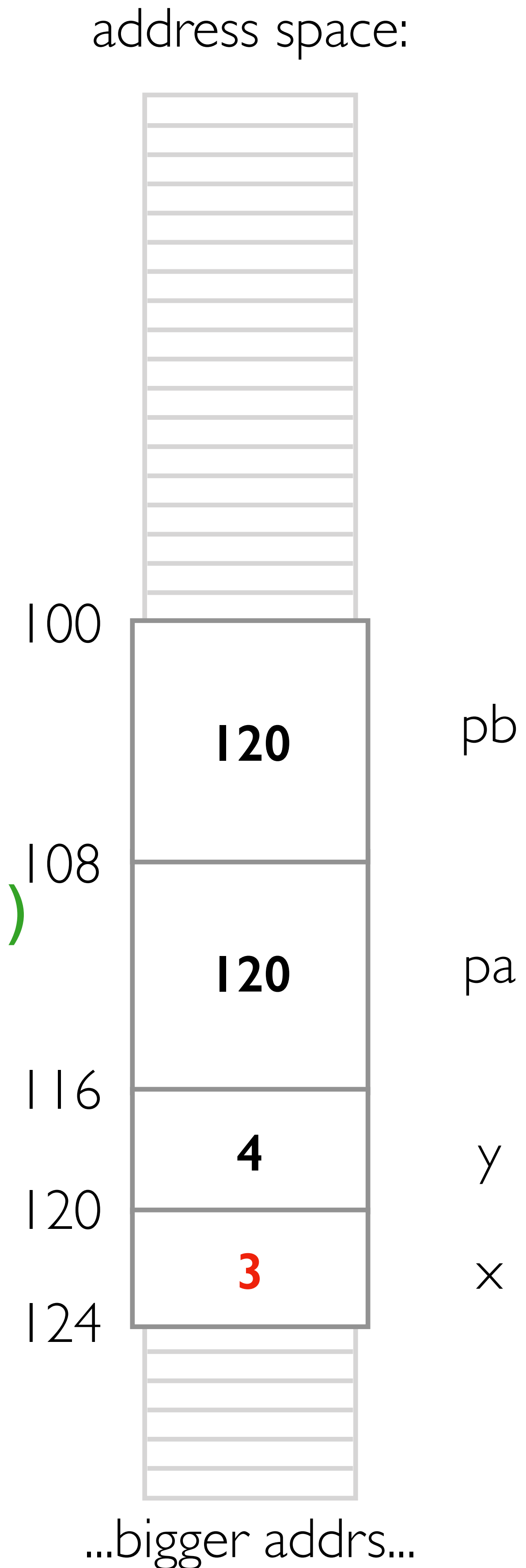
lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;

  // same (120):
  std::cout<< pa <<"\n";
  std::cout<< &*pa <<"\n";
}
```



lvalue vs. rvalue

```
int main() {
  int x{3};
  int y{4};
  int* pa = &x;
  int* pb = &x;

