CS 354
Machine Organization and Programming

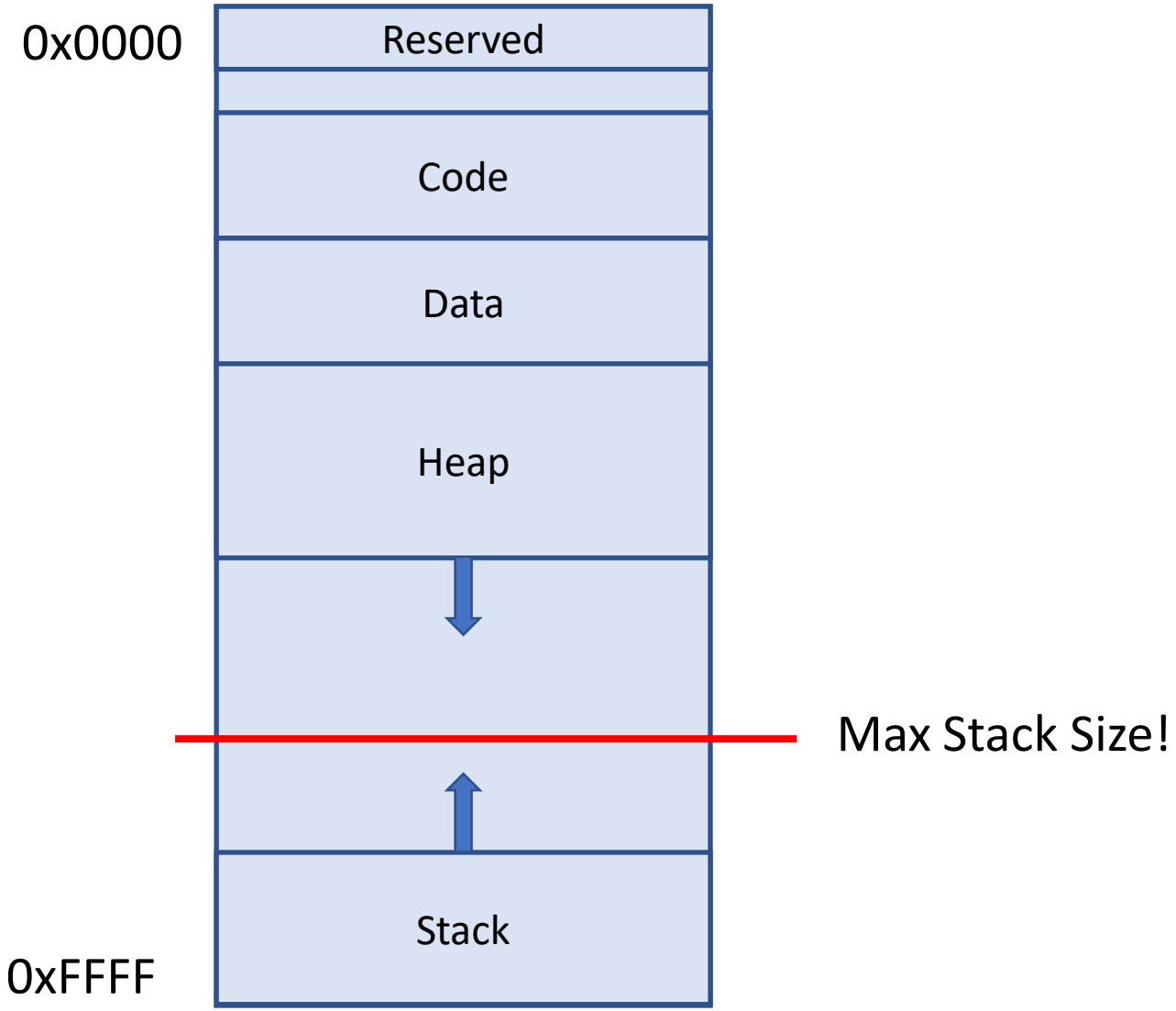
