600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Casting between types
  - Numbers
  - Pointers

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

```
<type-1> lhs;
<type-2> rhs;
lhs = rhs;
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are integers and `sizeof( LHS )>=sizeof( RHS )`
  ⇒ the conversion happens without loss of information

```
#include <stdio.h>
int main( void )
{
    char c = 'a';
    int i = c;
    printf( "%d -> %d\n" , c , i );
    return 0;
}
```

```
>> ./a.out
97 -> 97
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are integers and `sizeof( LHS )<sizeof( RHS )`
  ⇒ an implicit "modulo" operation is performed
  (modulo 2^b where b is the
  number of bits in the LHS)

```
#include <stdio.h>
int main( void )
{
    int i = 511;
    char c = i;
    printf( "%d -> %d\n" , i , c );
    return 0;
}
```

```
>> ./a.out
511 -> -1
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are floats and sizeof(LHS)>=sizeof(RHS)
  ⇒ the conversion happens without loss of information

```
#include <stdio.h>
int main( void )
{
    float f = 1.5;
    double d = f;
    printf( "%.8f -> %.8f\n" , f , d );
    return 0;
}
```

```
>> ./a.out
1.50000000 -> 1.50000000
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are floats and sizeof(LHS)<sizeof(RHS)
  ⇒ rounding is performed

```
#include <stdio.h>
int main( void )
{
    double d = 1.7;
    float f = d;
    printf( "%.8f -> %.8f\n" , d , f );
    return 0;
}
```

```
>> ./a.out
1.70000000 -> 1.70000005
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If the LHS is an integer and the RHS is a floating point value
  ⇒ the fractional part is discarded

```
#include <stdio.h>
int main( void )
{
    double d = -3.6;
    int i = d;
    printf( "%.8f -> %d\n" , d , i );
    return 0;
}
```

```
>> ./a.out
-3.60000000 -> -3
>>
```

Note that this is not the same thing as rounding down to the nearest integer

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If the LHS is a floating point value and the RHS is an integer
  ⇒ the closest floating point representation is used

```
#include <stdio.h>
int main( void )
{
    int i = 123456789;
    float f = i;
    printf( "%d -> %.0f\n" , i , f );
    return 0;
}
```

```
>> ./a.out
123456789 -> 123456792
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

- double → unsigned char:
  511.5 → 511 → 255

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

- double → unsigned char:
  511.5 → 511 → 255
- unsigned char → char:
  255 → -1

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

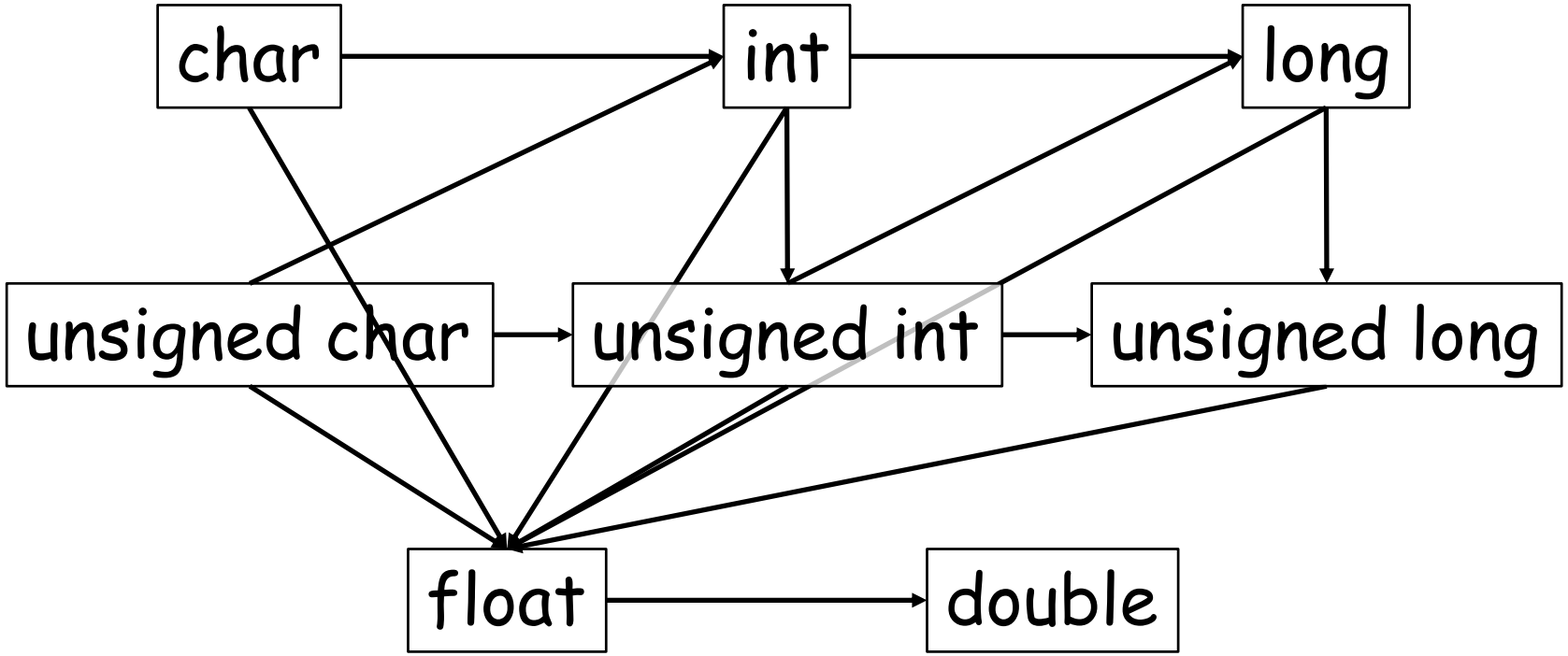
- double → unsigned char:
  511.5 → 511 → 255
- unsigned char → char:
  255 → -1
- char → float:
  -1 → -1.f

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

Casting between types (numbers)

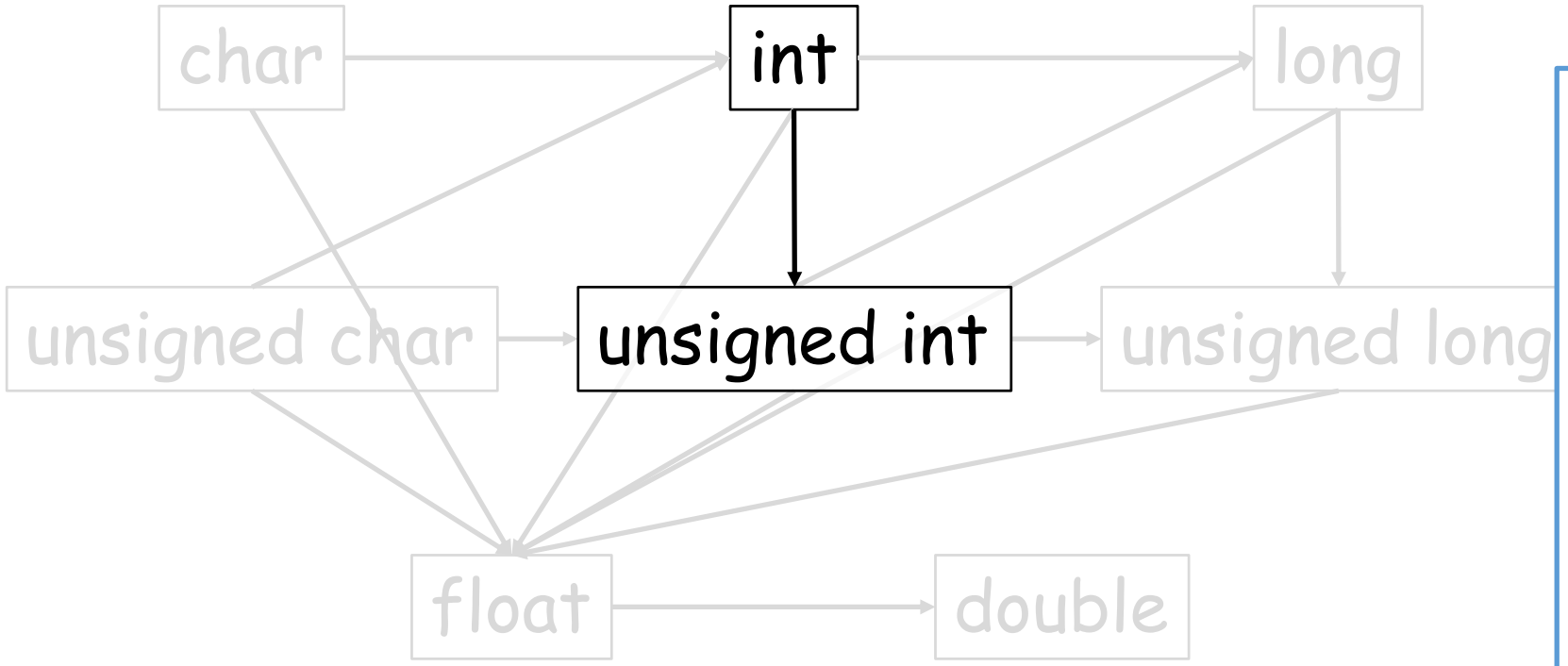
When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*