  // OK (lvalue on left)
  pa = &y;
  // not OK (rvalue on left)
  &*pa = &y;
}
```



Outline

Project Organization Demos

Structs

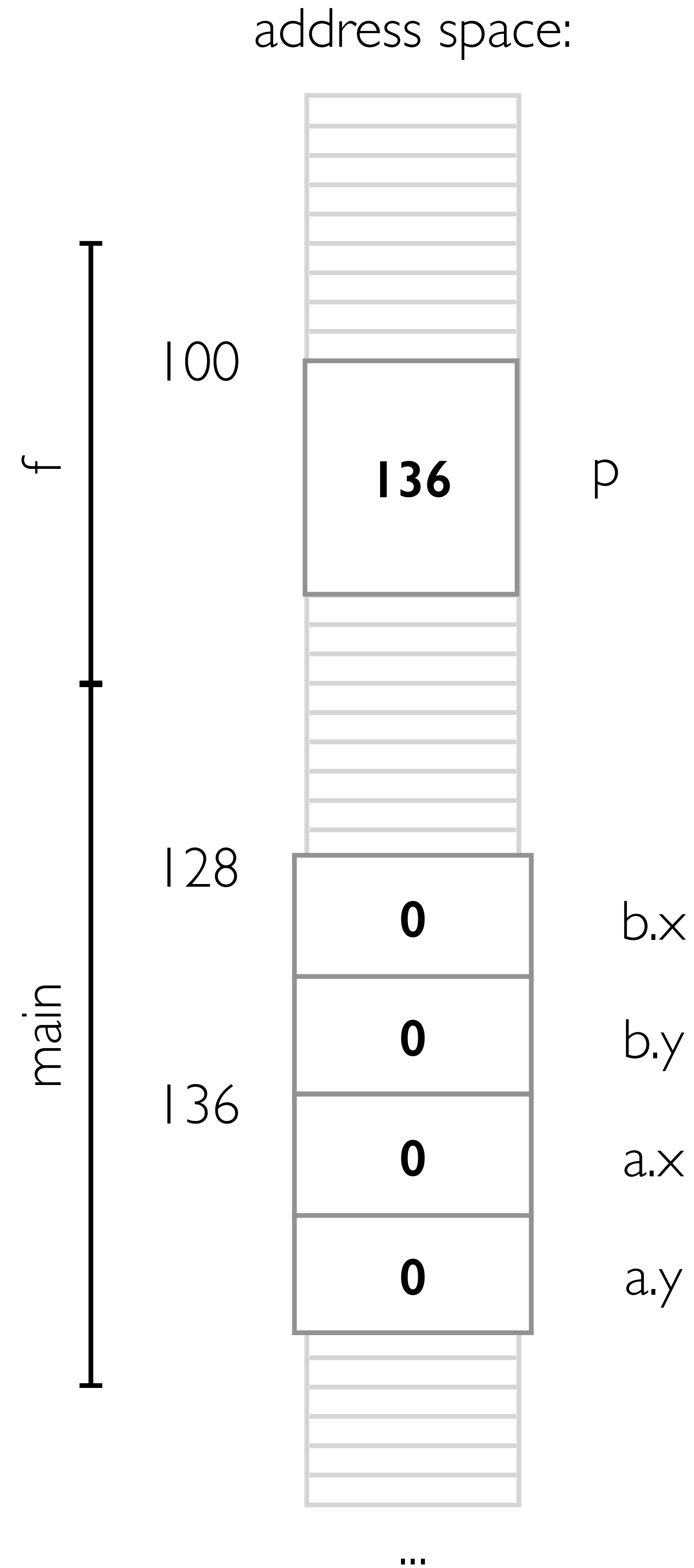
Pointers

lvalues vs. rvalues

