CPS 188

Computer Programming Fundamentals

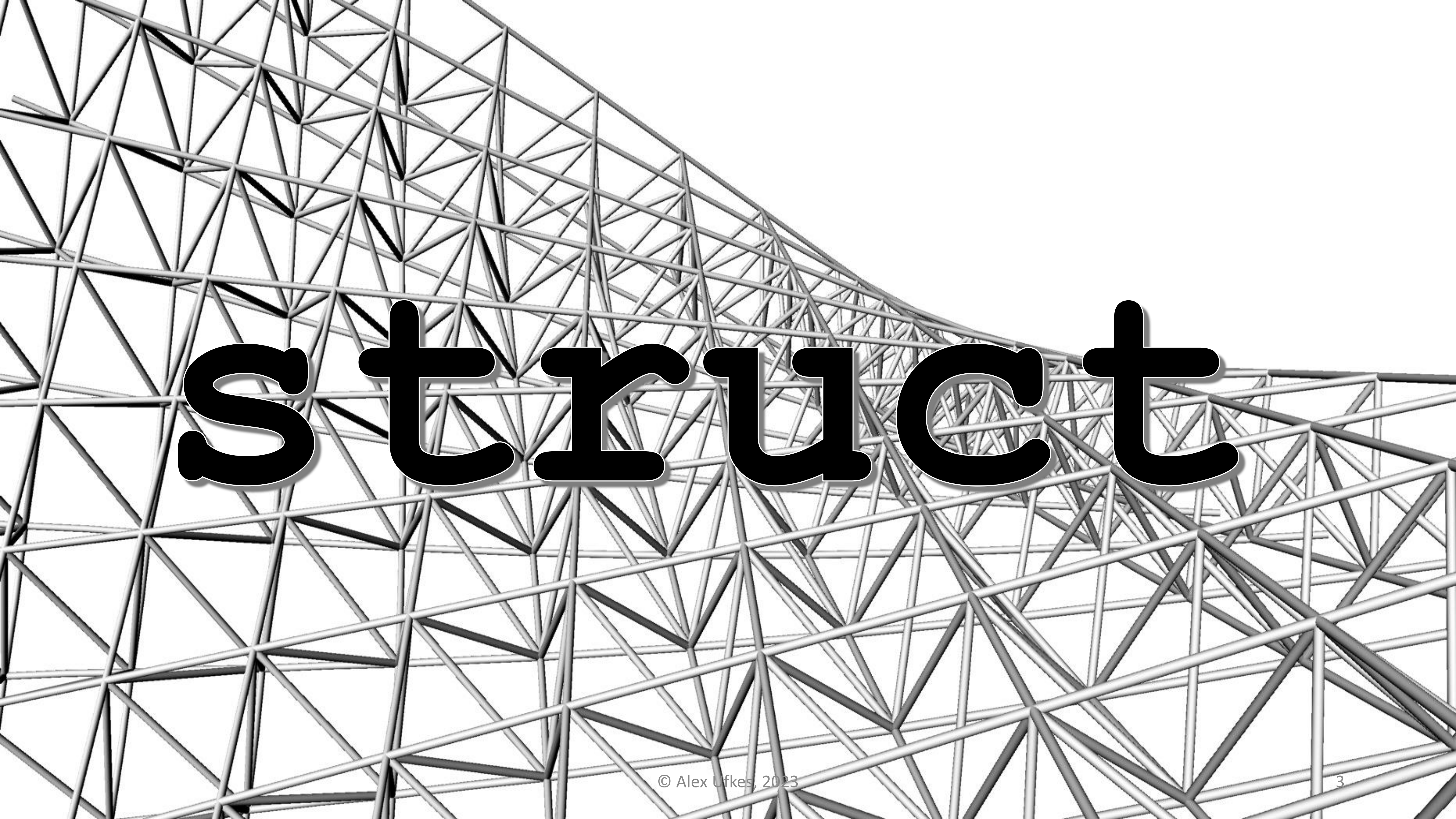
Prof. Alex Ufkes

Topic 10.1: Structures and unions

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:



struct

Structs

A collection of variables, types can vary

```
struct ball
{
   char style[10];
   double radius;
   double weight;
};
```

```c
#include <stdio.h>
#include <math.h>

struct ball
{
    char style[10];
    double radius;
    double weight;
};

int abc(int x);
double xyz();

int main()
{
    return 0;
}
```

Structure definitions:

- Typically placed between preprocessor directives and function prototypes.

struct variable declaration

```
struct ball
{
    char style[10];
    double radius;
    double weight;
};

int main()
{
    struct ball b1, b2, b3, b[50];


}
```

Or as an array

Declare individually

Accessing struct members

```
struct ball
{
    char style[10];
    double radius;
    double weight;
};

int main()
{
    struct ball b1;
    b1.radius = 4.6;
    b1.weight = 22.1;
    strcpy(b1.style, “Wilson”);
}
```

Use the dot operator to access a struct’s individual variables.

Arrays of structs

```
struct ball
{
    char style[10];
    double radius;
    double weight;
};

int main()
{
    struct ball b[50];
    int i;
    for (i = 0; i < 50; i++) {
        b[i].radius = 0.0;
        b[i].weight = 0.0;
    }
}
```

Initialize radius and weight to 0.0

`structure`
assignment

```
struct1 = struct2;
```

Assigns one struct to another

Assuming the same structure type

All member variable values are copied!

```
struct BirthdayInfo
{
    int year;
    int month;
    int day;
};
```

```
int main()
{
        struct BirthdayInfo x, y;
        x.year  = 1975;
        x.month = 8;
        x.day   = 15;
        y.year  = 2008;
        y.month = 12;
        y.day   = 2;
        y = x;
        printf(“%d/%d/%d\n”, y.year,
                     y.month, y.day);
}
```

```
y.year  = x.year;
y.month = x.month;
y.day   = x.day;
```

equivalent

```c
struct BirthdayInfo
{
    int year;
    int month;
    int day;
};
```

```c
int main()
{
     struct BirthdayInfo x, y;
     x.year  = 1975;
     x.month = 8;
     x.day   = 15;
     y.year  = 2008;
     y.month = 12;
     y.day   = 2;
     y = x;
     printf(“%d/%d/%d\n”, y.year,
            y.month, y.day);
}
```

stdout:

```
  1975/8/15
```

typedef

Create an Alias

```
#include <stdio.h>
typedef int integer;
int main()
{
    integer x, y, z;
    x = 1;
    y = 2;
    z = x + y;
}
```

Create "integer" alias for type int

`integer` can now be used as a type! It's the same as **`int`**, except for the name.

Common for Structures

```
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} ball;
```

Now, instead of declaring:

```
struct ball b1, b2, b3;
```

We can simply declare:

```
ball b1, b2, b3;
```

Common for Structures

Alias can be different from struct name:

```
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} basketball;
```

Declaration:

```
basketball b1, b2, b3;
```

Can still declare:

```
struct ball a, b, c;
```

```
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} basketball;

int main()
{
    basketball b1 = {“Wilson”, 3.4, 6.8};

    return 0;
}
```

Declare and Initialize

Order must match structure definition

```
typedef struct ball
{
   char style[10];
   double radius;
   double weight;
} basketball;


int main()
{
     basketball b1, b2;

     printf("What is the radius?");
     scanf("%lf", &b1.radius);

     return 0;
}
```

Read into struct fields

Can read values directly into a struct's member variables.

Structures & Functions

Structures passed as input to a function are **COPIED** into the corresponding function parameter. All member variables are copied as well.

This is **DIFFERENT** from arrays, which are **NOT** copied. Only the base address of the array is copied, not the elements.

```c
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} basketball;
```

struct ball is defined and given an alias *before* the function printElements

```c
void printElements(basketball b)
{
    printf(“Style:  %s\n”, b.style);
    printf(“Radius: %lf\n”, b.radius);
    printf(“Weight: %lf\n”, b.weight);
}
```

```
typedef struct ball
{
   char style[10];
   double radius;
   double weight;
} basketball;
```

```
void printElements(basketball b)
{
   printf(“Style:  %s\n”, b.style);
   printf(“Radius: %lf\n”, b.radius);
   printf(“Weight: %lf\n”, b.weight);
}
```

b1 and its member values are copied into b

```
int main()
{
     basketball b1 = {“Wilson”, 11.4, 212.8};

     printElements(b1);

     return 0;
}
```

Write a user-defined function that does the following:

Takes in two basketball structs as arguments and compares the radius and the weight. Returns 1 if they are the same, and 0 if they are different.