Lecture 07

Michael Doescher
Summer 2020

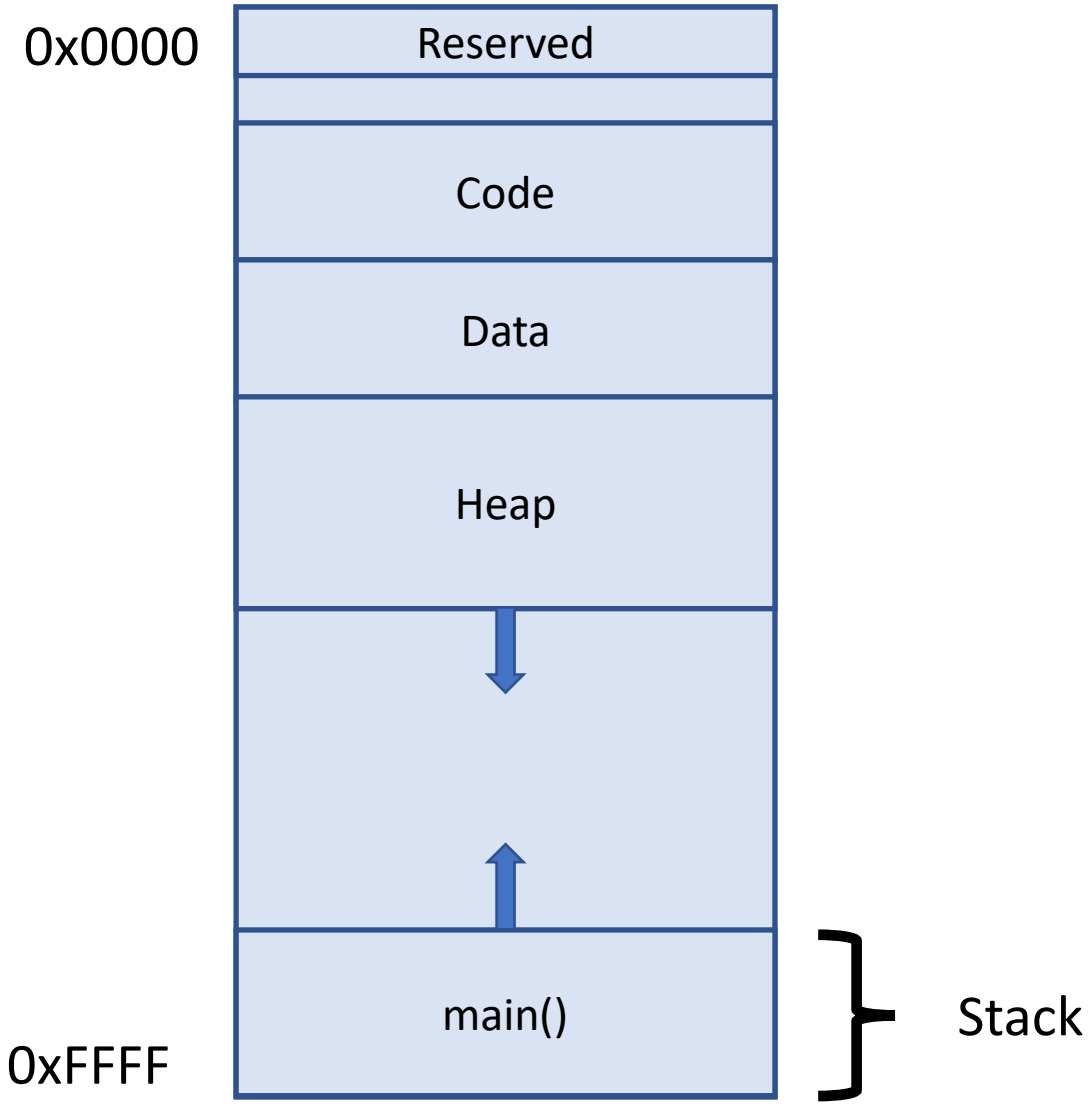
Stack