*If they are both integers and one is a char, it is first promoted to an int

Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*

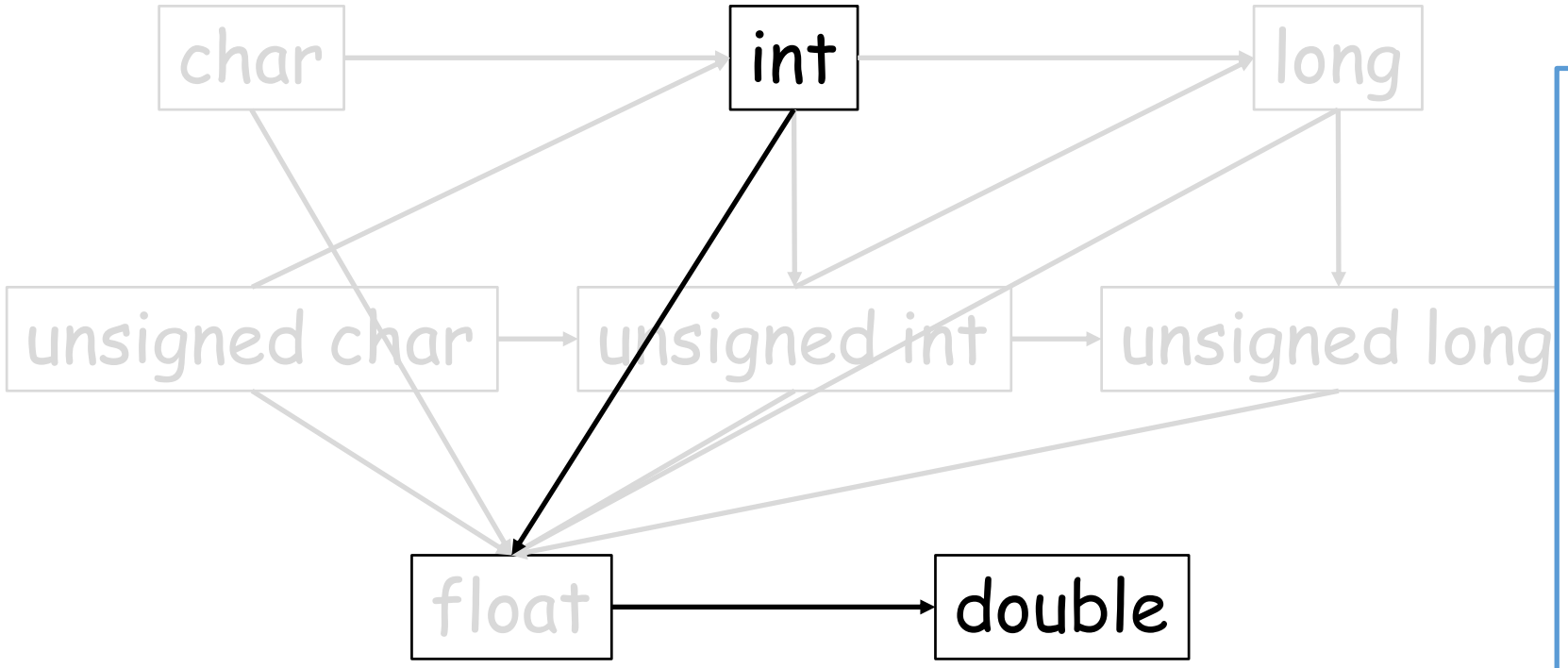



```
#include <stdio.h>
int main( void )
{
    int i = -1;
    unsigned int ui = 1;
    if( i<ui ) printf( "hi\n" );
    else printf( "bye\n" );
    return 0;
}
```

```
>> ./a.out
bye
>>
```

Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*

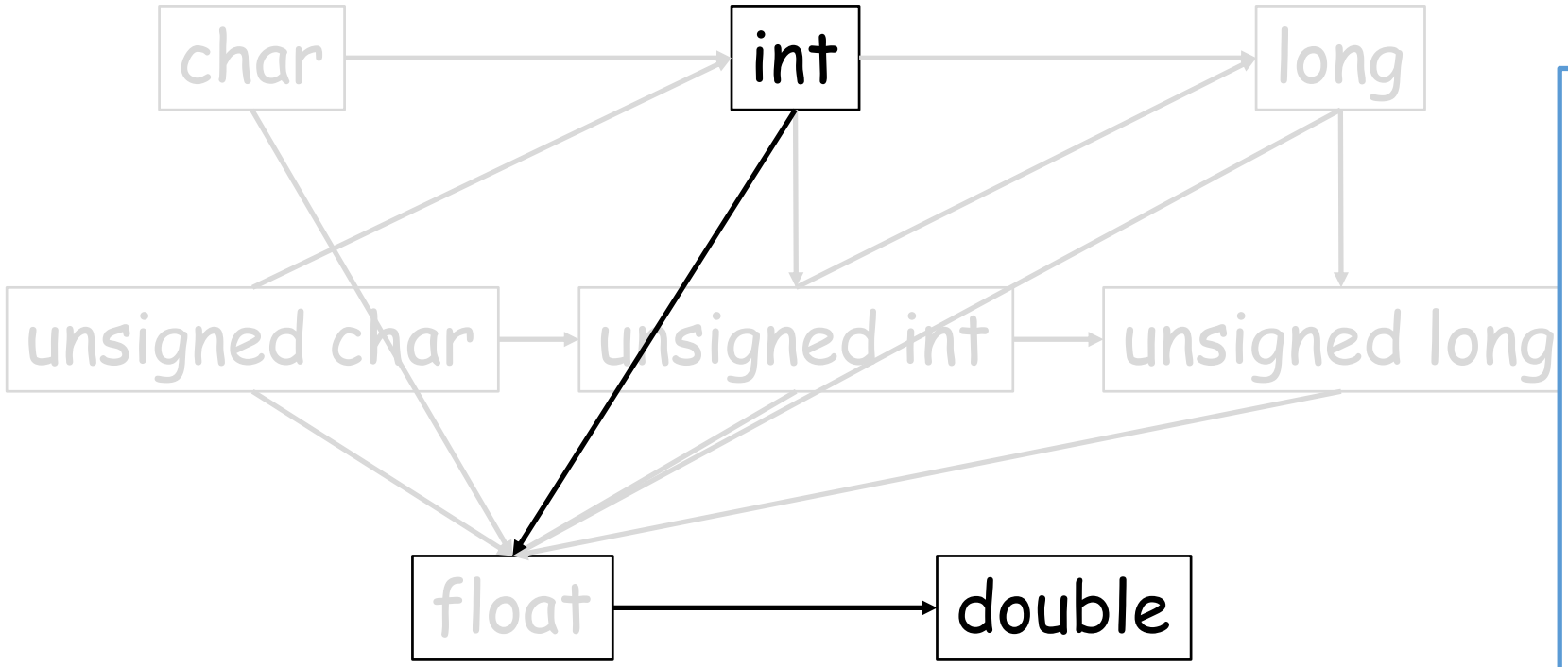



```
#include <stdio.h>
int main( void )
{
    int i = 2;
    double d = 2.5;
    i = i * d;
    printf( "%d\n" , i );
    return 0;
}
```

```
>> ./a.out
5
>>
```

Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*

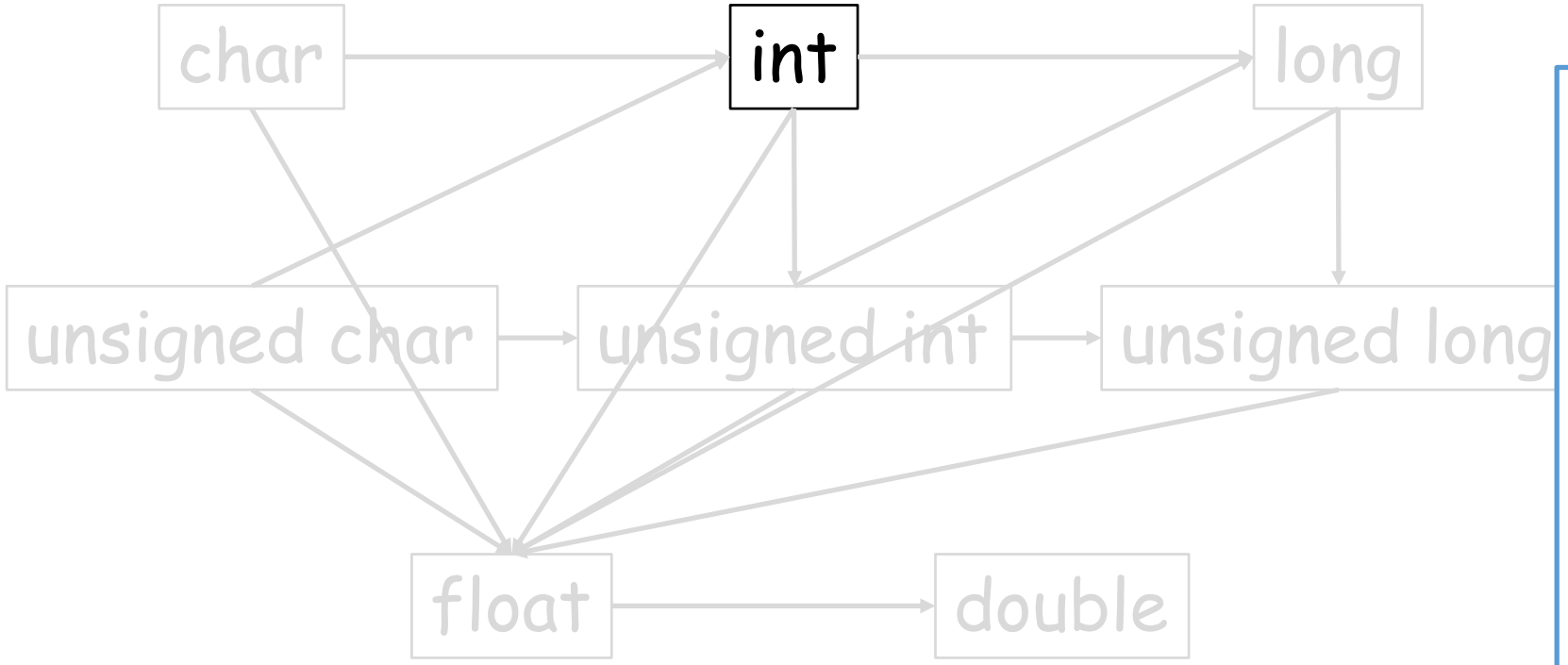



```
#include <stdio.h>
int main( void )
{
    int i = 2;
    double d = 2.5;
    i *= d;
    printf( "%d\n" , i );
    return 0;
}
```

```
>> ./a.out
5
>>
```

Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*

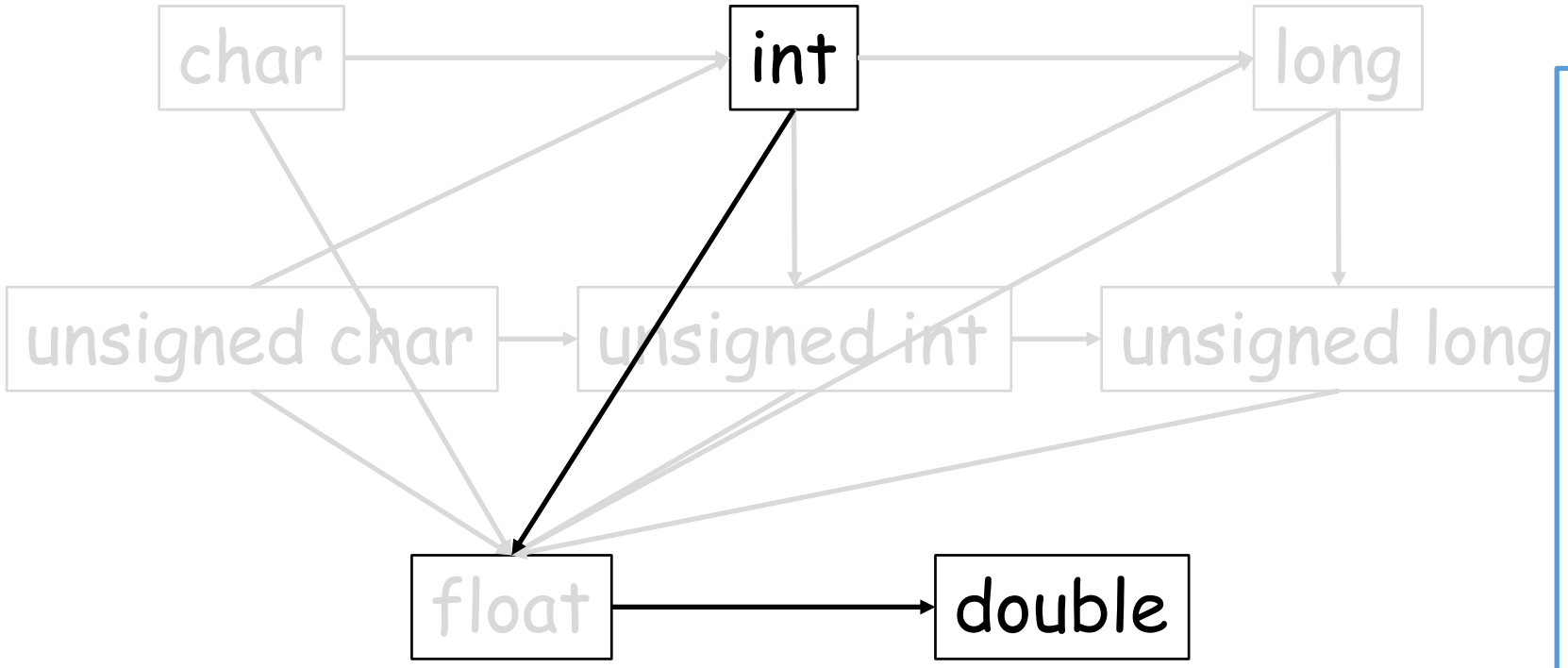



```
#include <stdio.h>
int main( void )
{
    int one = 1;
    int four = 4;
    int i = one / four * four;
    printf( "%d\n" , i );
    return 0;
}
```

```
>> ./a.out
0
>>
```

Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) the "lower ranked" operand is implicitly cast up (a.k.a. promoted)*