Stucts with Pointers

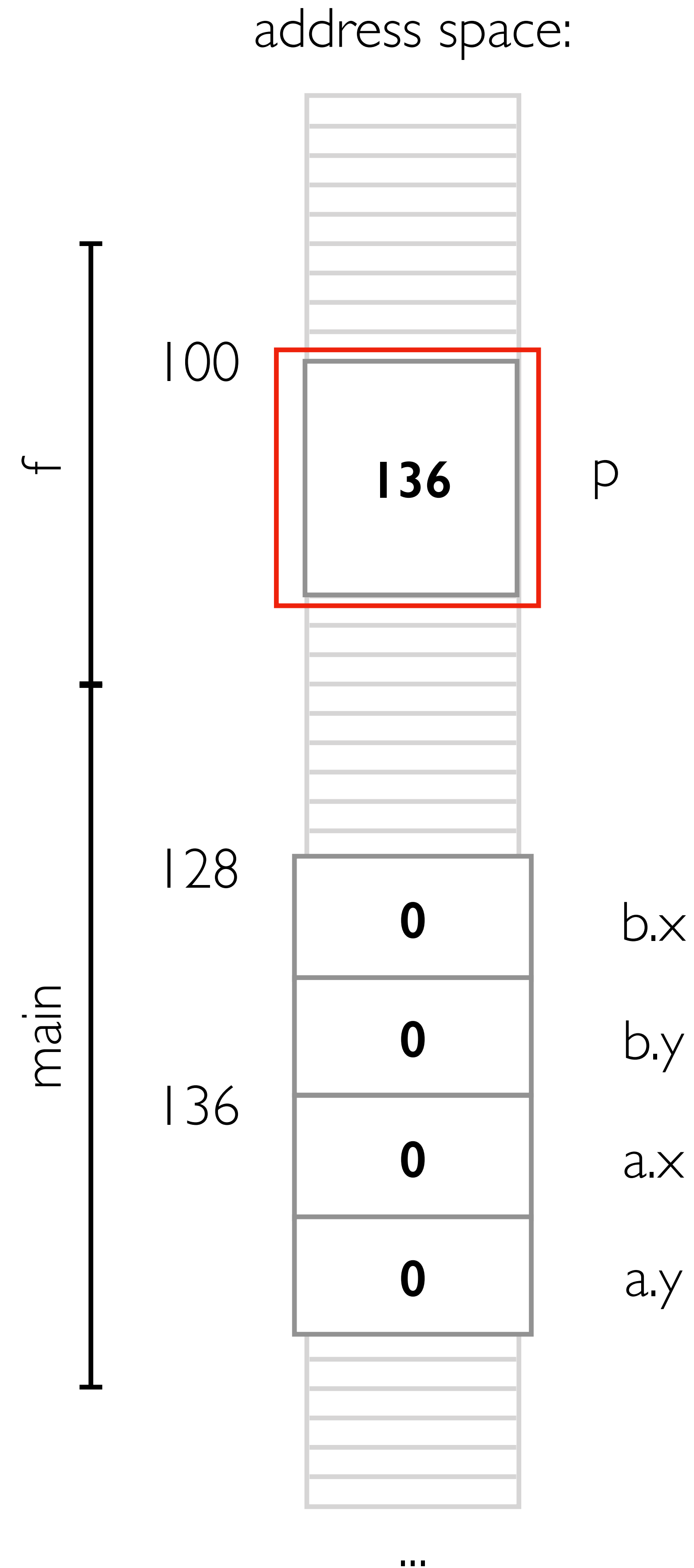
Structs with Pointers

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc* p) {
  // TODO: update x
}

int main() {
  Loc a{};
  Loc b{};
  f(&a);
}
```