Two basketball structs as input arguments

```
int comp(basketball b1, basketball b2)
{
    int isEqual;

    isEqual = (b1.radius == b2.radius) &&
              (b1.weight == b2.weight);

    return isEqual;
}
```

What is the output of a logical expression?

```c
int comp(basketball b1, basketball b2)
{
    return (b1.radius == b2.radius) && (b1.weight == b2.weight);
}

int main()
{
    basketball b1 = {“Wilson”, 11.4, 212.8};
    basketball b2 = {“Spalding”, 11.4, 212.8};
    if (comp(b1, b2))
        printf(“b1 and b2 are the same!\n”);
    else
        printf(“b1 and b2 are NOT the same!\n”);
    return 0;
}
```

Write a user-defined function that does the following:

Takes in zero arguments, and returns a `basketball` struct. The function will ask the user to enter values for each of `basketball`'s member variables.

```
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} basketball;
```

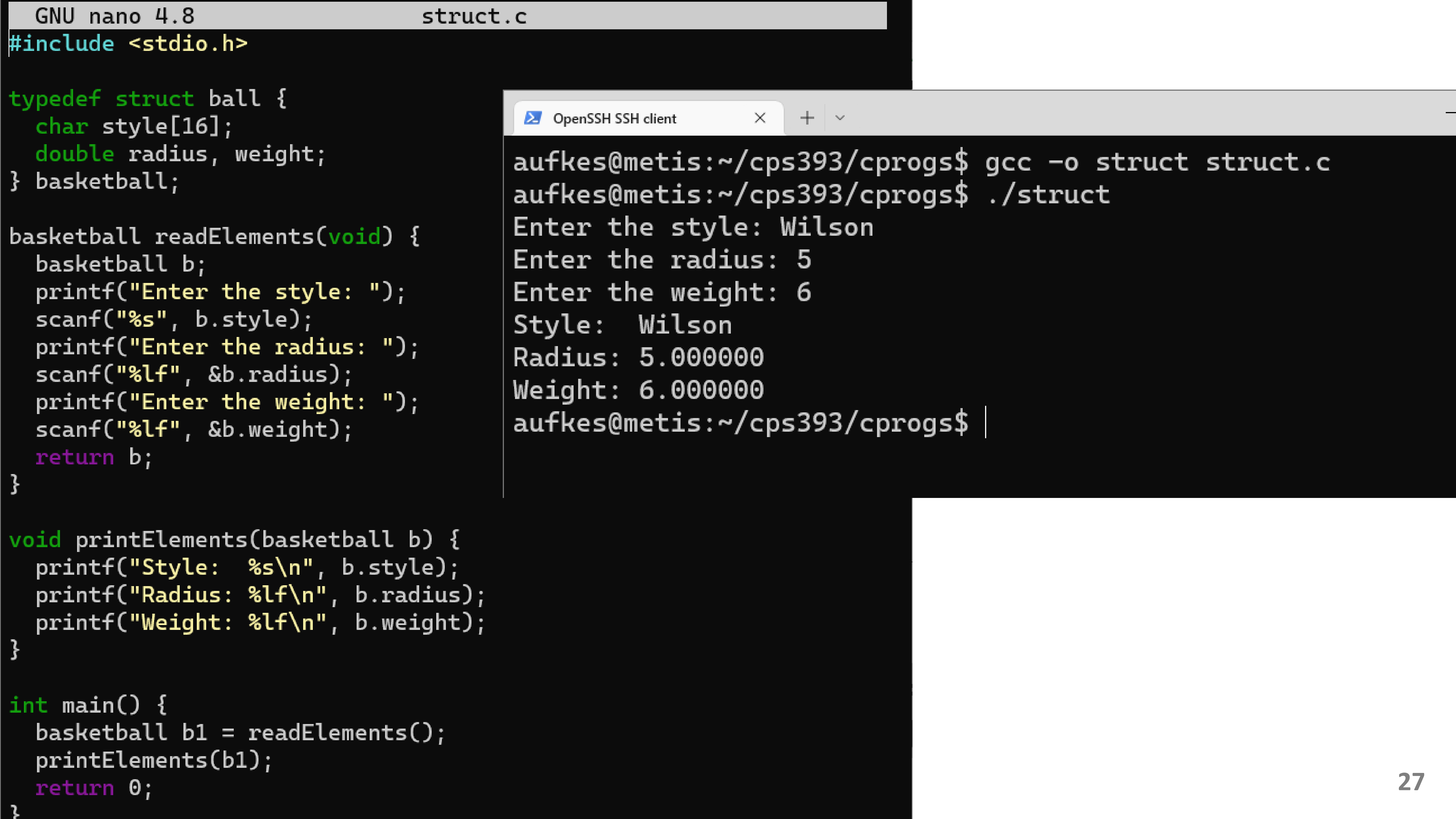
```
basketball readElements(void)
{
    basketball b;
    printf(“Enter the style: ”);
    scanf(“%s”, b.style);
    printf(“Enter the radius: ”);
    scanf(“%lf”, &b.radius);
    printf(“Enter the weight: ”);
    scanf(“%lf”, &b.weight);
    return b;
}
```

```
basketball readElements(void)
{
      basketball b;
      printf(“Enter the style: ”);
      scanf(“%s”, b.style);
      printf(“Enter the radius: ”);
      scanf(“%lf”, &b.radius);
      printf(“Enter the weight: ”);
      scanf(“%lf”, &b.weight);
      return b;
}
```

```
void printElements(basketball b)
{
   printf(“Style:  %s\n”, b.style);
   printf(“Radius: %lf\n”, b.radius);
   printf(“Weight: %lf\n”, b.weight);
}
```

It is possible to return an entire `struct`. *Unlike* arrays, ALL values are copied.

```
int main()
{
      basketball b1 = readElements();
      printElements(b1);
      return 0;
}
```



```c
#include <stdio.h>

typedef struct ball {
  char style[16];
  double radius, weight;
} basketball;

basketball readElements(void) {
  basketball b;
  printf("Enter the style: ");
  scanf("%s", b.style);
  printf("Enter the radius: ");
  scanf("%lf", &b.radius);
  printf("Enter the weight: ");
  scanf("%lf", &b.weight);
  return b;
}

void printElements(basketball b) {
  printf("Style:  %s\n", b.style);
  printf("Radius: %lf\n", b.radius);
  printf("Weight: %lf\n", b.weight);
}

int main() {
  basketball b1 = readElements();
  printElements(b1);
  return 0;
}
```

```
aufkes@metis:~/cps393/cprogs$ gcc -o struct struct.c
aufkes@metis:~/cps393/cprogs$ ./struct
Enter the style: Wilson
Enter the radius: 5
Enter the weight: 6
Style:  Wilson
Radius: 5.000000
Weight: 6.000000
aufkes@metis:~/cps393/cprogs$
```

Consider:

```
void swap(struct foo input)
{
   int tmp = input.a;
   input.a = input.b;
   input.b = tmp;
}
```

```
struct foo {
   int a, b;
};

int main()
{
  struct foo f1 = {1, 2};
  printf(“Before swap: %d %d”, f1.a, f1.b);
  swap(f1);
  printf(“After swap: %d %d”, f1.a, f1.b);
  return 0;
}
```

```
void swap(struct foo input)
{
    int tmp = input.a;
    input.a = input.b;
    input.b = tmp;
}

int main()
{
    struct foo f1 = {1, 2};
    printf(“Before swap: %d %d”, f1.a, f1.b);
    swap(f1);
    printf(“After swap:  %d %d”, f1.a, f1.b);
    return 0;
}
```

F1 is copied into input.

What is the output?

```
Before swap: 1 2
After swap:  1 2
```

POINTERS

Pass a pointer:

```
typedef struct foo {
    int a, b;
} foo;
```

```
void swap(foo input)
{
     int tmp = input.a;
     input.a = input.b;
     input.b = tmp;
}
```

```
void swap(foo *input)
{
   int tmp = input->a;
   input->a = input->b;
   input->b = tmp;
}
```

If `input` is a ***pointer*** to a struct, we use the **arrow operator (->)** instead of the dot operator.

```
void swap(foo *input)
{
   int tmp = input->a;
   input->a = input->b;
   input->b = tmp;
}

int main()
{
     foo f1 = {1, 2};
     printf(“Before swap: %d %d\n”, f1.a, f1.b);
     swap(&f1);
     printf(“After swap:  %d %d”, f1.a, f1.b);
     return 0;
}
```

What is the output?

```
Before swap: 1 2
After swap:  2 1
```

We can also dereference!

```
void printElements(basketball *b)
{
   printf(“Style:  %s\n”,  b->style);
   printf(“Radius: %lf\n”, b->radius);
   printf(“Weight: %lf\n”, b->weight);
}
```

```
typedef struct ball
{
    char style[10];
    double radius;
    double weight;
} basketball;
```

Is the same as:

```
void printElements(basketball *b)
{
   printf(“Style:  %s\n”,  (*b).style);
   printf(“Radius: %lf\n”, (*b).radius);
   printf(“Weight: %lf\n”, (*b).weight);
}
```

Must use parentheses (*b) otherwise the dot operator is applied first

Enums

and

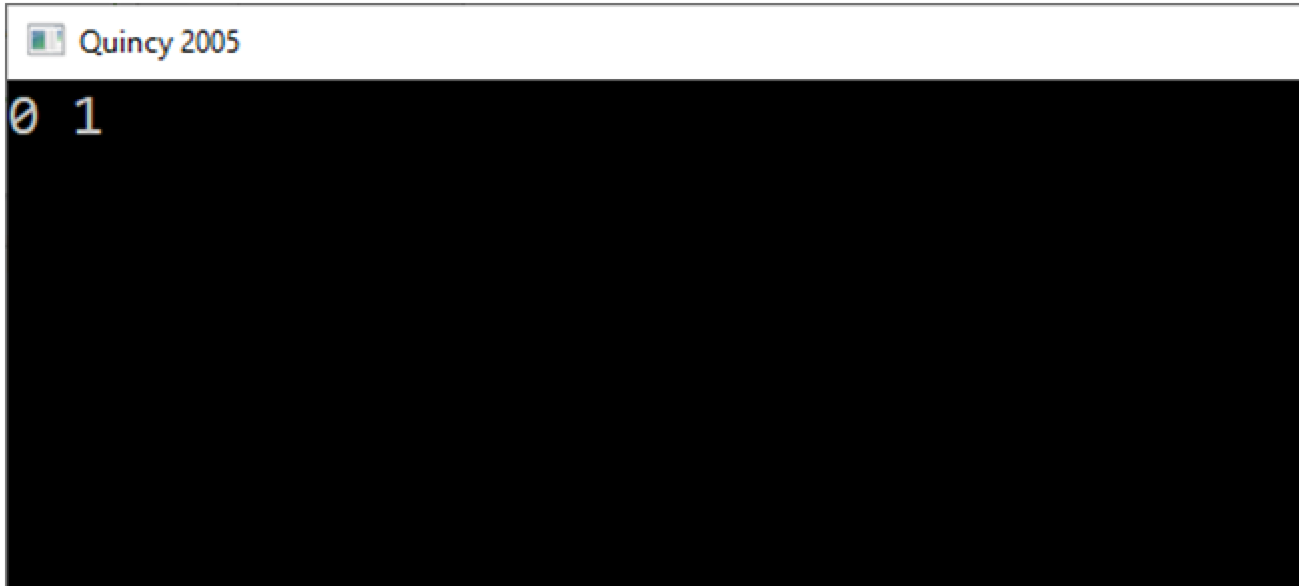
Unions

Enums

Enumerations allow us to define a custom type AND the values that type can take:

```
#include <stdio.h>
enum boolean {FALSE, TRUE};
int main()
{
   enum boolean f, t;
   f = FALSE;
   t = TRUE;
   printf("%d %d\n", f, t);
   return 0;
}
```

- The enum values (FALSE, TRUE) alias their numeric index in the enum
- FALSE == 0, TRUE == 1



```
0 1
```

Enums

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

```
for (int i = Sun; i <= Sat; i++)
   printf(“Day %d\n”, i);
```

What prints? `Day 0, Day 1, …, Day 5, Day 6`

Numbering starts at 0 by default.