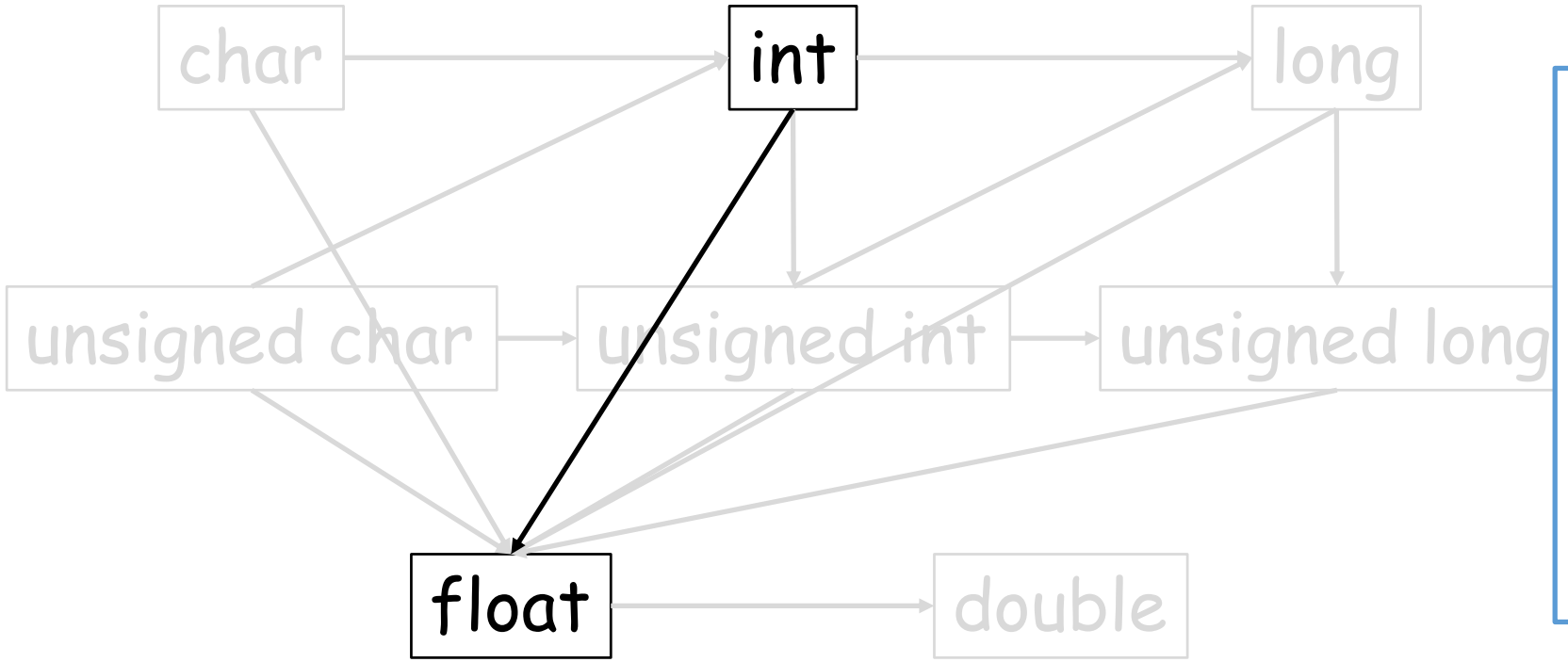



```
#include <stdio.h>
int main( void )
{
    double one = 1;
    int four = 4;
    int i = one / four * four;
    printf( "%d\n" , i );
    return 0;
}
```

```
>> ./a.out
1
>>
```

Casting between types (numbers)

- Since evaluation precedes assignment, we may get truncated results even when the LHS doesn't require it




```
#include <stdio.h>
int main( void )
{
    int one = 1 , four = 4;
    float f = one / four;
    printf( "%g\n" , f );
    return 0;
}
```

```
>> ./a.out
0
>>
```

Casting between types (numbers)

- Since evaluation precedes assignment, we may get truncated results even when the LHS doesn't require it
- The desired behavior can be forced with explicit casting:
  - Preceding the variable name with (<type-name>) converts the variable to type <type-name>
  - Since casting takes precedence over arithmetic operations:
    1. We convert one to a float
    2. And then divide a float by an int
       a. This implicitly promotes four to a float
       b. And then performs float by float division

```
#include <stdio.h>
int main( void )
{
    int one = 1 , four = 4;
    float f = (float)one / four;
    printf( "%g\n" , f );
    return 0;
}
```

```
>> ./a.out
0.25
>>
```

Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly

```
#include <stdio.h>
int main( void )
{
    …
    int i = 1;
    int* ip = &i;
    float *fp= (float*)ip;
    …
}
```

Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly
    - Unless one of them has type void*

```
...
void * malloc( size_t );
...
```

```
#include <stdio.h>
int main( void )
{
    ...
    float* a = malloc( 10 * sizeof( float ) );
    ...
}
```

Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly
    - Unless one of them has type `void*`
  - We can also explicitly cast between pointers and integers
    - This needs to be done with care since a pointer can have different sizes on different machines:
      - 4 bytes on a 32-bit machine
      - 8 bytes on a 64-bit machine
    - The `size_t` type is guaranteed to always have the size of a pointer

```
#include <stdio.h>
int main( void )
{
    int i = 100;
    int* ip = &i;
    size_t addr = (size_t)ip;
    printf( "Address is: %zu\n" , addr );
    return 0;
}
```

Casting between types

- A recursive function

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

Casting between types

- A recursive function
  - That allocates memory

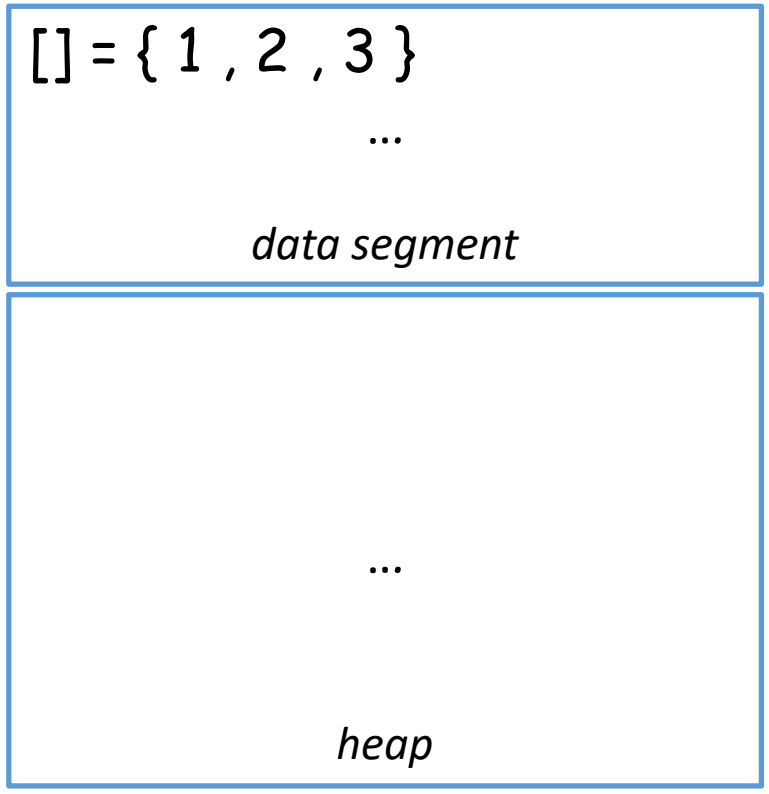
```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

Casting between types

- A recursive function
  - That allocates memory
  - And then prints outs its addresses

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

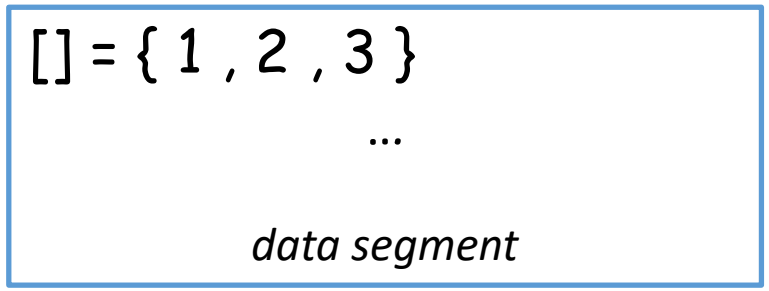
Casting between types

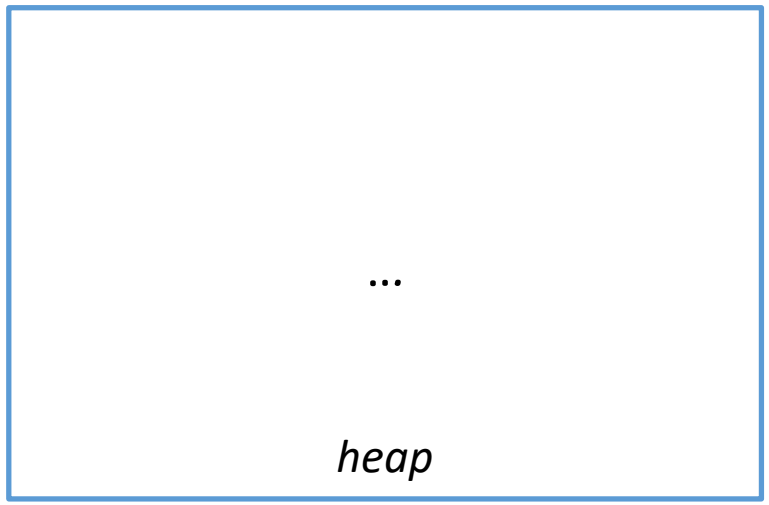



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
```