Dynamic Memory Allocation
Heap
malloc

Complete Memory Address Space

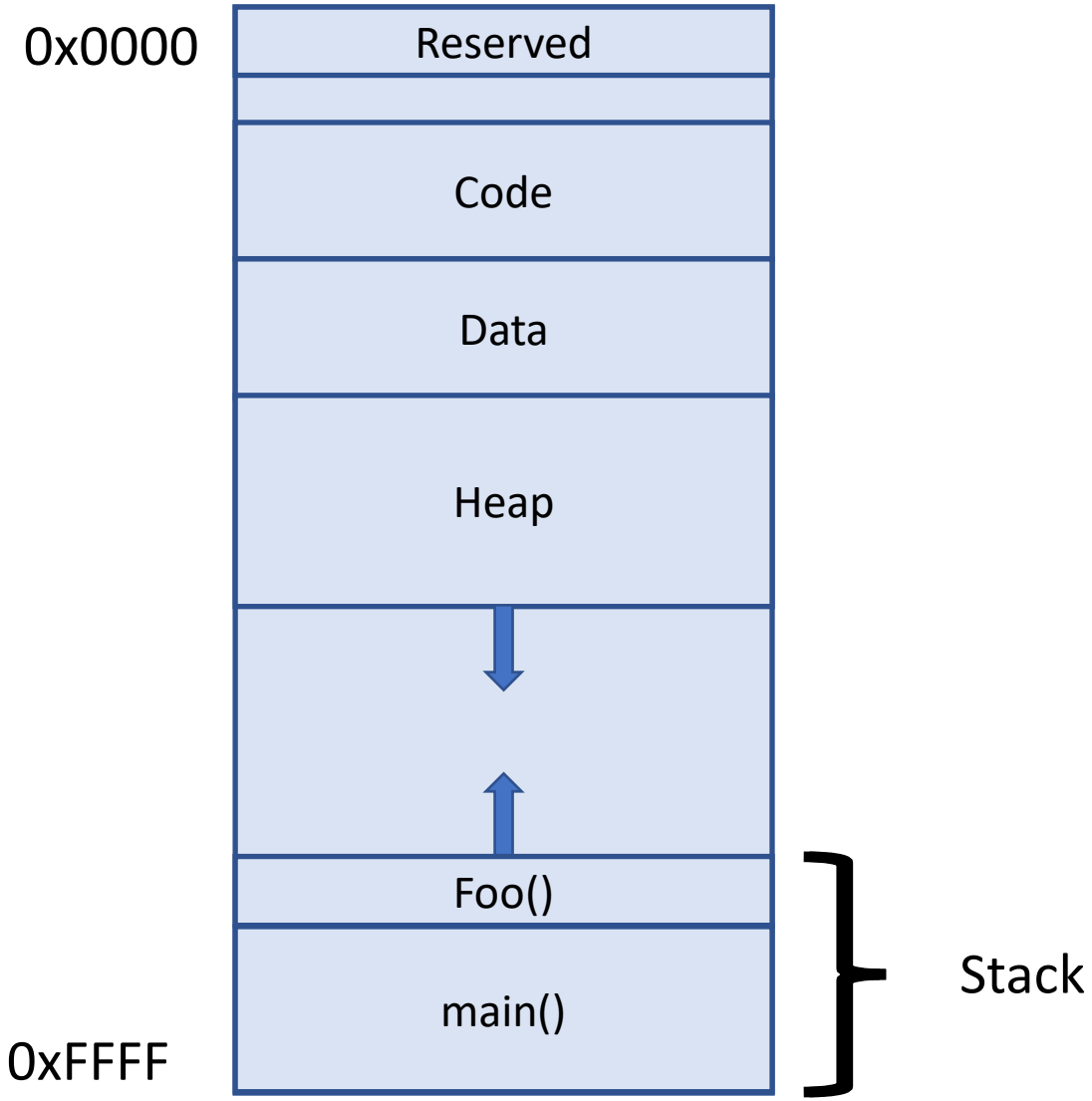
Complete Memory Address Space



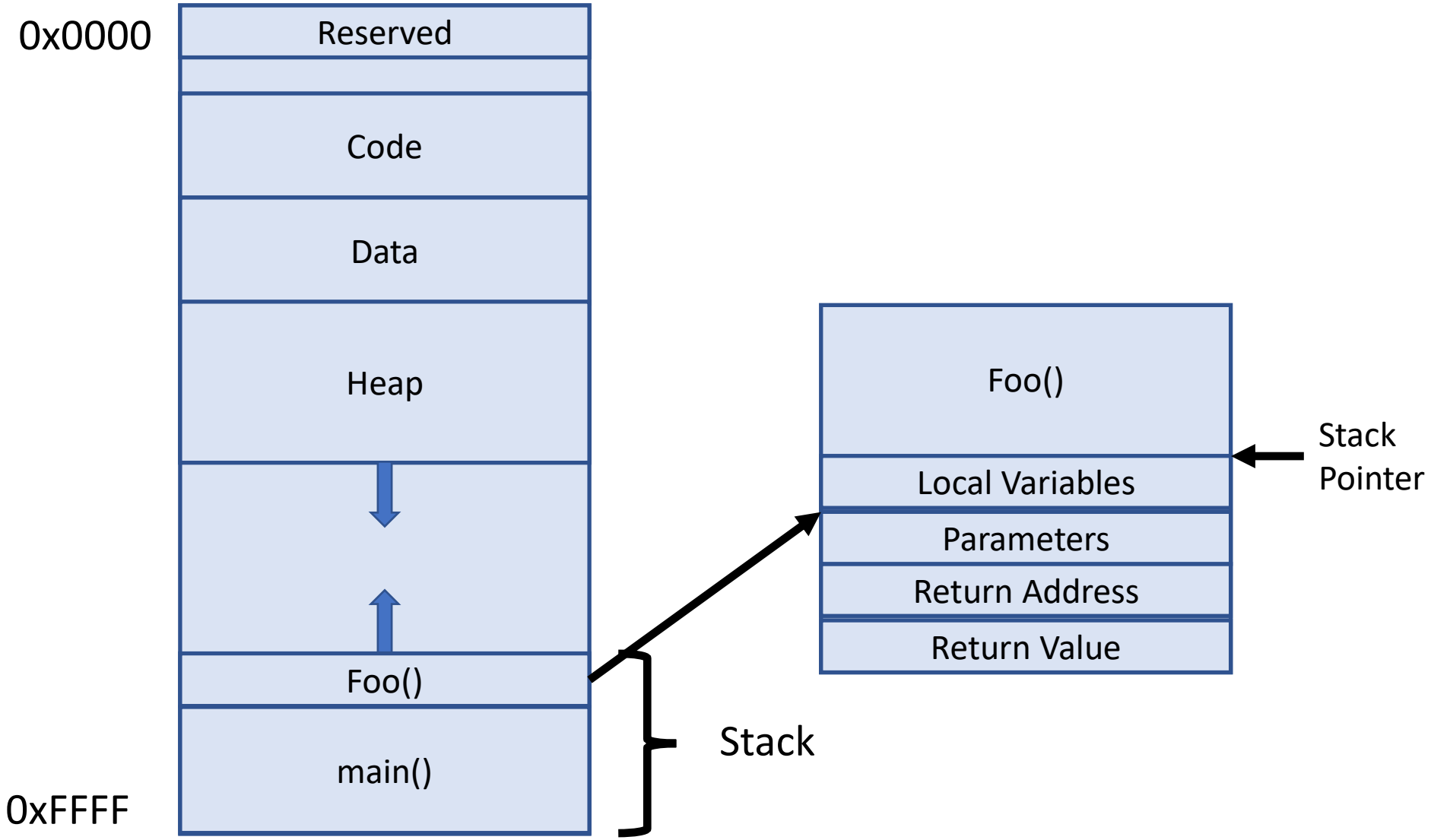
