Initial code setup

$ cd eecs211
$ wget $URL211/lec/02types_values.tgz
...
$ tar zxf 02types_values.tgz
$ cd 02types_values
Introduction to int and double
Defining a variable

Every variable in C must be defined with a type:

```c
int x = 5;
double f = 5.1;
```

What does this do?
Defining a variable

Every variable in C must be defined with a type:

```c
int x = 5;
double f = 5.1;
```

What does this do?

A variable names an object of the given type, which is a chunk of memory that can hold a value of that type:

```
x: +0x00000005
f: +5.099999999999999964...e+00
```

(The notation $AeB$ means $A \times 10^B$)
Let’s observe this in C!

```c
#include <stdio.h>

int main()
{
    int x = 5;
    double f = 5.1;

    printf("x: %d\n", x);
    printf("f: %.60e\n", f);

    printf("sizeof x: %zu bytes\n", sizeof x);
    printf("sizeof f: %zu bytes\n", sizeof f);
}
```
Output from the previous slide

```bash
$ make build/types
c -o build/types src/types.c -std=c11 -pedantic -W...

build/types

x: 5
f: 5.0999999999999996447286321199499070644378662109...

sizeof x: 4 bytes
sizeof f: 8 bytes
```
Output from the previous slide

$ make build/types

c -o build/types src/types.c -std=c11 -pedantic -

build/types

x: 5
f: 5.0999999999999996447286321199499070644378662109…

sizeof x: 4 bytes
sizeof f: 8 bytes
Output from the previous slide

$ make build/types
cc -o build/types src/types.c -std=c11 -pedantic -W...
$

$
Output from the previous slide

$ make build/types
cc -o build/types src/types.c -std=c11 -pedantic -W...
$ build/types
Output from the previous slide

```
$ make build/types
cc -o build/types src/types.c -std=c11 -pedantic -W...
$ build/types
x: 5
f: 5.0999999999999996447286321199499070644378662109...
sizeof x: 4 bytes
sizeof f: 8 bytes
```
Including headers

This is a directive that causes the functions defined in `stdio.h` to be known to the compiler:

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

(Without it, we wouldn’t have access to `printf`.)
The main function

C programs can have multiple functions, but they always start by calling main:

```c
int main()
{
    // ...
}
```

(The `int` is main’s return type. C programs return an *error code* to the OS, where 0 means success and non-zero means failure. The main function magically returns 0 for you if you don’t tell it otherwise.)
Producing output

The usual way to print in C is the `printf` function, which takes a format string followed by arguments to interpolate in place of the format string’s directives:

```c
printf("x: %d\n", x);
```

(Prints format string “x: %d\n”, replacing directive %d with the value of x.)
Producing output

The usual way to print in C is the `printf` function, which takes a *format string* followed by arguments to *interpolate* in place of the format string’s *directives*:

```
printf("x:␣%d\n", x);
```

(Prints format string “x: %d\n”, replacing directive %d with the value of x.)

Each directive specifies the type of the argument to print, possibly with some options:

- `%d` expects an *int*
- `%.60e` expects a *double*; includes 60 digits of precision
- `%zu` expects a *size_t* (the result of `sizeof`)
Reading input

To input numbers in C, use the `scanf` function.
Reading input

To input numbers in C, use the `scanf` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```c
int x = 0;
scanf("%d", &x);
```
Reading input

To input numbers in C, use the `scanf` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```c
int x = 0;
scanf("%d", &x);
```

- Like `printf`, `scanf` uses a format string to determine what type to convert the input to.
- But `scanf`'s directives are not all the same as `printf`'s! (Use `%lf` to read a `double`.)
- An argument `x` would pass the value of variable `x` to `scanf`, but `&x` means to pass `x`'s location.
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;

    printf("Enter two integers: ");
    scanf("%d%d", &x, &y);
    printf("%d * %d == %d\n", x, y, x * y);
}

Output from the previous slide

$make build/input$

cc -o build/input src/input.c -std=c11 -pedantic -W...

build/input

Enter two integers:

5 7

5 * 7 == 35

build/input

Enter two integers:

5 seven

5 * 0 == 0

build/input

Enter two integers:

five 7

0 * 0 == 0

build/input

Enter two integers:

^D

0 * 0 == 0

Output from the previous slide

$ make build/input

Enter two integers:
5 7
5 * 7 == 35

Enter two integers:
5 seven
5 * 0 == 0

Enter two integers:
five 7
0 * 0 == 0

Enter two integers:
^D
0 * 0 == 0
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$
$
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Output from the previous slide

$ make build/input
cce -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers:
Output from the previous slide

```
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
```

12
Output from the previous slide

```
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
```


Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...