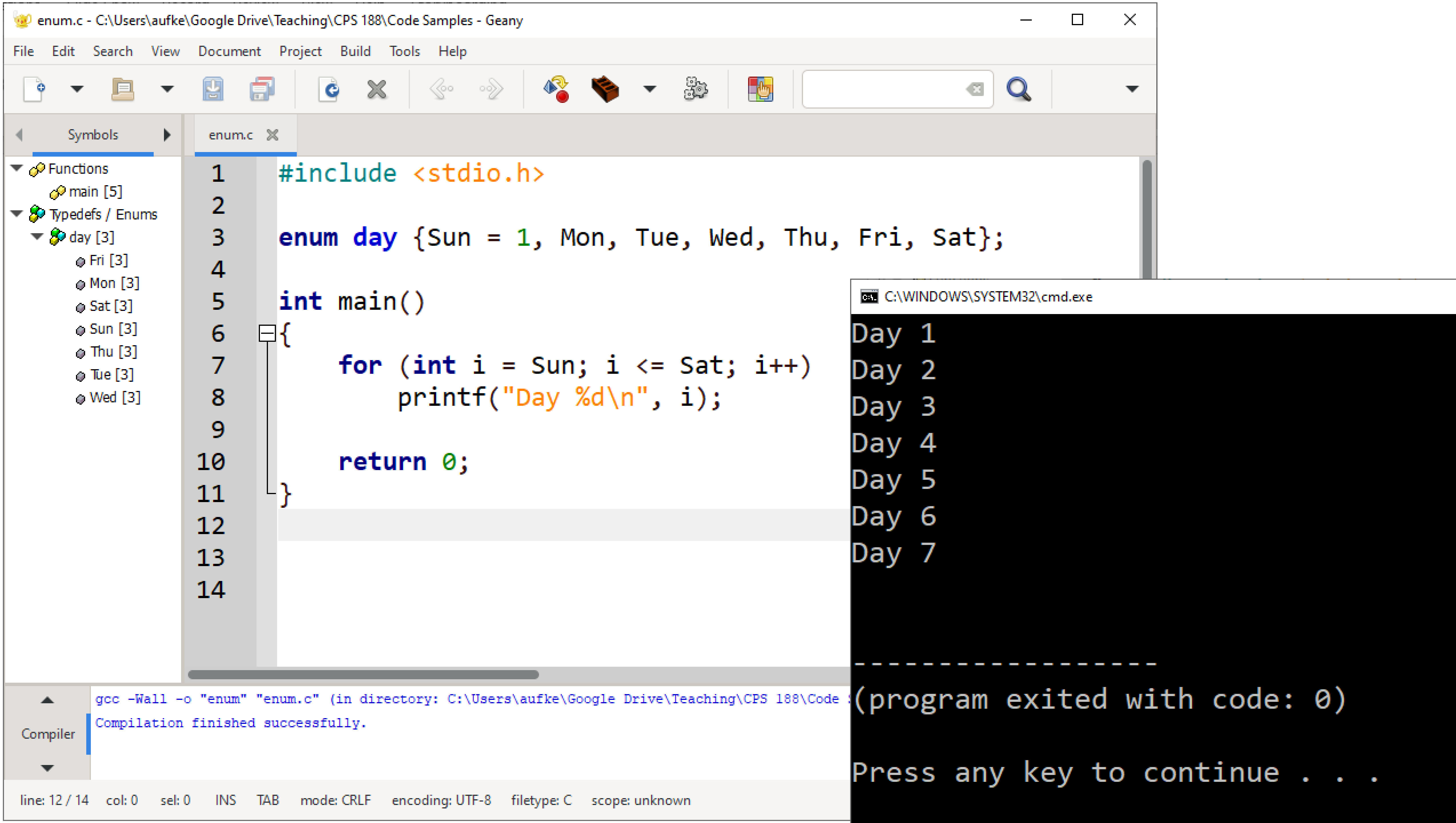
Enums

If we specify an integer for the first element, counting will start from there:

```
enum day {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};
```

```
for (int i = Sun; i <= Sat; i++)
   printf(“Day %d\n”, i);
```

What prints? `Day 1, Day 2, …, Day 6, Day 7`

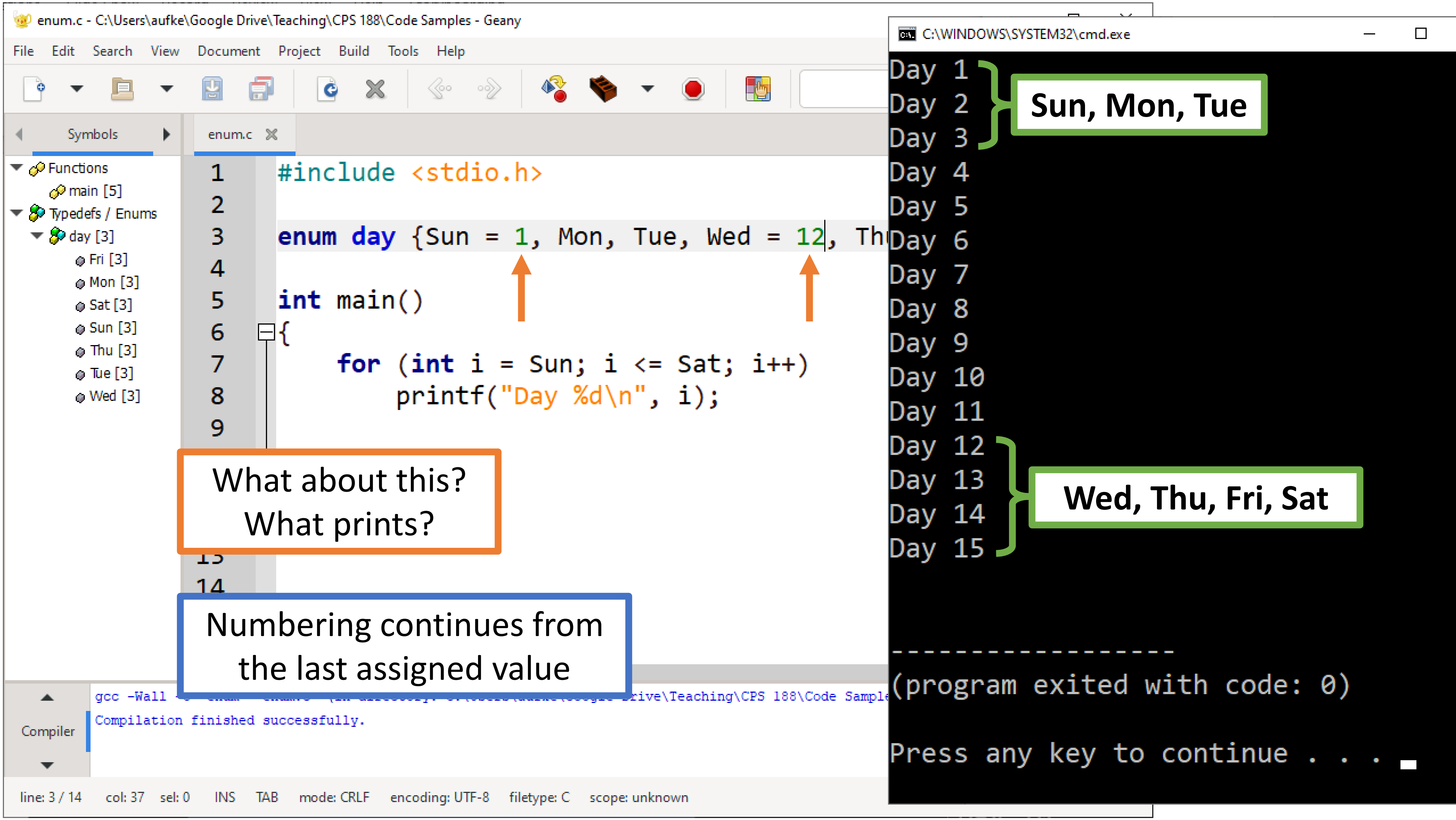


```c
#include <stdio.h>

enum day {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);

    return 0;
}
```

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code
Compilation finished successfully.
```

line: 12 / 14   col: 0   sel: 0   INS   TAB   mode: CRLF   encoding: UTF-8   filetype: C   scope: unknown

```
Day 1
Day 2
Day 3
Day 4
Day 5
Day 6
Day 7


------------------
(program exited with code: 0)