Complete Memory Address Space



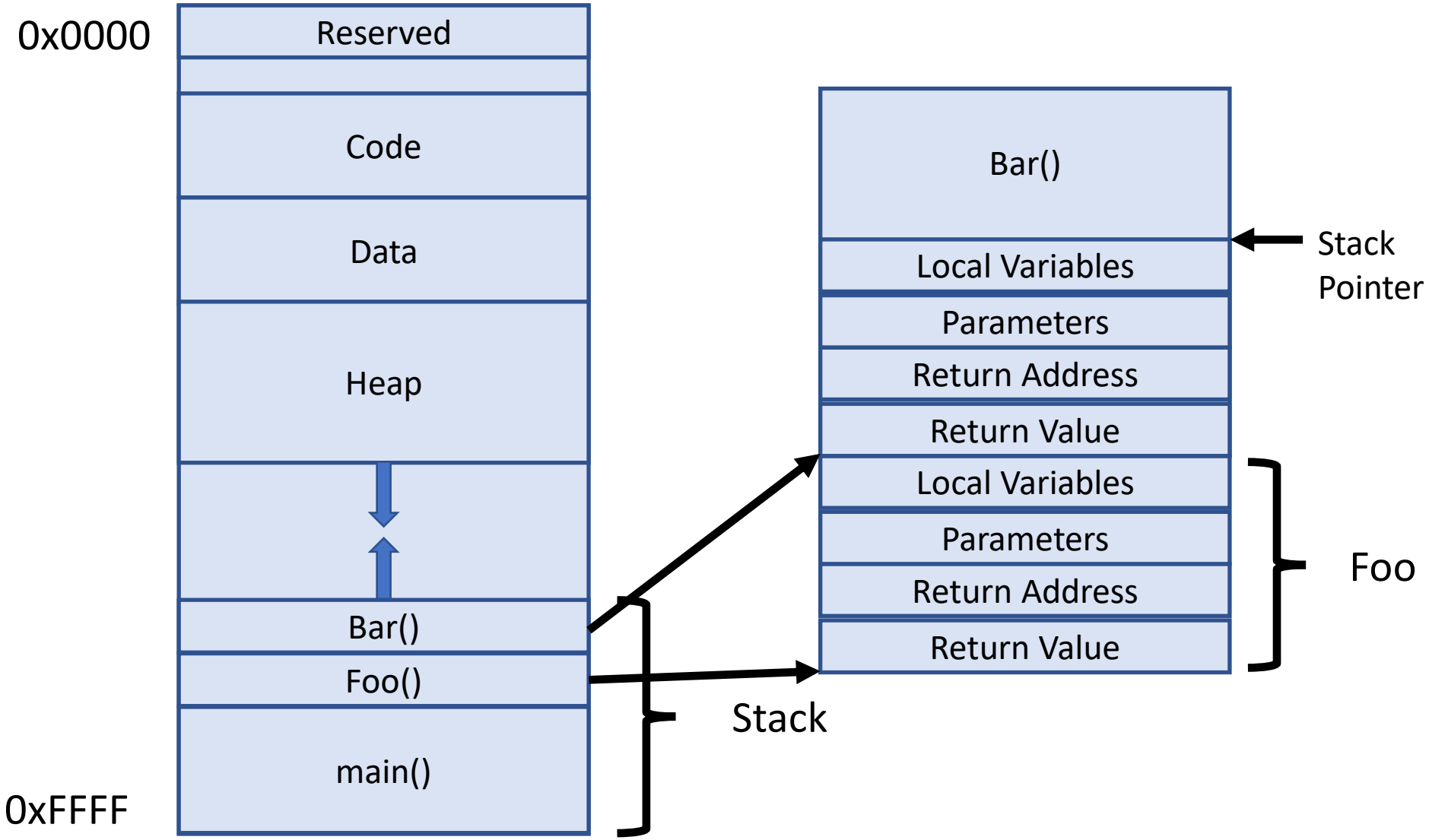