Casting between types

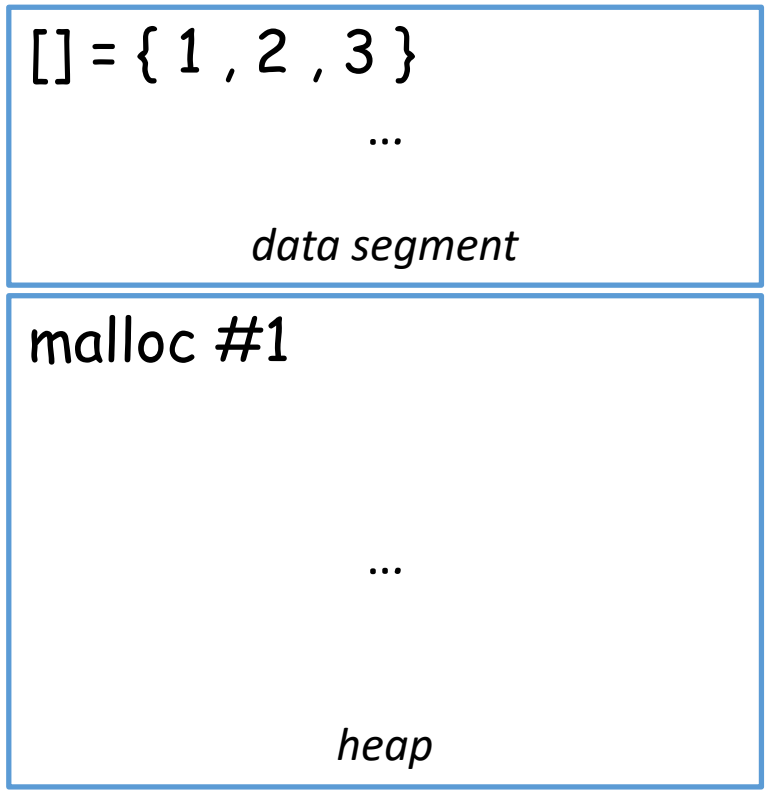





```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
```