Press any key to continue . . .
```



enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany
File Edit Search View Document Project Build Tools Help
Symbols
enum.c
Functions
main [5]
Typedefs / Enums
day [3]
Fri [3]
Mon [3]
Sat [3]
Sun [3]
Thu [3]
Tue [3]
Wed [3]
Compiler
C:\WINDOWS\SYSTEM32\cmd.exe

```c
#include <stdio.h>

enum day {Sun = 1, Mon, Tue, Wed = 12, Th

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);
```

What about this? What prints?

Numbering continues from the last assigned value

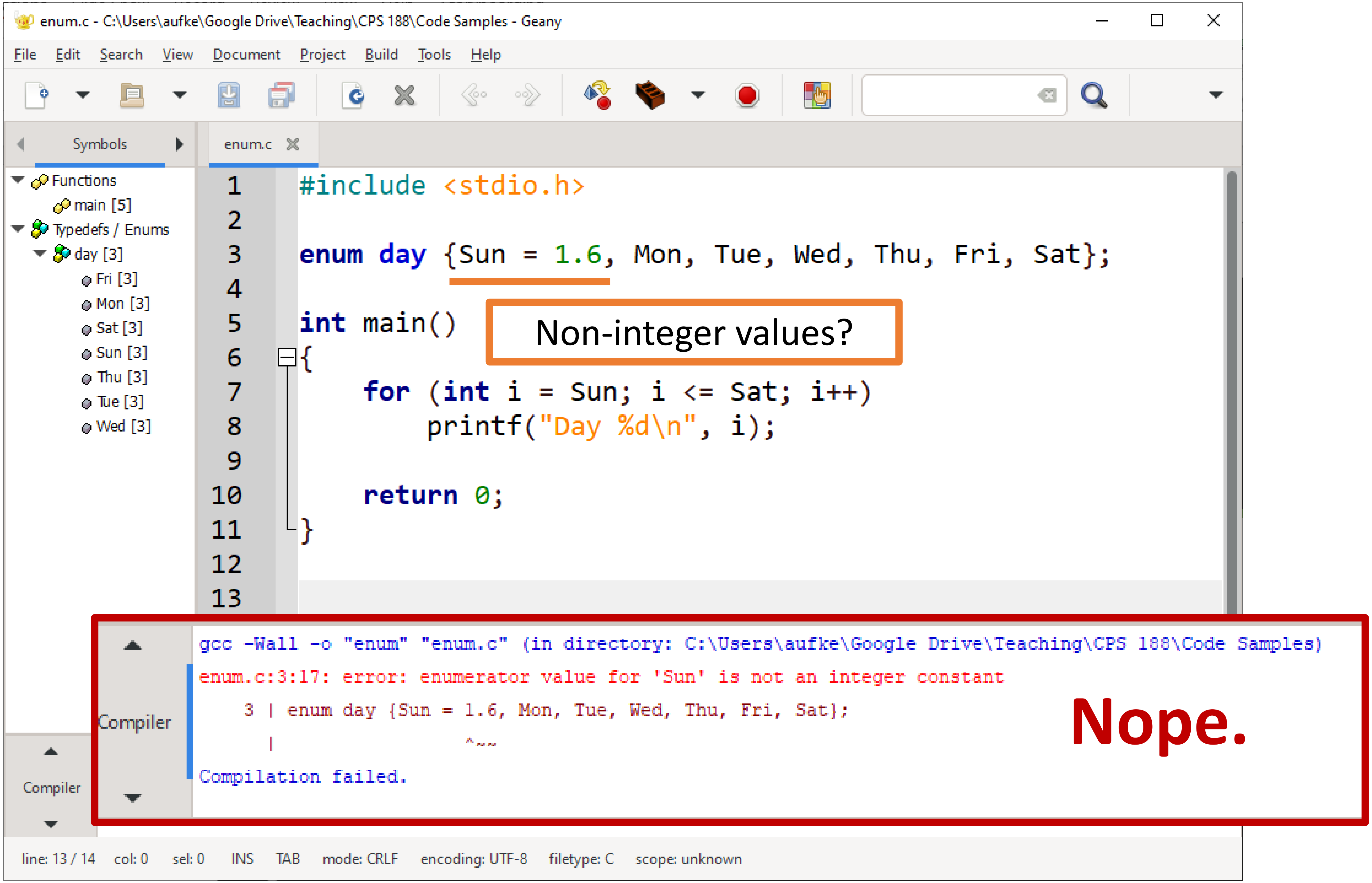
```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Sample
Compilation finished successfully.
```

```
line: 3 / 14   col: 37   sel: 0   INS   TAB   mode: CRLF   encoding: UTF-8   filetype: C   scope: unknown
```

```
Day 1
Day 2
Day 3
Day 4
Day 5
Day 6
Day 7
Day 8
Day 9
Day 10
Day 11
Day 12
Day 13
Day 14
Day 15


--------------------
(program exited with code: 0)

Press any key to continue . . .
```

Sun, Mon, Tue

Wed, Thu, Fri, Sat

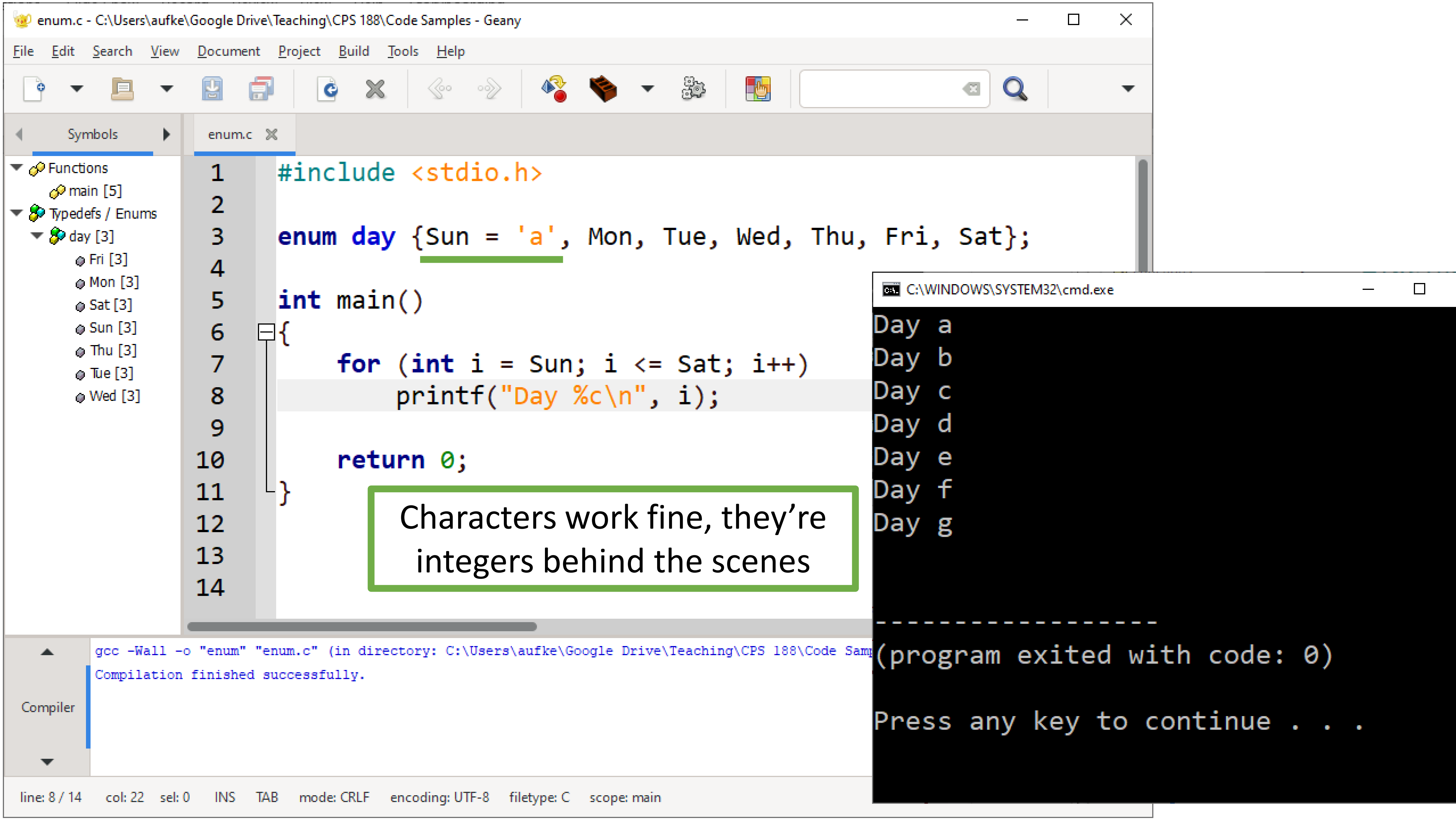

enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany
File Edit Search View Document Project Build Tools Help
Symbols
enum.c
Functions
main [5]
Typedefs / Enums
day [3]
Fri [3]
Mon [3]
Sat [3]
Sun [3]
Thu [3]
Tue [3]
Wed [3]
Compiler
Compiler
line: 13 / 14 col: 0 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: unknown

```c
#include <stdio.h>

enum day {Sun = 1.6, Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);

    return 0;
}
```

Non-integer values?

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples)
enum.c:3:17: error: enumerator value for 'Sun' is not an integer constant
    3 | enum day {Sun = 1.6, Mon, Tue, Wed, Thu, Fri, Sat};
      |                 ^~~
Compilation failed.
```

Nope.


enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany
File Edit Search View Document Project Build Tools Help
Symbols
enum.c
Functions
main [5]
Typedefs / Enums
day [3]
Fri [3]
Mon [3]
Sat [3]
Sun [3]
Thu [3]
Tue [3]
Wed [3]
Compiler
C:\WINDOWS\SYSTEM32\cmd.exe

```c
#include <stdio.h>

enum day {Sun = 'a', Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %c\n", i);

    return 0;
}
```