Complete Memory Address Space



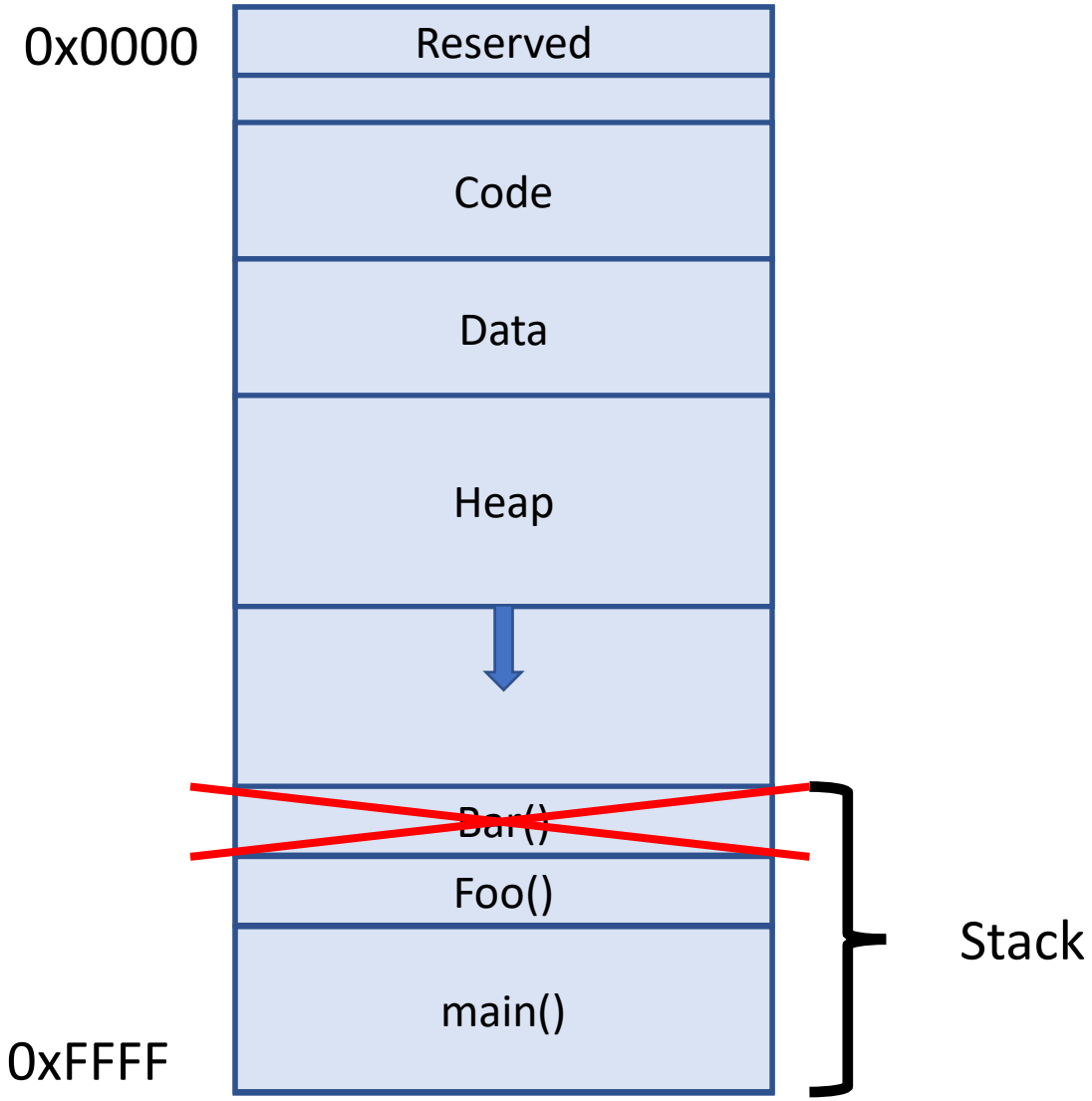