$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers:
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...

$ build/input
Enter two integers: 5 7
5 * 7 == 35

$ build/input
Enter two integers: 5 seven
5 * 0 == 0

$ build/input
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers:
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers: five 7
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers: five 7
0 * 0 == 0
$
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers: five 7
0 * 0 == 0
$ build/input
Output from the previous slide

```
$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers: five 7
0 * 0 == 0
$ build/input
Enter two integers:
```
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...

$ build/input
Enter two integers: 5 7
5 * 7 == 35

$ build/input
Enter two integers: 5 seven
5 * 0 == 0

$ build/input
Enter two integers: five 7
0 * 0 == 0

$ build/input
Enter two integers: ^D
Output from the previous slide

$ make build/input
cc -o build/input src/input.c -std=c11 -pedantic -W...
$ build/input
Enter two integers: 5 7
5 * 7 == 35
$ build/input
Enter two integers: 5 seven
5 * 0 == 0
$ build/input
Enter two integers: five 7
0 * 0 == 0
$ build/input
Enter two integers: ^D0 * 0 == 0
$
How `scanf` reports errors

`scanf` returns the number of successful conversions.
#include <stdio.h>

int main()
{
    int x, y;

    printf("Enter two integers: ");

    if (scanf("%d%d", &x, &y) != 2) {
        printf("Input error\n");
        return 1;
    }

    printf("%d * %d == %d\n", x, y, x * y);
}


Syntax for functions and arithmetic

```c
#include <stdio.h>

unsigned long factorial(unsigned long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main()
{
    unsigned long n = 0;
    scanf("%lu", &n);
    printf("%lu! == %lu\n", n, factorial(n));
}
```
Facts from the previous slide

- **long** is an integral type that might have more bits than **int** (like maybe 64 instead of 32)
- **unsigned** means it does not include negative numbers (which means it includes twice as many positive numbers instead)
- * multiplies, – subtracts, and == compares for equality
- The result of a function must be given in a **return** statement
- The **printf** and **scanf** directive for **unsigned long** is %lu
Something funny about int

Not every mathematical integer can fit in a C int.
Something funny about int

Not every mathematical integer can fit in a C int.

- An int is stored in a finite number of bits (like 16 or 32 or 64)
Something funny about *int*

Not every mathematical integer can fit in a C *int*.

- An *int* is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
Something funny about int

Not every mathematical integer can fit in a C int.

- An int is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
- For example, 32-bit ints (usually) range from \(-2^{31}\) to \(2^{31} - 1\) (inclusive)
Something funny about int

Not every mathematical integer can fit in a C int.

- An int is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
- For example, 32-bit ints (usually) range from \(-2^{31}\) to \(2^{31} - 1\) (inclusive)
- The actual values are defined in limits.h as INT_MIN and INT_MAX
Something funny about int

Not every mathematical integer can fit in a C int.

• An int is stored in a finite number of bits (like 16 or 32 or 64)
• This means that it has a finite range
• For example, 32-bit ints (usually) range from $-2^{31}$ to $2^{31} - 1$ (inclusive)
• The actual values are defined in limits.h as INT_MIN and INT_MAX
• An int operation whose mathematical result is out of range produces UNDEFINED BEHAVIOR
WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error…
It’s like a kind of error…

But the computer doesn’t necessarily notice…
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error…
But the computer doesn’t necessarily notice…
Your program might just keep running and produce nonsense!
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error…
But the computer doesn’t necessarily notice…
Your program might just keep running and produce nonsense!
Technically, a program with UB has no meaning. It’s allowed to do anything:
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error…
But the computer doesn’t necessarily notice…
Your program might just keep running and produce nonsense!
Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error...

But the computer doesn’t necessarily notice...

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
- Keep going
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error…
But the computer doesn’t necessarily notice…
Your program might just keep running and produce nonsense!
Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
- Keep going
- Reformat your hard disk
WTF IS UNDEFINED BEHAVIOR?!?!?