Characters work fine, they're integers behind the scenes

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samp
Compilation finished successfully.
```

line: 8 / 14   col: 22   sel: 0   INS   TAB   mode: CRLF   encoding: UTF-8   filetype: C   scope: main

```
Day a
Day b
Day c
Day d
Day e
Day f
Day g


------------------
(program exited with code: 0)

Press any key to continue . . .
```

Unions

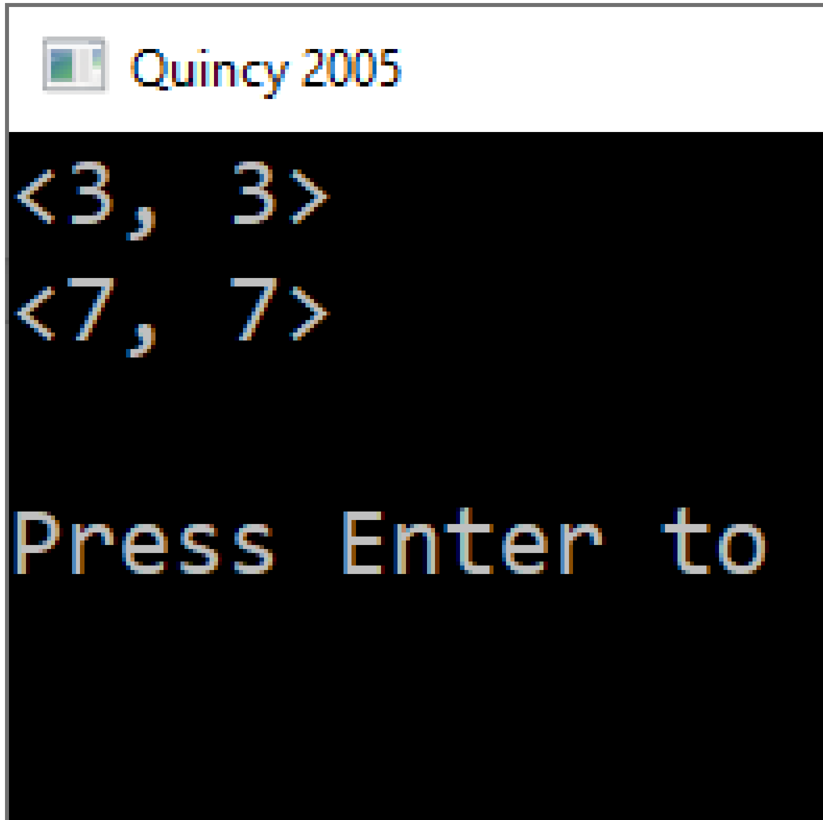
Store multiple variables at the same memory location:

```
union point {
   int x, y;
};
```

- x and y are two names for the same memory location.
- Changing one will affect the other.

Unions

```c
#include <stdio.h>

typedef union point { int x, y; } point;

int main()
{
        point pt;
        pt.x = 3;
        printf("<%d, %d>\n", pt.x, pt.y);
        pt.y = 7;
        printf("<%d, %d>\n", pt.x, pt.y);
        return 0;
}
```

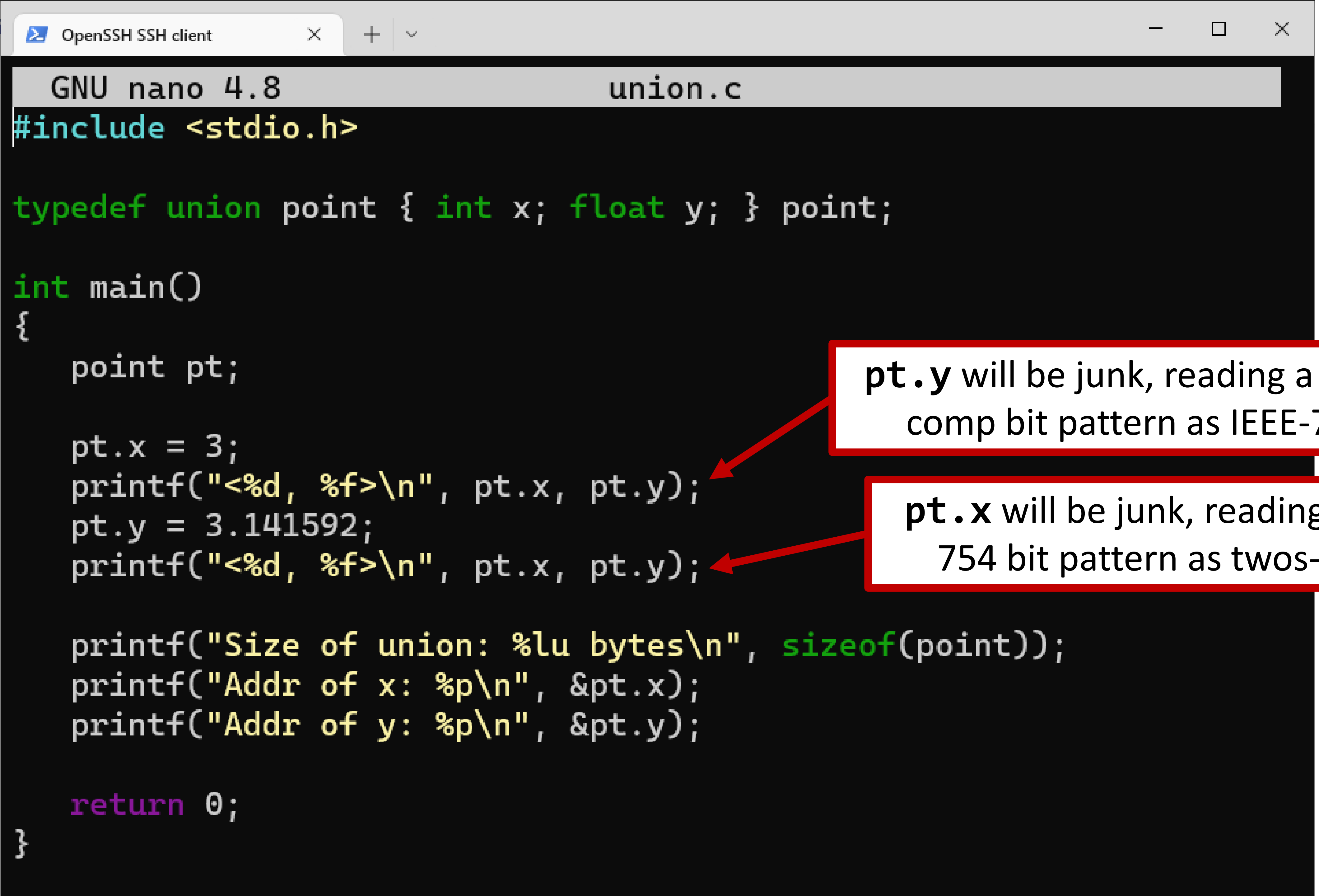


```
<3, 3>
<7, 7>

Press Enter to
```

Hmmm…

```
typedef union point {
   int x;
   float y;
} point;
```

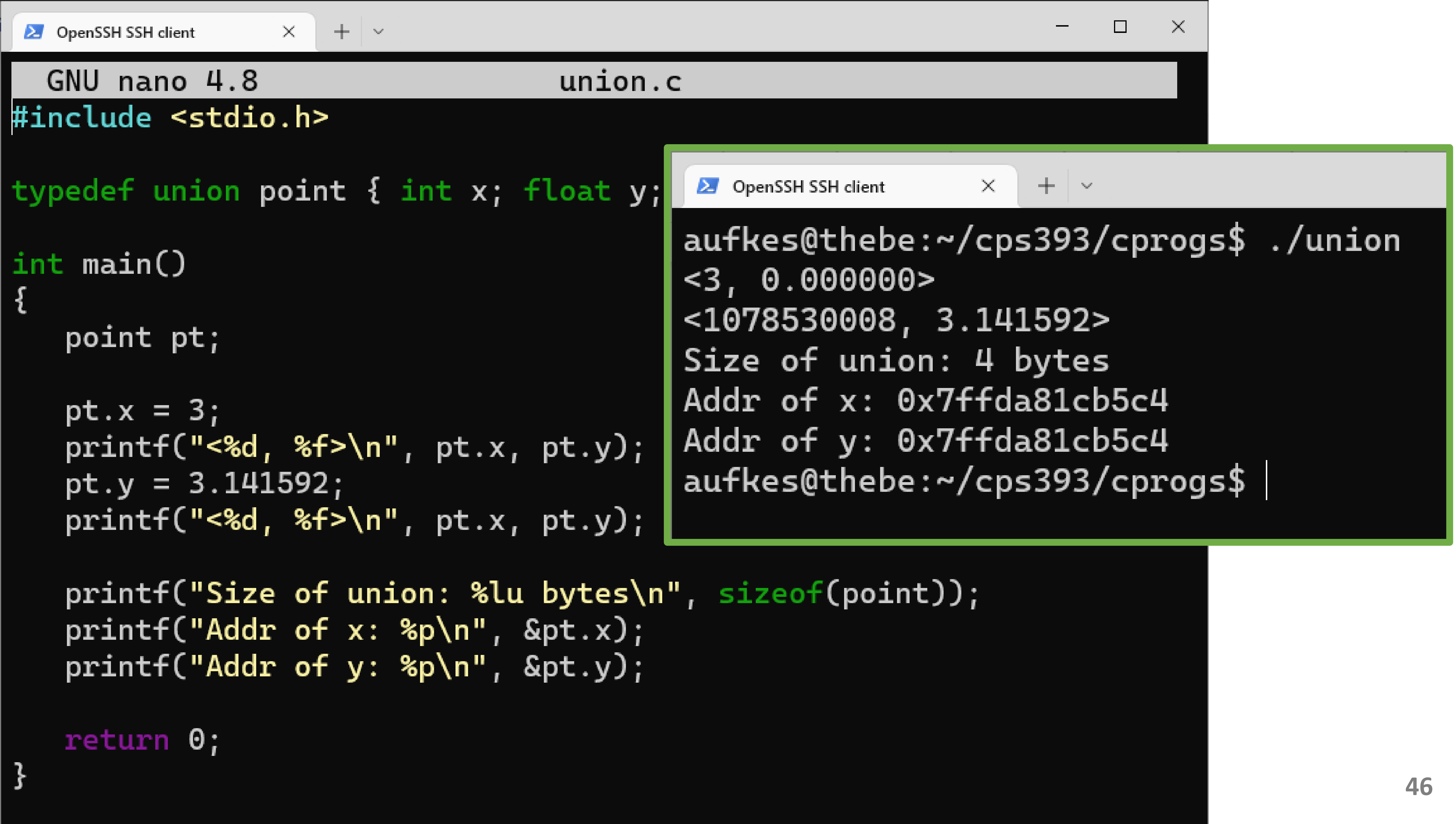
- Can overlap different types!
- If I store a float in y, I should not try and read x
- If I store an int in x, I should not read y.