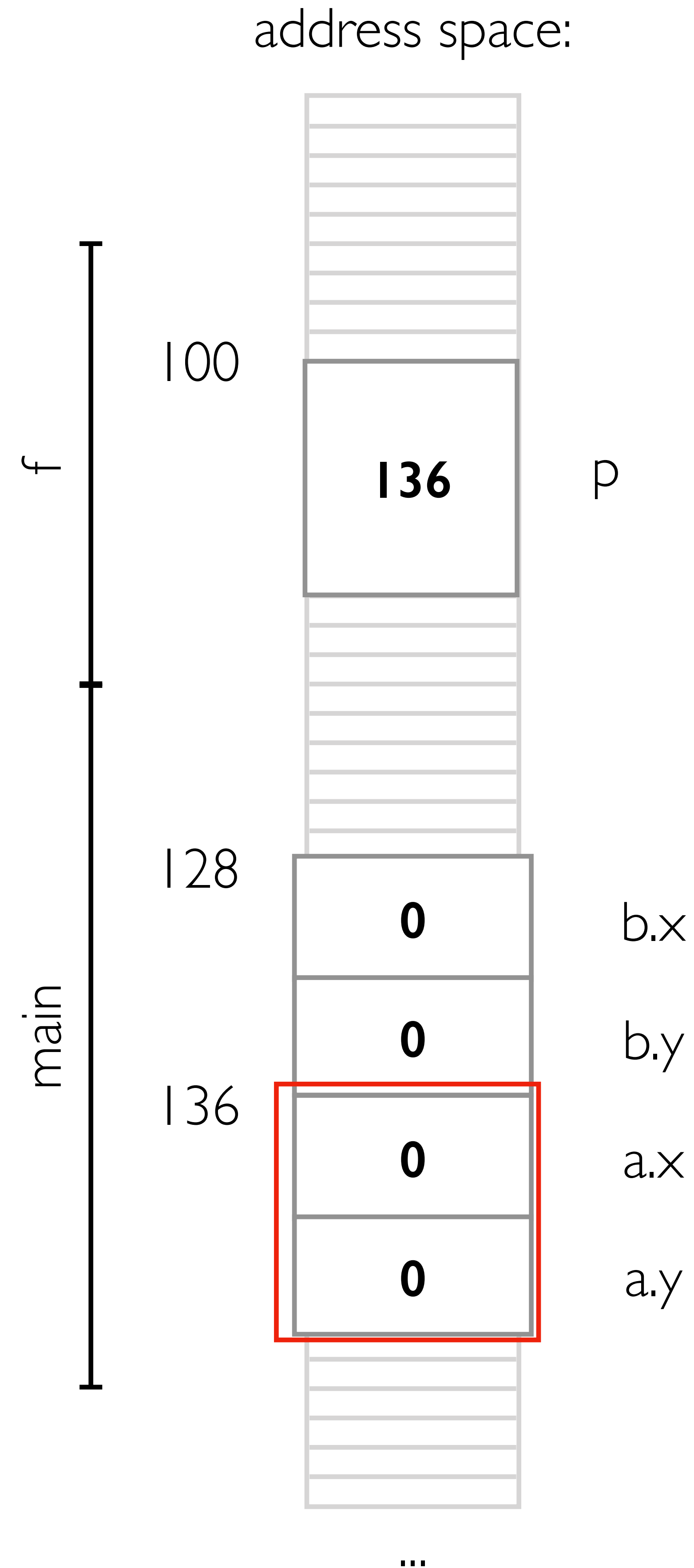
Structs with Pointers

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc* p) {
  p
}

int main() {
  Loc a{};
  Loc b{};
  f(&a);
}
```



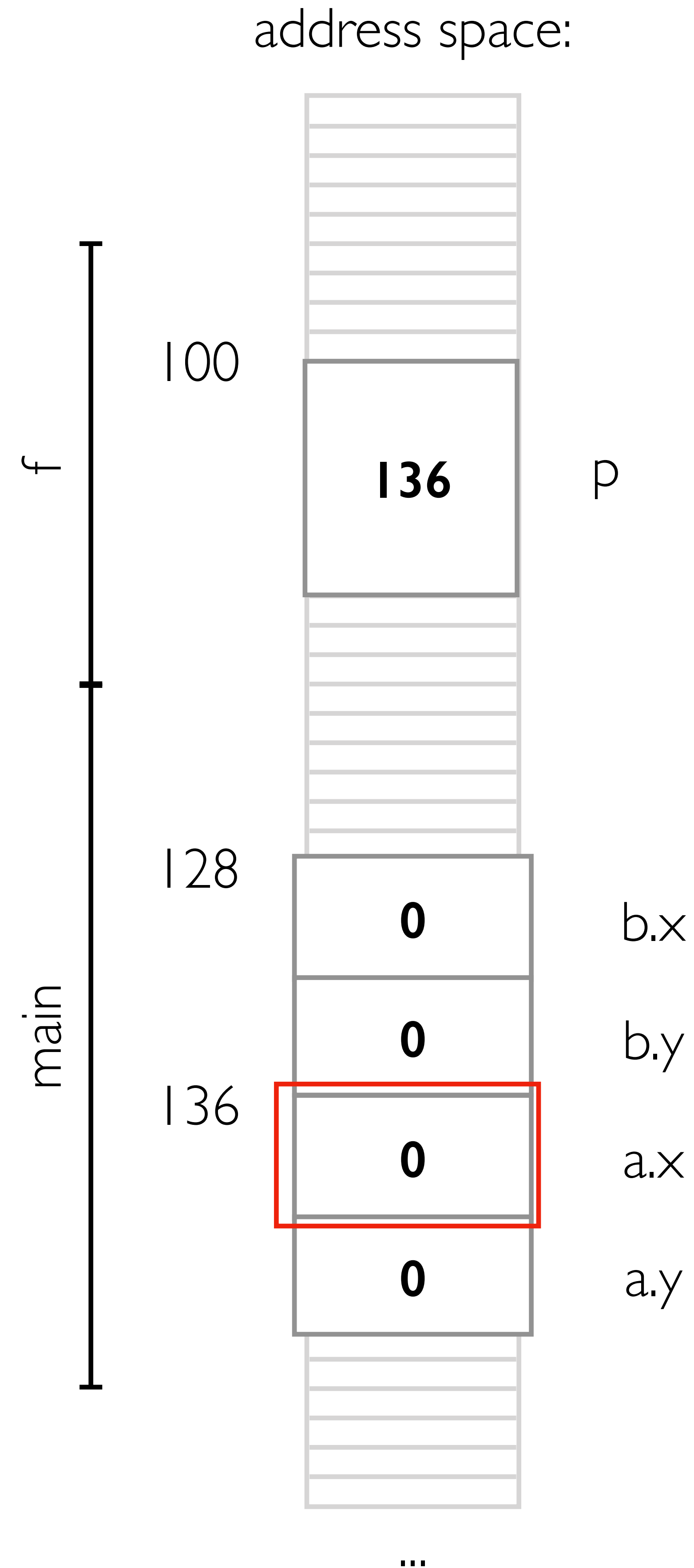
Structs with Pointers