Complete Memory Address Space



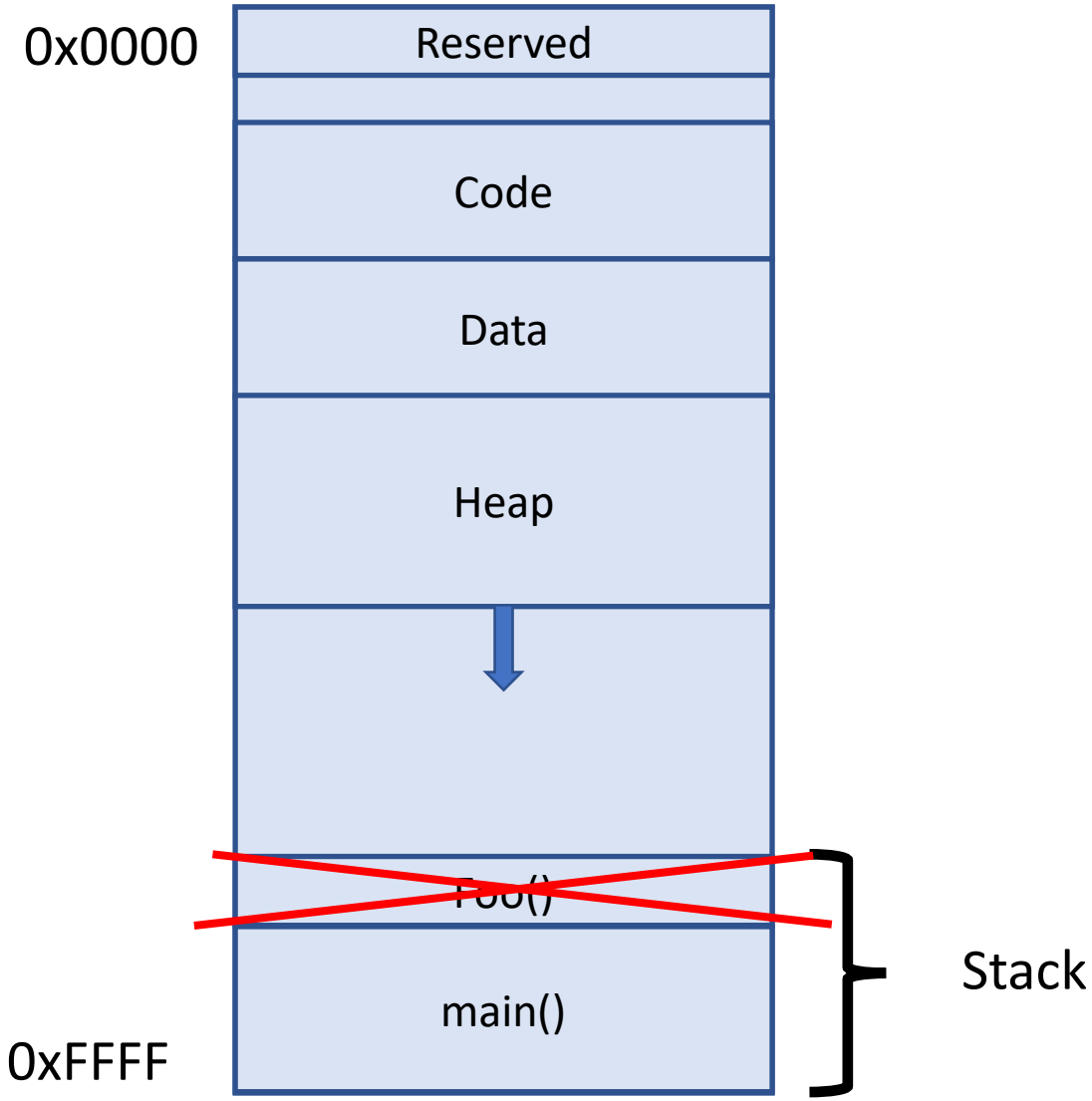
Complete Memory Address Space