```
  GNU nano 4.8                    union.c
#include <stdio.h>

typedef union point { int x; float y; } point;

int main()
{
   point pt;

   pt.x = 3;
   printf("<%d, %f>\n", pt.x, pt.y);
   pt.y = 3.141592;
   printf("<%d, %f>\n", pt.x, pt.y);

   printf("Size of union: %lu bytes\n", sizeof(point));
   printf("Addr of x: %p\n", &pt.x);
   printf("Addr of y: %p\n", &pt.y);

   return 0;
}
```

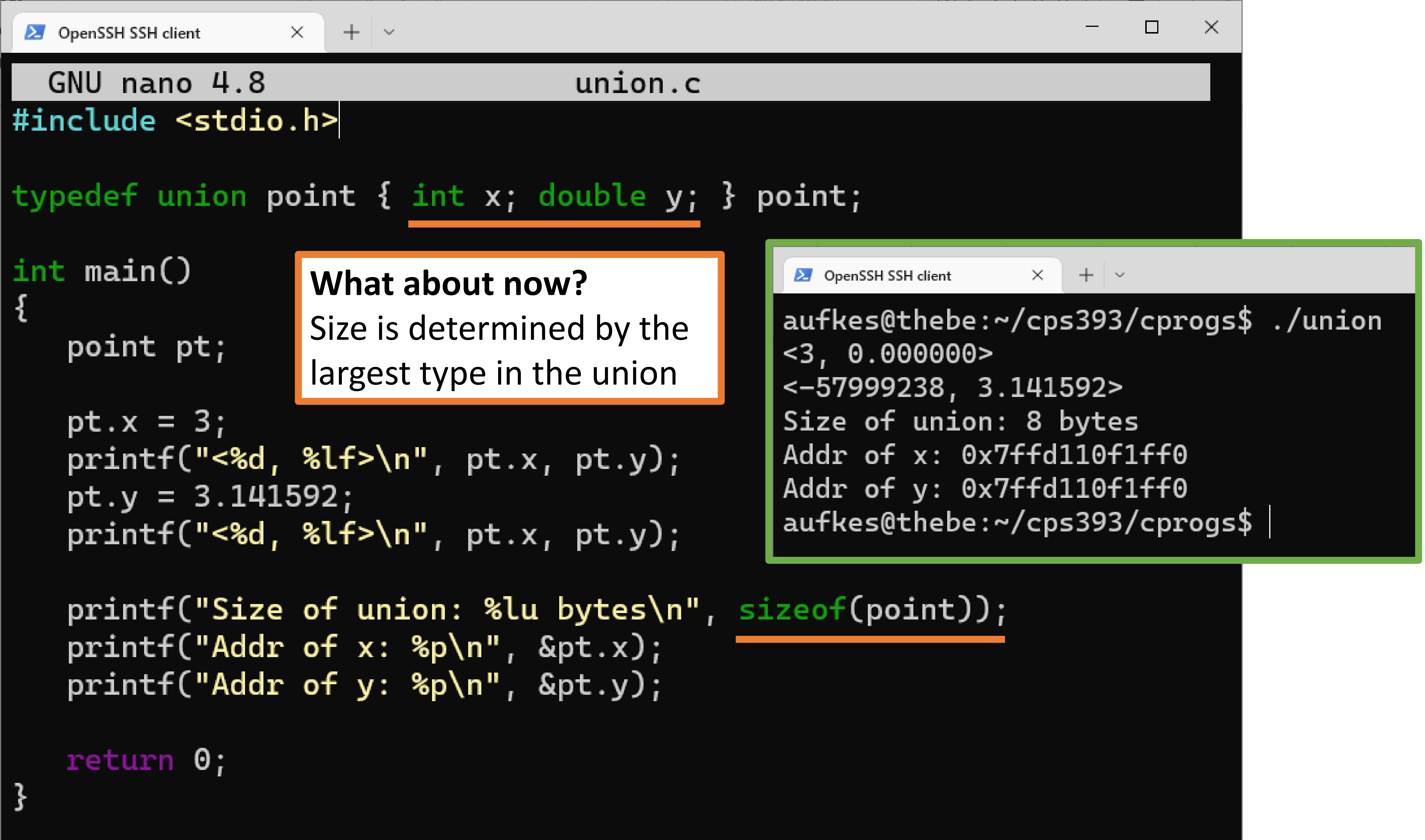
`pt.y` will be junk, reading a twos-comp bit pattern as IEEE-754

`pt.x` will be junk, reading a IEEE-754 bit pattern as twos-comp



```c
#include <stdio.h>

typedef union point { int x; float y;

int main()
{
   point pt;

   pt.x = 3;
   printf("<%d, %f>\n", pt.x, pt.y);
   pt.y = 3.141592;
   printf("<%d, %f>\n", pt.x, pt.y);

   printf("Size of union: %lu bytes\n", sizeof(point));
   printf("Addr of x: %p\n", &pt.x);
   printf("Addr of y: %p\n", &pt.y);

   return 0;
}
```

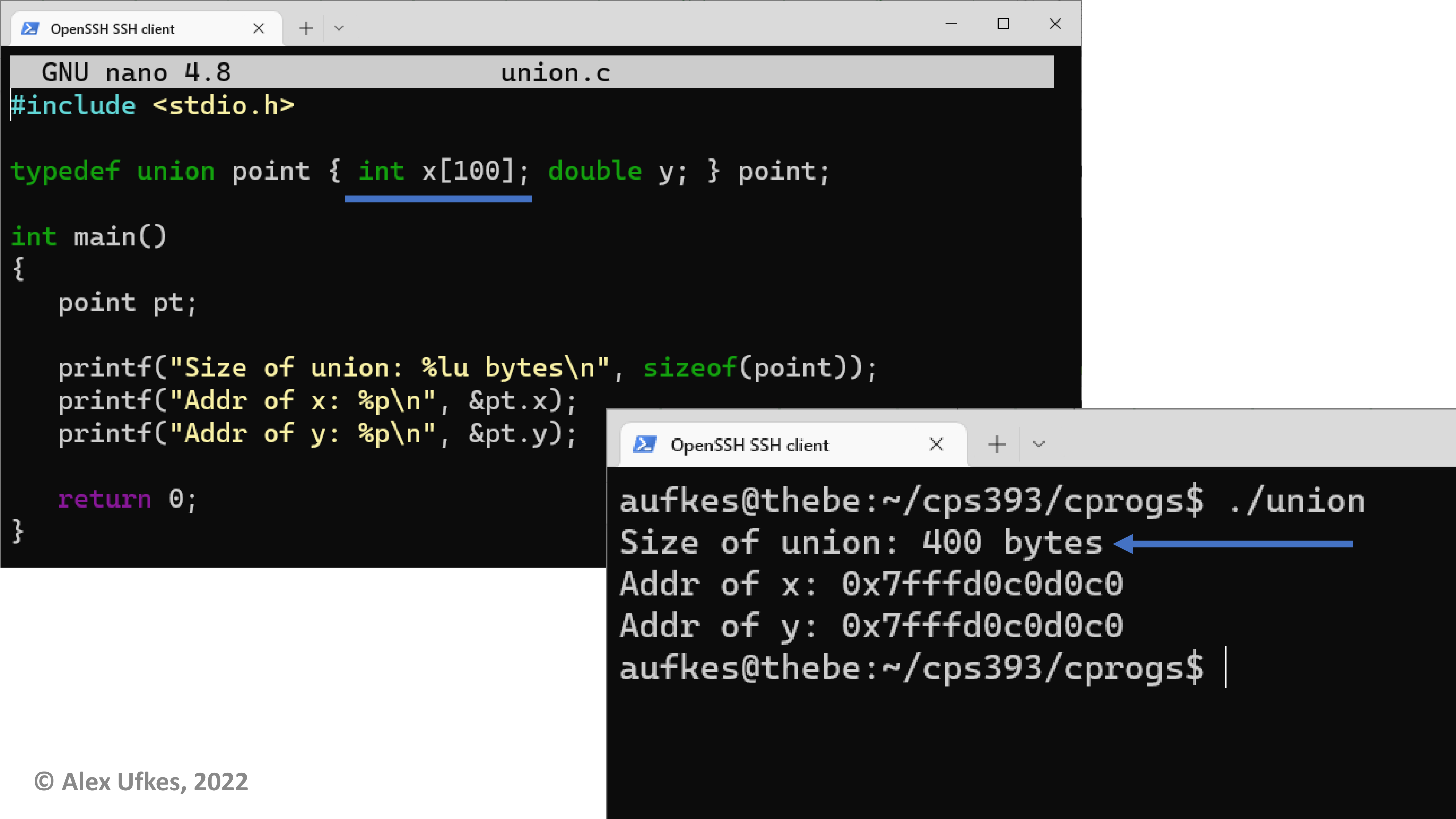
```
aufkes@thebe:~/cps393/cprogs$ ./union
<3, 0.000000>
<1078530008, 3.141592>
Size of union: 4 bytes
Addr of x: 0x7ffda81cb5c4
Addr of y: 0x7ffda81cb5c4
aufkes@thebe:~/cps393/cprogs$
```