```
struct Loc {
   int x = 0;
   int y = 0;
};

void f(Loc* p) {
   *p
}

int main() {
   Loc a{};
   Loc b{};
   f(&a);
}
```



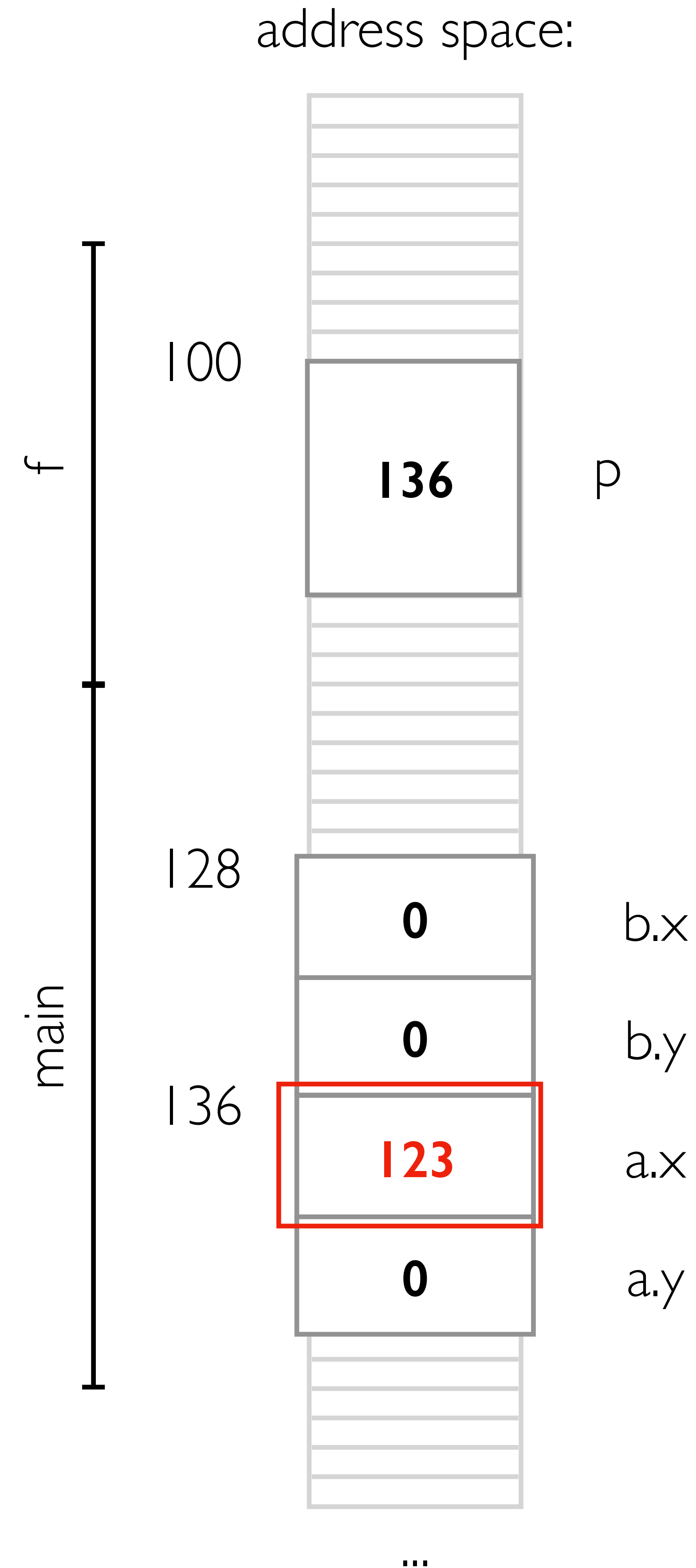
Structs with Pointers

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc* p) {
  (*p).x
}

int main() {
  Loc a{};
  Loc b{};
  f(&a);
}
```



Structs with Pointers

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc* p) {
  (*p).x = 123;
}

int main() {
  Loc a{};
  Loc b{};
  f(&a);
}
```



Structs with Pointers

```
struct Loc {
  int x = 0;
  int y = 0;
};

void f(Loc* p) {
  // same:
  (*p).x = 123;
  p->x = 123;
}

int main() {
  Loc a{};
  Loc b{};
  f(&a);
}
```