Casting between types

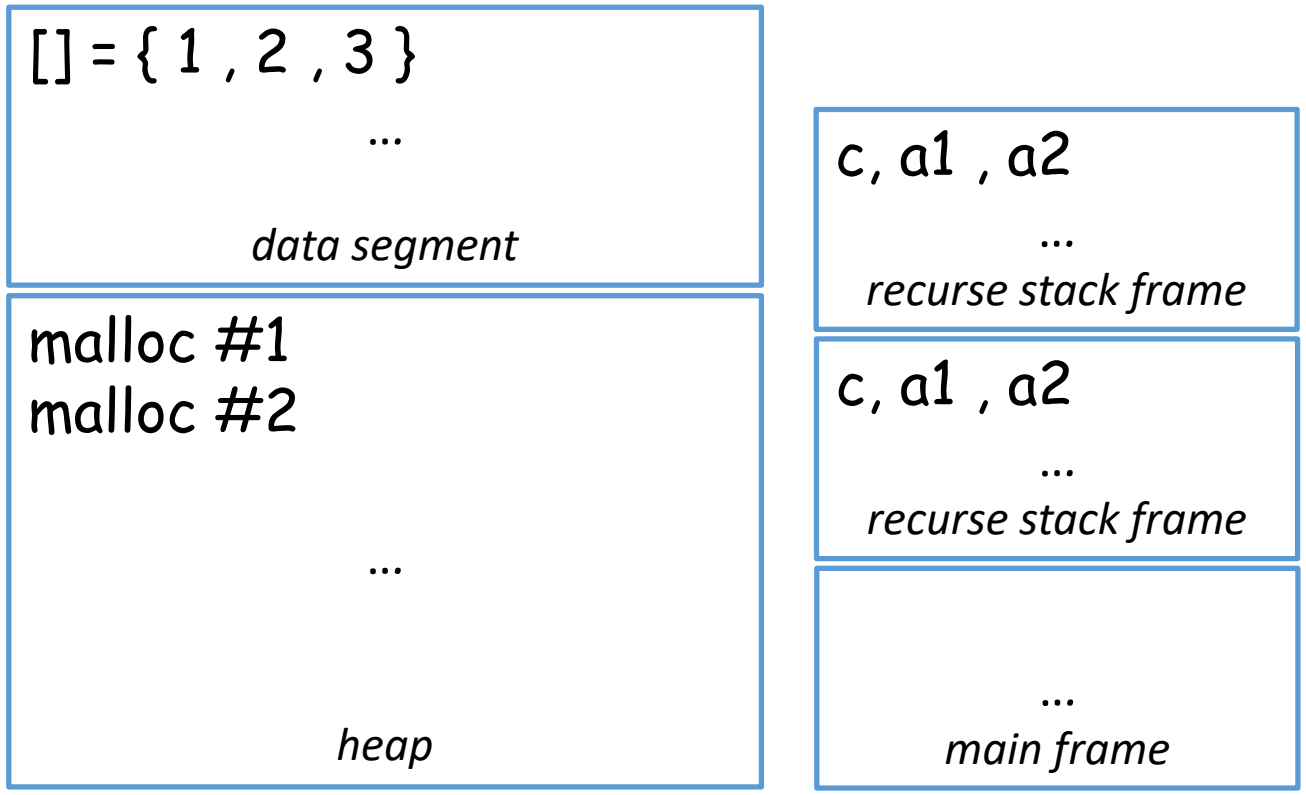



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=3

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
```

Casting between types

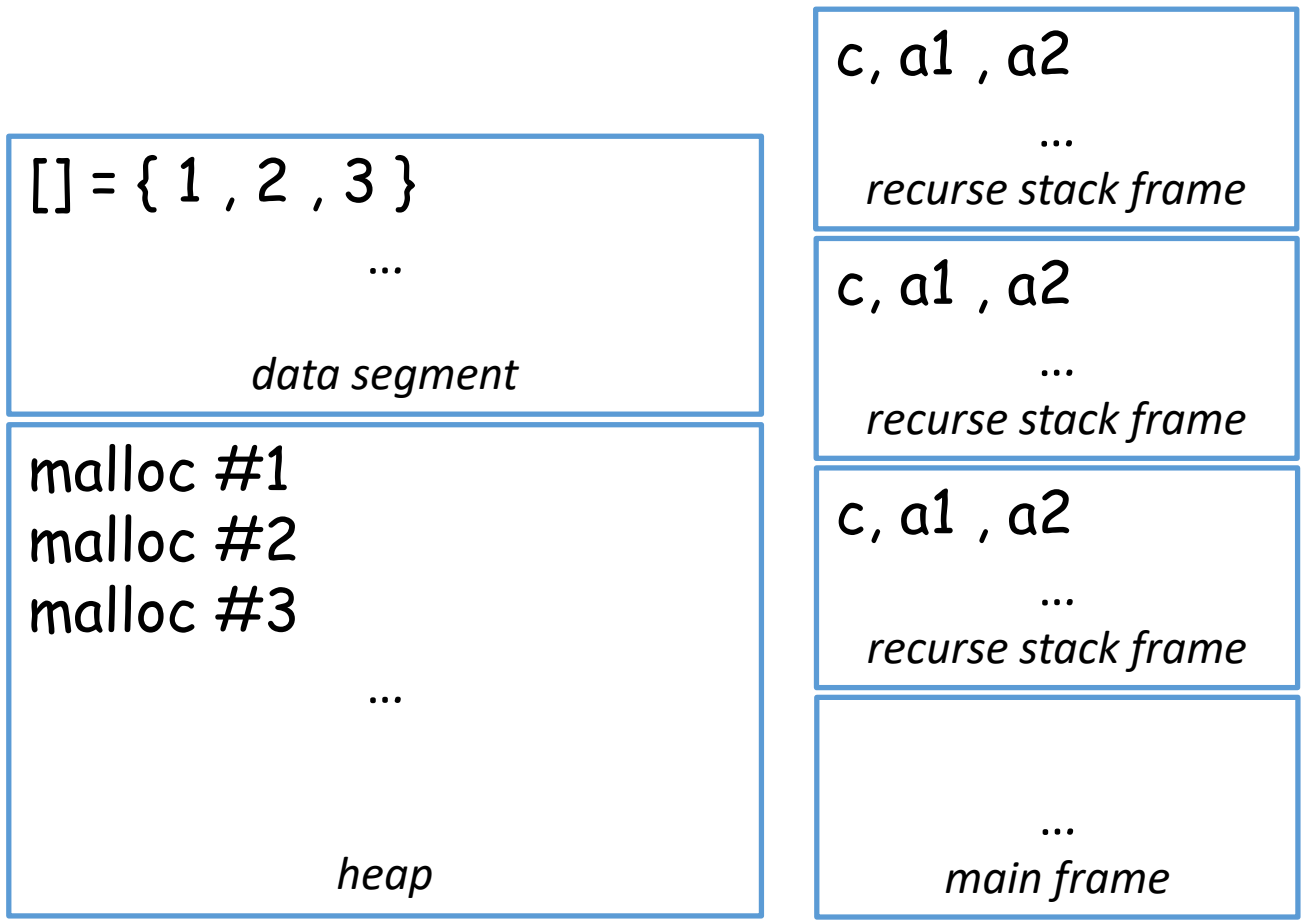



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=2

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
```