```
  GNU nano 4.8                    union.c
#include <stdio.h>

typedef union point { int x; double y; } point;

int main()
{
   point pt;

   pt.x = 3;
   printf("<%d, %lf>\n", pt.x, pt.y);
   pt.y = 3.141592;
   printf("<%d, %lf>\n", pt.x, pt.y);

   printf("Size of union: %lu bytes\n", sizeof(point));
   printf("Addr of x: %p\n", &pt.x);
   printf("Addr of y: %p\n", &pt.y);

   return 0;
}
```

What about now?
Size is determined by the largest type in the union

```
aufkes@thebe:~/cps393/cprogs$ ./union
<3, 0.000000>
<-57999238, 3.141592>
Size of union: 8 bytes
Addr of x: 0x7ffd110f1ff0
Addr of y: 0x7ffd110f1ff0
aufkes@thebe:~/cps393/cprogs$
```



```
  GNU nano 4.8                        union.c
#include <stdio.h>

typedef union point { int x[100]; double y; } point;

int main()
{
   point pt;

   printf("Size of union: %lu bytes\n", sizeof(point));
   printf("Addr of x: %p\n", &pt.x);
   printf("Addr of y: %p\n", &pt.y);

   return 0;
}
```

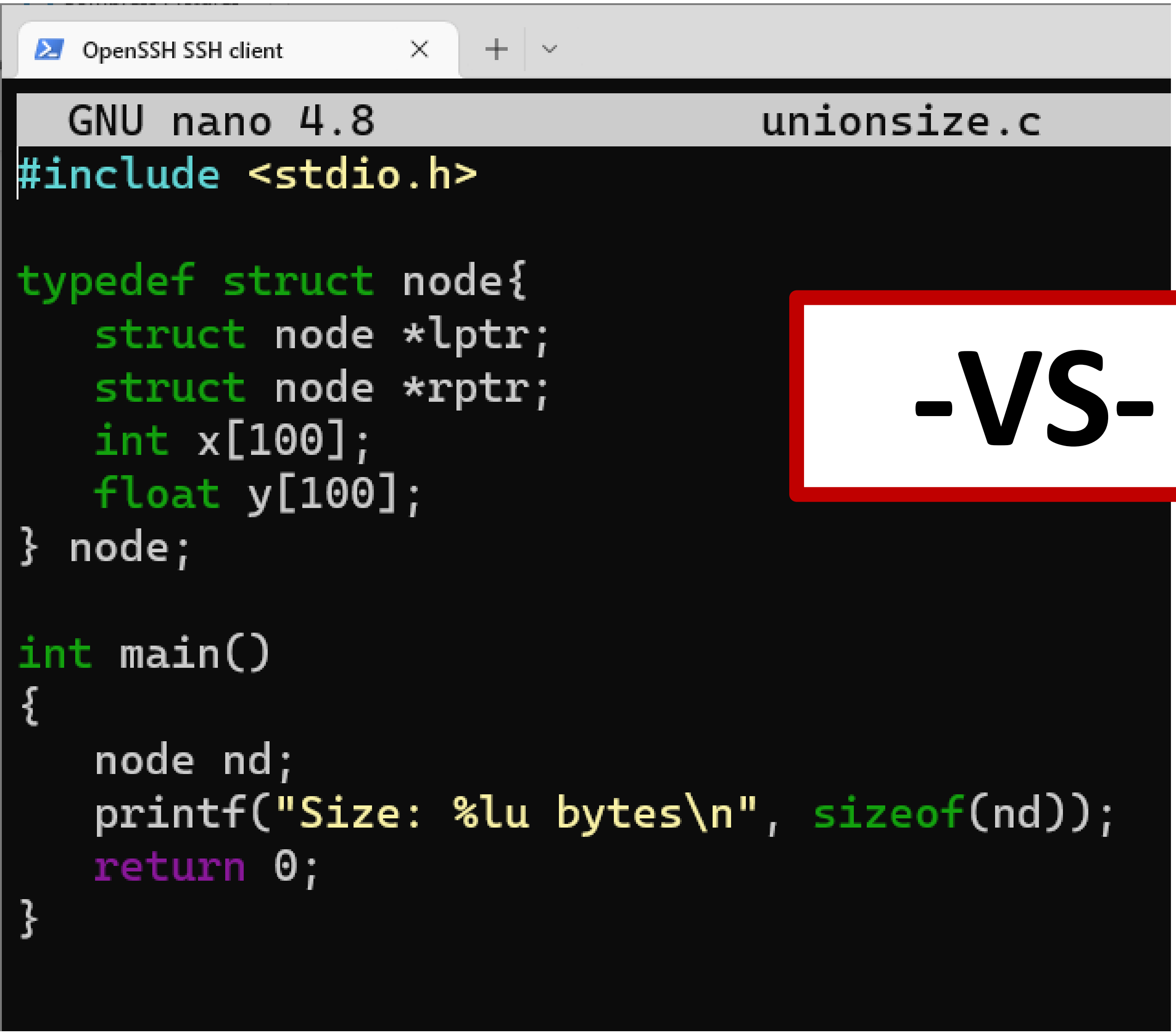
```
aufkes@thebe:~/cps393/cprogs$ ./union
Size of union: 400 bytes
Addr of x: 0x7fffd0c0d0c0
Addr of y: 0x7fffd0c0d0c0
aufkes@thebe:~/cps393/cprogs$
```

What’s the Point?

Saving space in memory:

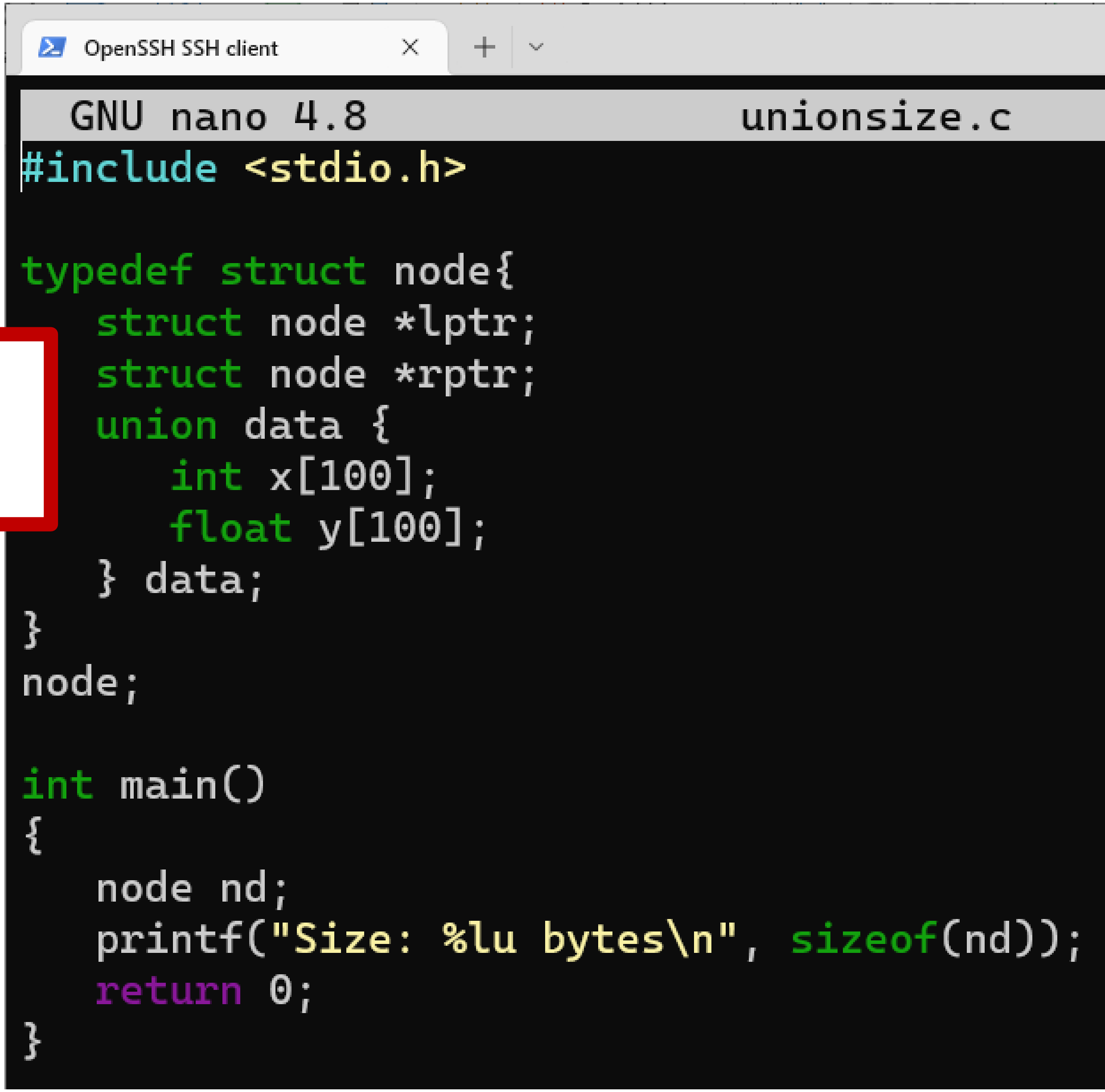
Consider a binary tree implemented using a linked data structure:

- Interior nodes store **floating point** data, leaf nodes store **integer** data.
- Instead of every node having float *and* int variables, we can just have a union with a float and int.
- Cuts the size of each node’s data in half.
- We don’t know about binary trees yet, but we’ll see them in our last week

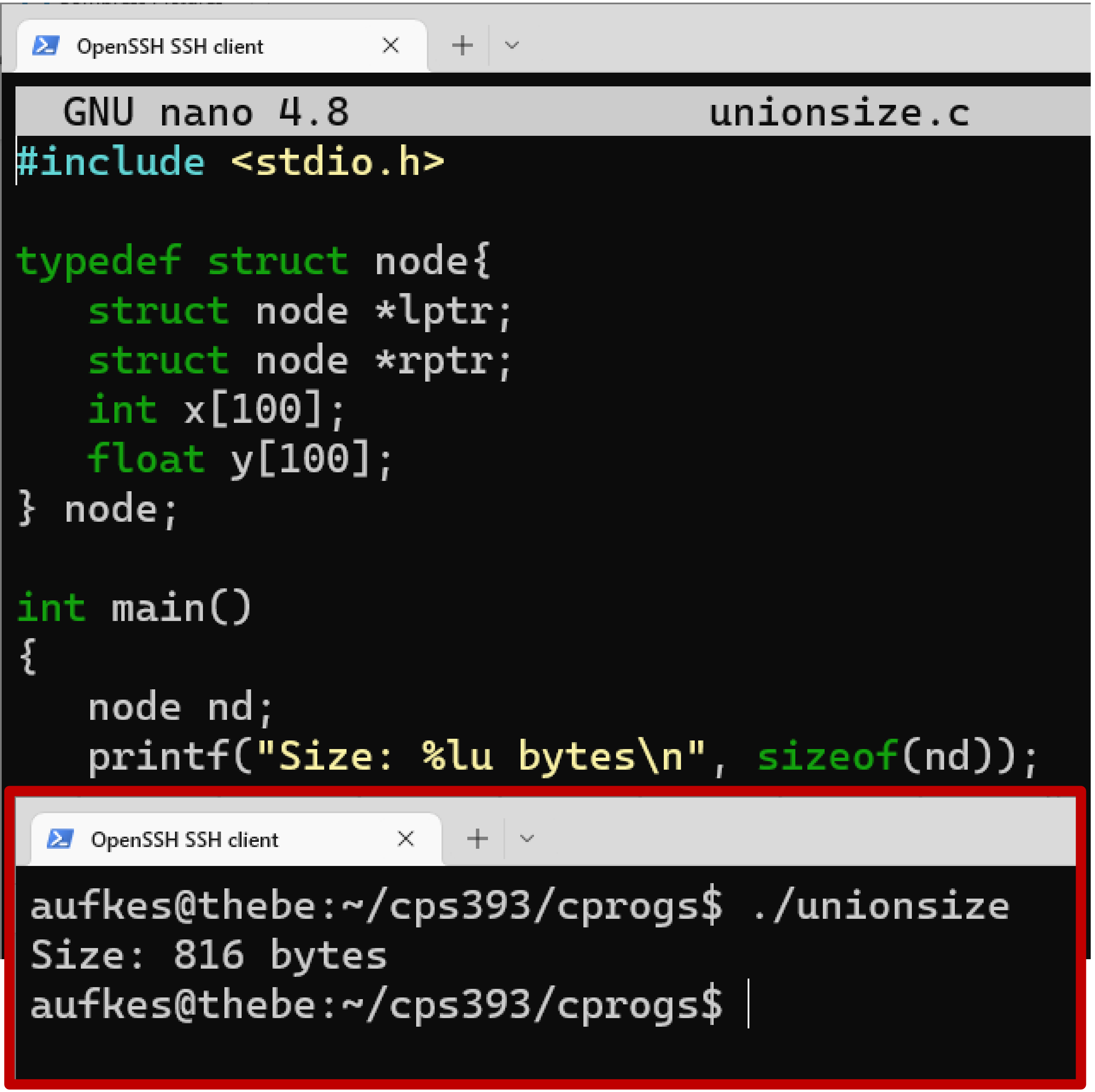