It’s like a kind of error…

But the computer doesn’t necessarily notice…

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
- Keep going
- Reformat your hard disk
- Launch the missiles
Examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer “overflow”
Examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer “overflow”

Example of all three:

```c
int x, y;
scanf("%d%d", &x, &y);
printf("%d\n", x / y);
```
Examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer “overflow”

Example of all three:

```c
int x, y;
scanf("%d%d", &x, &y);
printf("%d\n", x / y);
```

Fix for all three:

```c
int x, y;
if (scanf("%d%d", &x, &y) == 2 &&
    y != 0 &&
    !(x == INT_MIN && y == -1))
    printf("%d\n", x / y);
```
UB is really weird

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

void check_int(int z)
{
    if (z < z + 1)
        printf("math\n");
    else
        printf("C.S.\n");
}

int main()
{
    check_int(0);
    check_int(INT_MAX);
}
```
The results depend on the optimization level

$\$

(This is very, very bad.)
The results depend on the optimization level

$ make build/int_max

(This is very, very bad.)
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedanti...
$

(This is very, very bad.)
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedantic...
$ build/int_max

(This is very, very bad.)
The results depend on the optimization level

```
$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedanti...
$ build/int_max
math
C.S.
$ 
```
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedantic...
$ build/int_max
math
C.S.
$ make build/int_max.opt
The results depend on the optimization level

```
$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedantic...
$ build/int_max
math
C.S.
$ make build/int_max.opt
cc -O2 -o build/int_max.opt src/int_max.c -std=c11 ...
$
```
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedantic...
$ build/int_max
math
C.S.
$ make build/int_max.opt
cc -O2 -o build/int_max.opt src/int_max.c -std=c11 ...
$ build/int_max.opt

(This is very, very bad.)
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedanti...
$ build/int_max
math
C.S.
$ make build/int_max.opt
cc -O2 -o build/int_max.opt src/int_max.c -std=c11 ...
$ build/int_max.opt
math
math
$

(This is very, very bad.)
The results depend on the optimization level

$ make build/int_max
cc -o build/int_max src/int_max.c -std=c11 -pedantic...
$ build/int_max
math
C.S.
$ make build/int_max.opt
cc -O2 -o build/int_max.opt src/int_max.c -std=c11 ...
$ build/int_max.opt
math
math
math
$

(This is very, very bad.)
Structure types
Structure types in C

C (like BSL/ISL) uses structures to define new data types by composition of existing data types.

A structure type has a name and some number of fields, each of which must be declared with a type.
Syntax to define a struct type

```c
struct posn
{
    double x;
    double y;
};

struct circle
{
    struct posn center;
    double radius;
};
```

Note that the type defined by the `struct posn` definition, and used for field `center` of `struct circle` is `struct posn`, not merely `posn`. (In C++ you could refer to it either way, but not in C.)
Syntax to define a struct type

```c
struct posn
{
    double x;
    double y;
};

struct circle
{
    struct posn center;
    double radius;
};
```

Note that the type defined by the `struct posn` definition, and used for field `center` of `struct circle` is `struct posn`, not merely `posn`. (In C++ you could refer to it either way, but not in C.)
Suppose we have a variable \( p \) whose type is `struct posn`. How do we access \( p \)'s fields?

```c
#include <math.h>

// Finds the Manhattan distance between two points.
double manhattan_dist(struct posn p, struct posn q)
{
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
```
Syntax to use a structure

Suppose we have a variable \( p \) whose type is `struct posn`. How do we access \( p \)'s fields? \( p.x \) and \( p.y \)

```c
// Finds the Manhattan distance between two points.
double manhattan_dist(struct posn p, struct posn q) {
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
```
Syntax to use a structure

Suppose we have a variable `p` whose type is `struct posn`. How do we access `p`'s fields? `p.x` and `p.y`

Let's write a function to compute the Manhattan distance between two points. Mathematically,

\[ d_1((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2| \]

// For the fabs(double) function:
#include <math.h>

// Finds the Manhattan distance between two points.
double manhattan_dist(struct posn p, struct posn q)
{
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
Syntax to use a structure

Suppose we have a variable p whose type is `struct posn`. How do we access p’s fields? `p.x` and `p.y`.

Let’s write a function to compute the Manhattan distance between two points. Mathematically,

\[
d_1((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|
\]

// For the fabs(double) function:
#include <math.h>

// Finds the Manhattan distance between two points.
double manhattan_dist(struct posn p, struct posn q) {
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
Creating a structure

C offers *literal* syntax for most types:
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, -6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
<tr>
<td>char</td>
<td>'a', '␣', '0', 'n'</td>
</tr>
<tr>
<td>&quot;string&quot;</td>
<td>&quot;hello,␣world!&quot;</td>
</tr>
<tr>
<td>struct (struct posn)</td>
<td>{3.0, 4.0}</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td>3, −6, 0xBAADF00D</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, –6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, −6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
<tr>
<td>char</td>
<td>'a', '␣', '0', 'n'</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, –6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
<tr>
<td>char</td>
<td>'a', '␣', '0', 'n'</td>
</tr>
<tr>
<td>“string”</td>
<td>&quot;hello,␣world!&quot;</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, –6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
<tr>
<td>char</td>
<td>'a', '␣', '0', 'n'</td>
</tr>
<tr>
<td>“string”</td>
<td>&quot;hello,␣world!&quot;</td>
</tr>
<tr>
<td>struct</td>
<td>(struct posn) {3.0, 4.0}</td>
</tr>
</tbody>
</table>
Creating a structure

C offers *literal* syntax for most types:

<table>
<thead>
<tr>
<th>type</th>
<th>examples of literal syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>3, –6, 0xBAADF00D</td>
</tr>
<tr>
<td>double</td>
<td>3.5, 6.0221409e+23</td>
</tr>
<tr>
<td>char</td>
<td>'a', '␣', '0', 'n'</td>
</tr>
<tr>
<td>“string”</td>
<td>&quot;hello,␣world!&quot;</td>
</tr>
<tr>
<td>struct</td>
<td>(struct posn) {3.0, 4.0}</td>
</tr>
</tbody>
</table>

But this syntax for creating a `struct` is obscure! So the usual way of doing things is a bit more awkward…
Defining and initializing a structure

Usually to get a structure in C, first you define a structure variable and then initialize it by assigning each field:

```c
struct posn p;
p.x = 3.0;
p.y = 4.0;

struct circle c;
c.center.x = 7.0;
c.center.y = -9.2;
c.radius = 6.4;
```

C won’t force you to initialize all the fields, but guess what happens if you access a field that hasn’t been initialized?
Factory functions

If you get tired of initializing structures as on the previous slide, you can always define a factory function to do the work:

```c
struct circle
make_circle(struct posn center, double radius)
{
    struct circle result;
    result.center = center;
    result.radius = radius;
    return result;
}
```

(Note that functions can both take and return structure values.)
Visualizing structure value layout

```c
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```
Visualizing structure value layout

```c
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```

Diagram of the structure layout:

```
c: ________________________________
|          |          |
|          |          |
|          |          |
```

29
Visualizing structure value layout

```c
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```

c: 1.000000000e1

---

29
Visualizing structure value layout

```c
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```
Visualizing structure value layout

```c
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```

| c: | 1.0000000000e1 | -7.0000000000e0 | 5.0000000000e1 |
Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, widgets, etc. Example: 3 is an int value.
Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, widgets, etc. Example: 3 is an `int` value.
- An object is a chunk of memory that can hold a value. Example: if a function `f` has a declared parameter `int x`, then each time `f` is invoked, a fresh object that can hold an `int` value is created for it.
Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, widgets, etc. Example: 3 is an `int` value.
- An object is a chunk of memory that can hold a value. Example: if a function `f` has a declared parameter `int x`, then each time `f` is invoked, a fresh object that can hold an `int` value is created for it.
- A variable is the name of an object, such as `x` from the previous bullet point.
Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, widgets, etc. Example: 3 is an \texttt{int} value.
- An object is a chunk of memory that can hold a value. Example: if a function \( f \) has a declared parameter \texttt{int} \( x \), then each time \( f \) is invoked, a fresh object that can hold an \texttt{int} value is created for it.
- A variable is the name of an object, such as \( x \) from the previous bullet point.

Assigning a variable changes the value stored in the object that is named by the variable.
Example of definition and assignment

```c
int z = 5;
z = 7;
z = z + 4;
```

What happens?
Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

What happens?

The first statement is a definition, `int z = 5`. It creates an `int` object, names it `z`, and initializes it to the value 5.
Example of definition and assignment

```java
int z = 5;
z = 7;
z = z + 4;
```

What happens?

The first statement is a definition, `int z = 5`. It creates an `int` object, names it `z`, and initializes it to the value 5.

The second statement is an assignment, `z = 7;`. It replaces the value 5 stored in the object named by `z` with the value 7.
Example of definition and assignment

int z = 5;
z = 7;
z = z + 4;

What happens? 

The first statement is a definition, int z = 5. It creates an int object, names it z, and initializes it to the value 5.

The second statement is an assignment, z = 7; . It replaces the value 5 stored in the object named by z with the value 7.

The third statement is also an assignment, z = z + 4; . It first retrieves the current value of z (7), then adds 4 to it, and then stores the result (11) back in the object named by z.
The key point: Indirection

A variable in C does not stand directly for a value.
A variable in C refers to a value *indirectly*, by naming an object that *contains* a value.
How to increment a variable

Simple:

\[ x = x + 1; \]

Terse:

\[ x += 1; \]

Auto-increment;

\[ ++x; \]

(Each of the above is actually an expression, and it has a value: the new value of \( x \).)
– Next: Separate compilation –