Casting between types

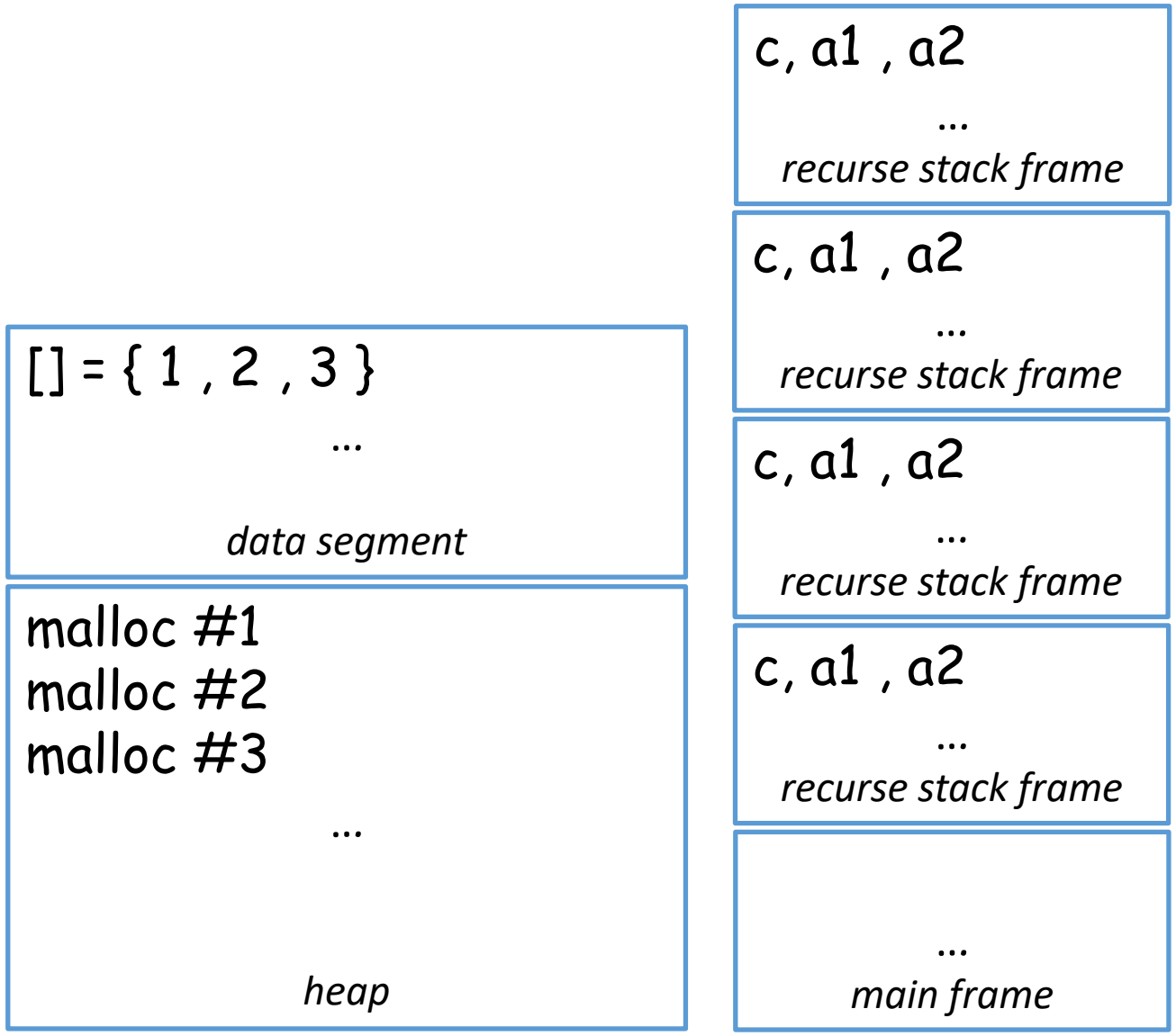



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=1

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types

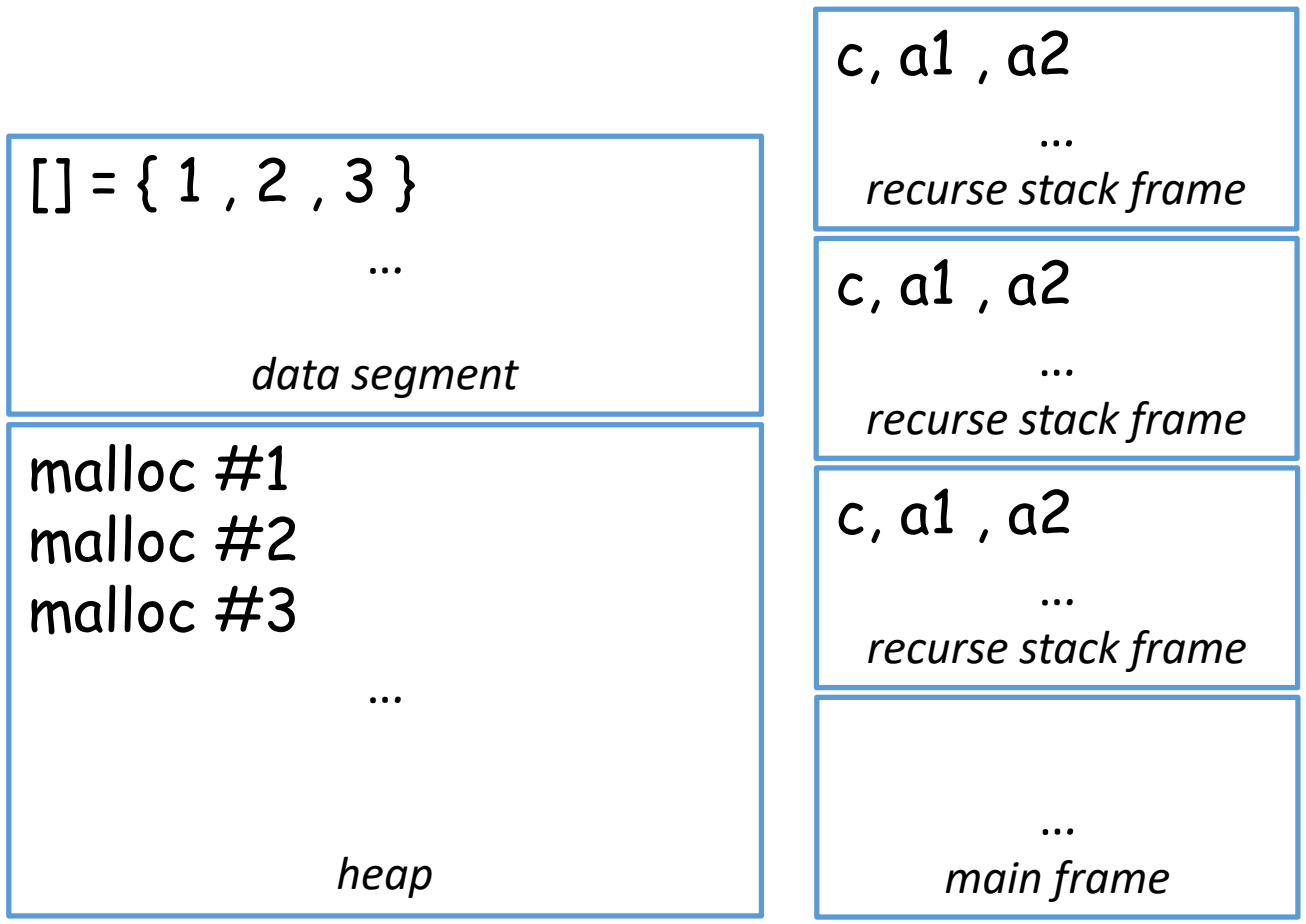



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=0

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types

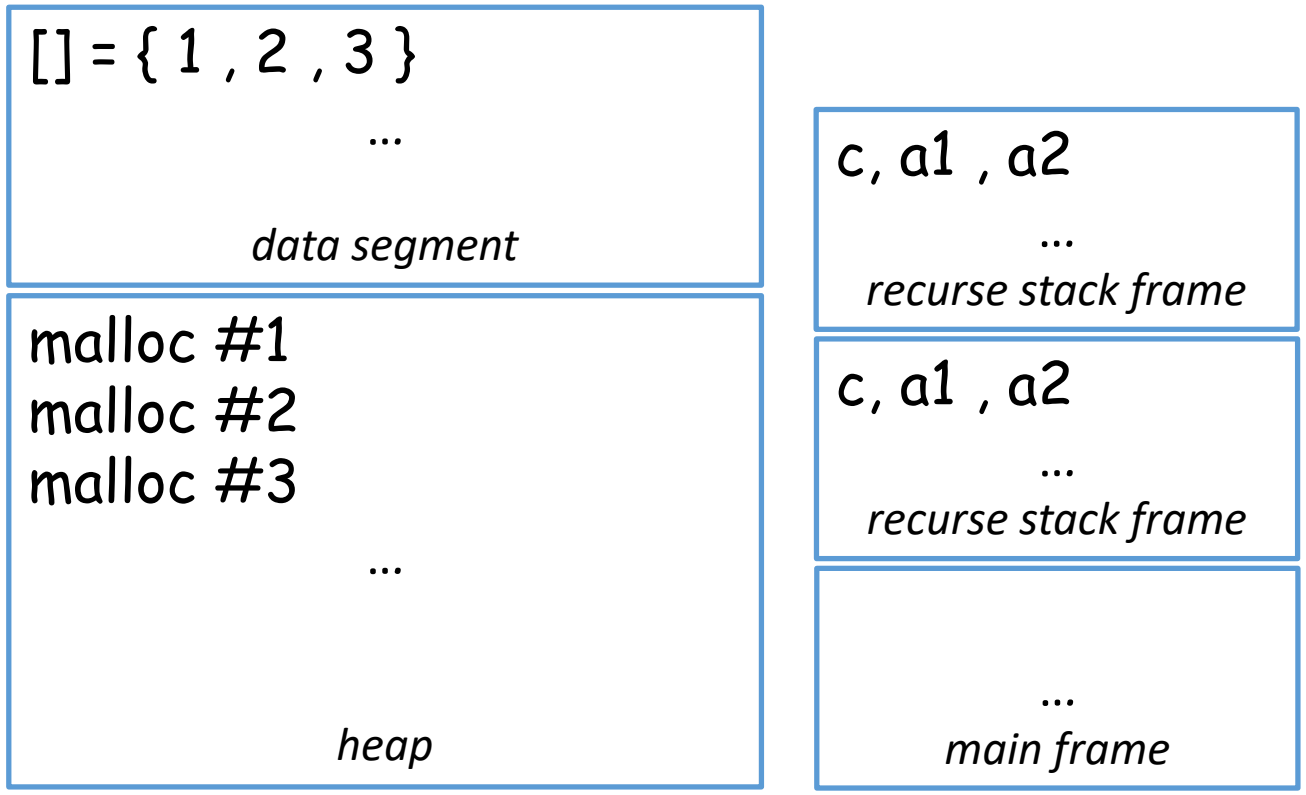



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=1

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types

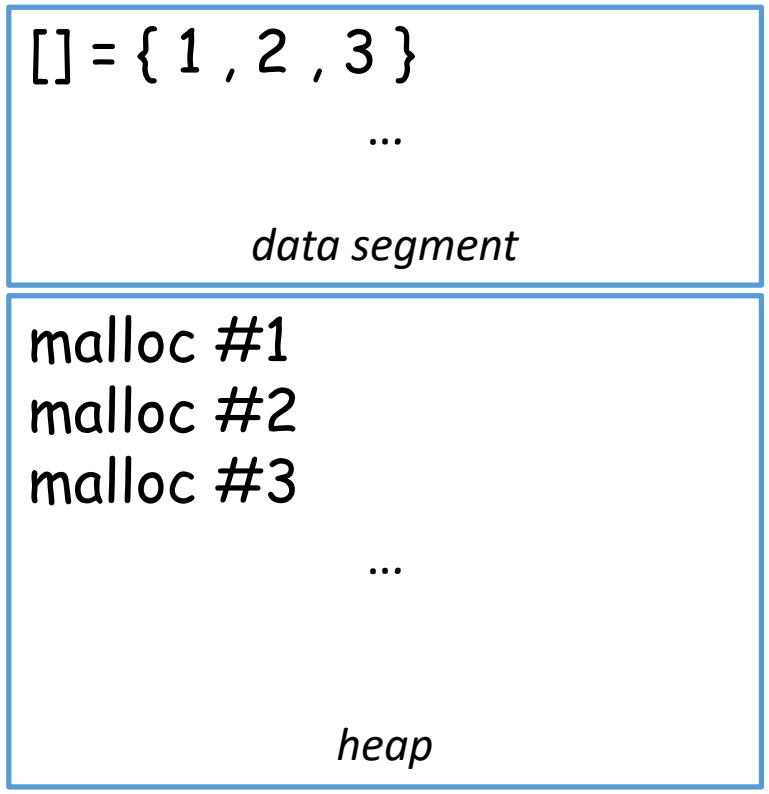


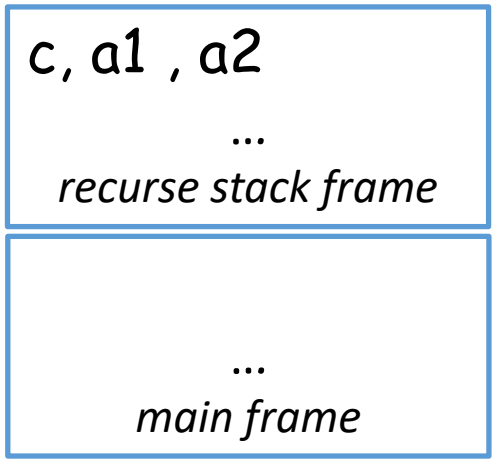

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=2

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types



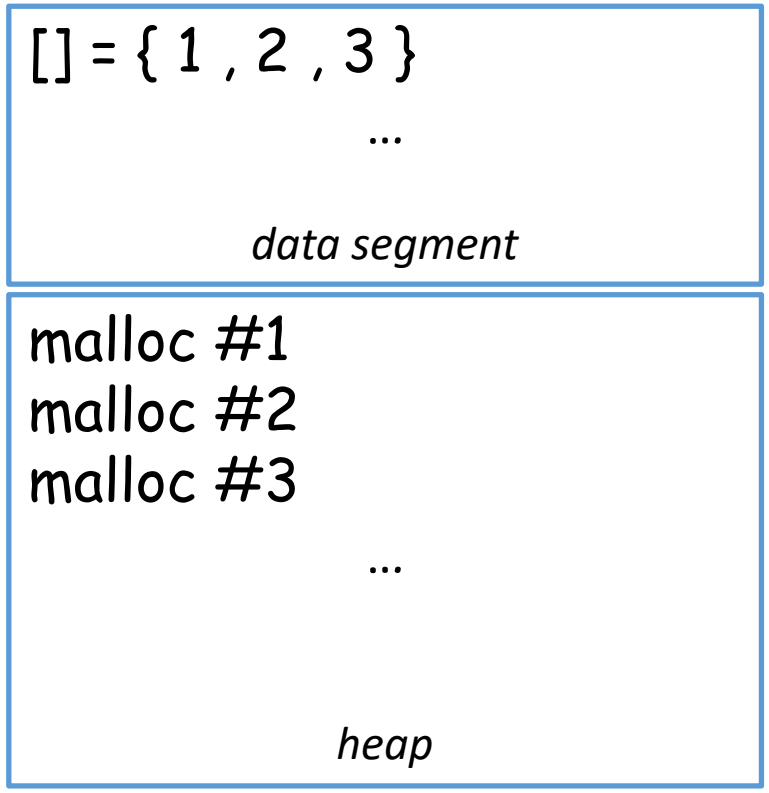



```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

c=3

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types




```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
```

Casting between types

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Casting between types

Variables:

- **static** are created once
  ⇒ The address **a1** does not change

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Casting between types

Variables:

- **static** are created once
  ⇒ The address **a1** does not change
- Dynamic are on the heap
  ⇒ The address **a2** increases

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Casting between types

Variables:

- **static** are created once
  ⇒ The address **a1** does not change
- Dynamic are on the heap
  ⇒ The address **a2** increases
- Local are on the stack frame
  ⇒ The address of **count** decreases