```
  GNU nano 4.8                    unionsize.c
#include <stdio.h>

typedef struct node{
   struct node *lptr;
   struct node *rptr;
   int x[100];
   float y[100];
} node;

int main()
{
   node nd;
   printf("Size: %lu bytes\n", sizeof(nd));
   return 0;
}
```

-VS-

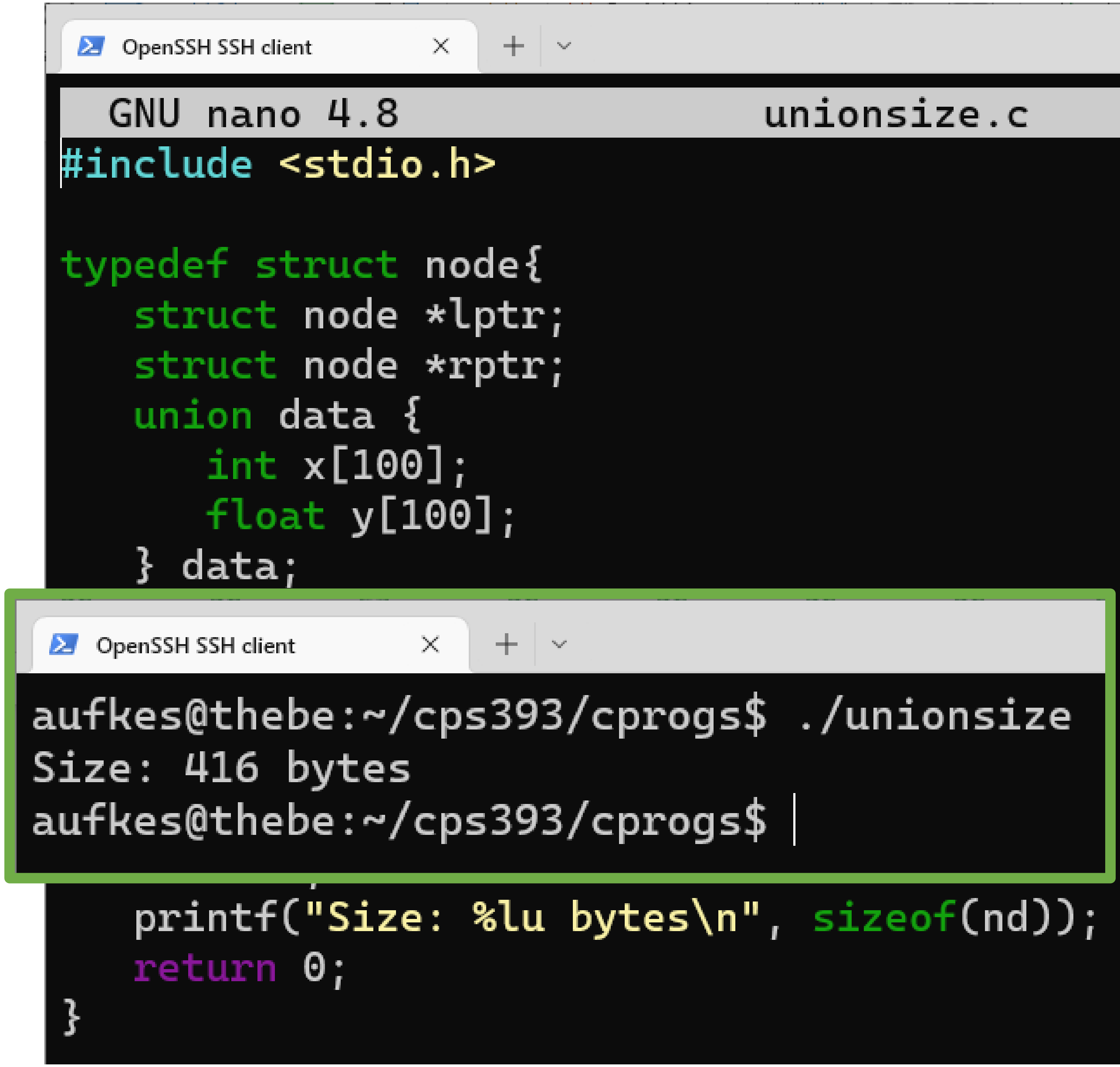


```
  GNU nano 4.8                    unionsize.c
#include <stdio.h>

typedef struct node{
   struct node *lptr;
   struct node *rptr;
   union data {
      int x[100];
      float y[100];
   } data;
}
node;

int main()
{
   node nd;
   printf("Size: %lu bytes\n", sizeof(nd));
   return 0;
}
```



```
  GNU nano 4.8                    unionsize.c
#include <stdio.h>

typedef struct node{
   struct node *lptr;
   struct node *rptr;
   int x[100];
   float y[100];
} node;

int main()
{
   node nd;
   printf("Size: %lu bytes\n", sizeof(nd));
```

```
aufkes@thebe:~/cps393/cprogs$ ./unionsize
Size: 816 bytes
aufkes@thebe:~/cps393/cprogs$
```



```
  GNU nano 4.8                    unionsize.c
#include <stdio.h>

typedef struct node{
   struct node *lptr;
   struct node *rptr;
   union data {
      int x[100];
      float y[100];
   } data;
```

```
aufkes@thebe:~/cps393/cprogs$ ./unionsize
Size: 416 bytes
aufkes@thebe:~/cps393/cprogs$
```

```
   printf("Size: %lu bytes\n", sizeof(nd));
   return 0;
}
```

Questions?