```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Casting between types

Variables:

- **static** are created once
  - ⇒ The address **a1** does not change
- Dynamic are on the heap
  - ⇒ The address **a2** increases
- Local are on the stack frame
  - ⇒ The address of **count** decreases
- Addresses ordered (small to large):
  - **static**
  - dynamic
  - local

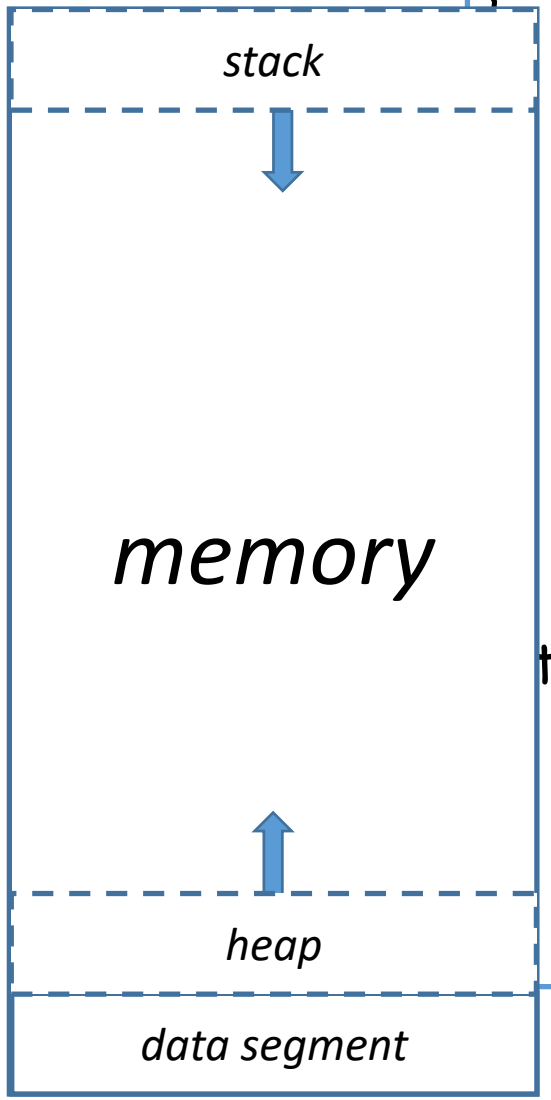
```
#include <stdio.h>
void recurse( int c )
{
    if( !c ) return;

    static int a1[] = { 1 , 2 , 3 };
    int* a2 = malloc( sizeof(int)*3 );
    printf( "S[%d] %zu\n" , c , (size_t)a1 );
    printf( "D[%d] %zu\n" , c , (size_t)a2 );
    printf( "L[%d] %zu\n" , c , (size_t)&c );
    recurse( --c );
}
int main( void )
{
    recurse( 3 );
    return 0;
}
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Casting between types

Memory Layout:

- static
  - The data segment is at the very bottom of the address space
- Dynamic
  - The heap starts after the data segment and grows upwards
- Local
  - The stack starts at the top and grows down




```
#include <stdio.h>
void recurse( int c )
{
  if( !c ) return;

  static int a1[] = { 1 , 2 , 3 };
  int* a2 = malloc( sizeof(int)*3 );
  printf( "S[%d] %zu\n" , c , (size_t)a1 );
  printf( "D[%d] %zu\n" , c , (size_t)a2 );
  printf( "L[%d] %zu\n" , c , (size_t)&c );
  recurse( --c );

t main( void )

  recurse( 3 );
  return 0;
```

```
>> ./a.out
S[3] 6295600
D[3] 11928672
L[3] 140731509410092
S[2] 6295600
D[2] 11928704
L[2] 140731509410044
S[1] 6295600
D[1] 11928736
L[1] 140731509409996
>>
```

Piazza → Resources section → Resources tab → Exercise 4-2