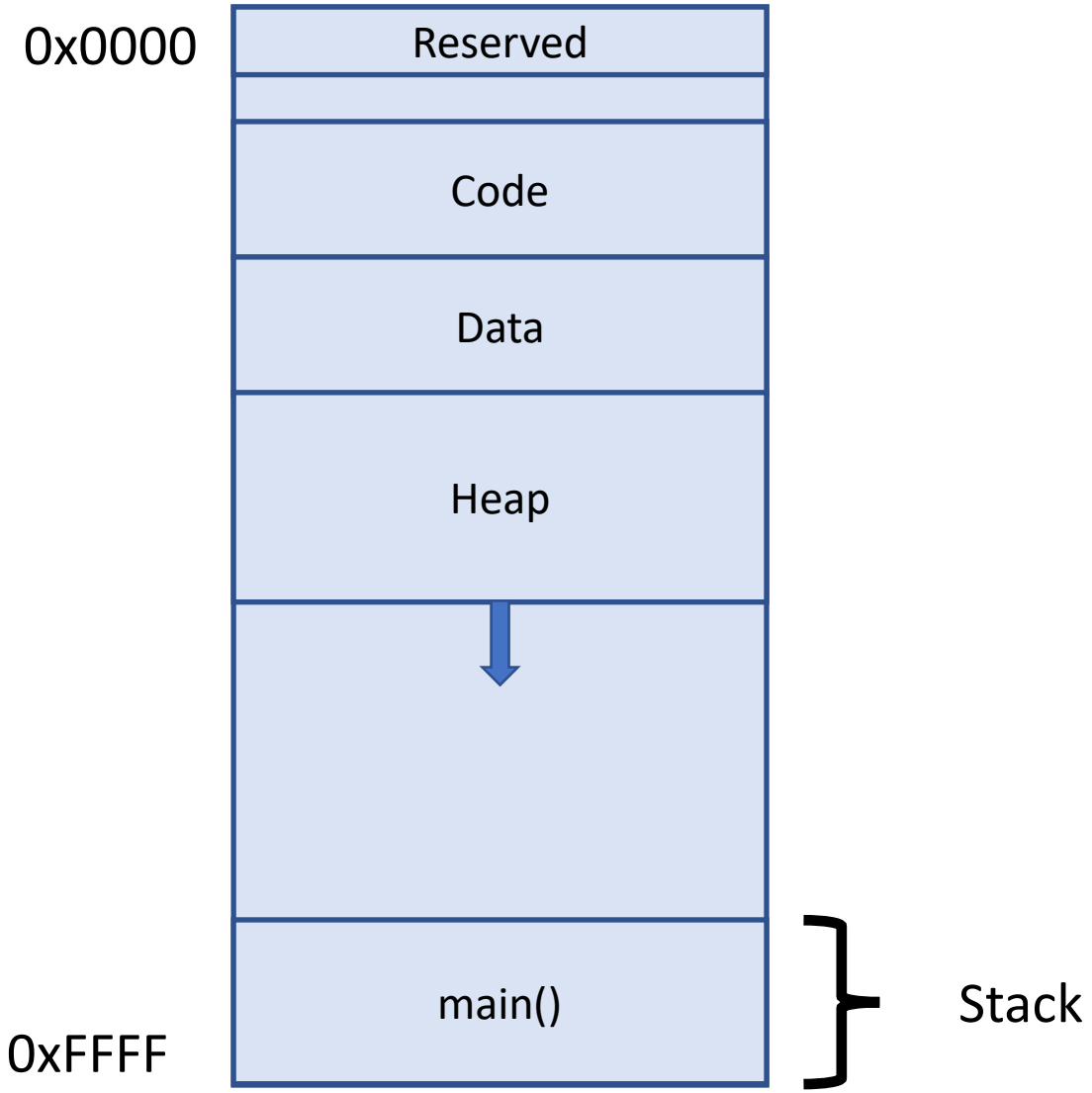
Complete Memory Address Space



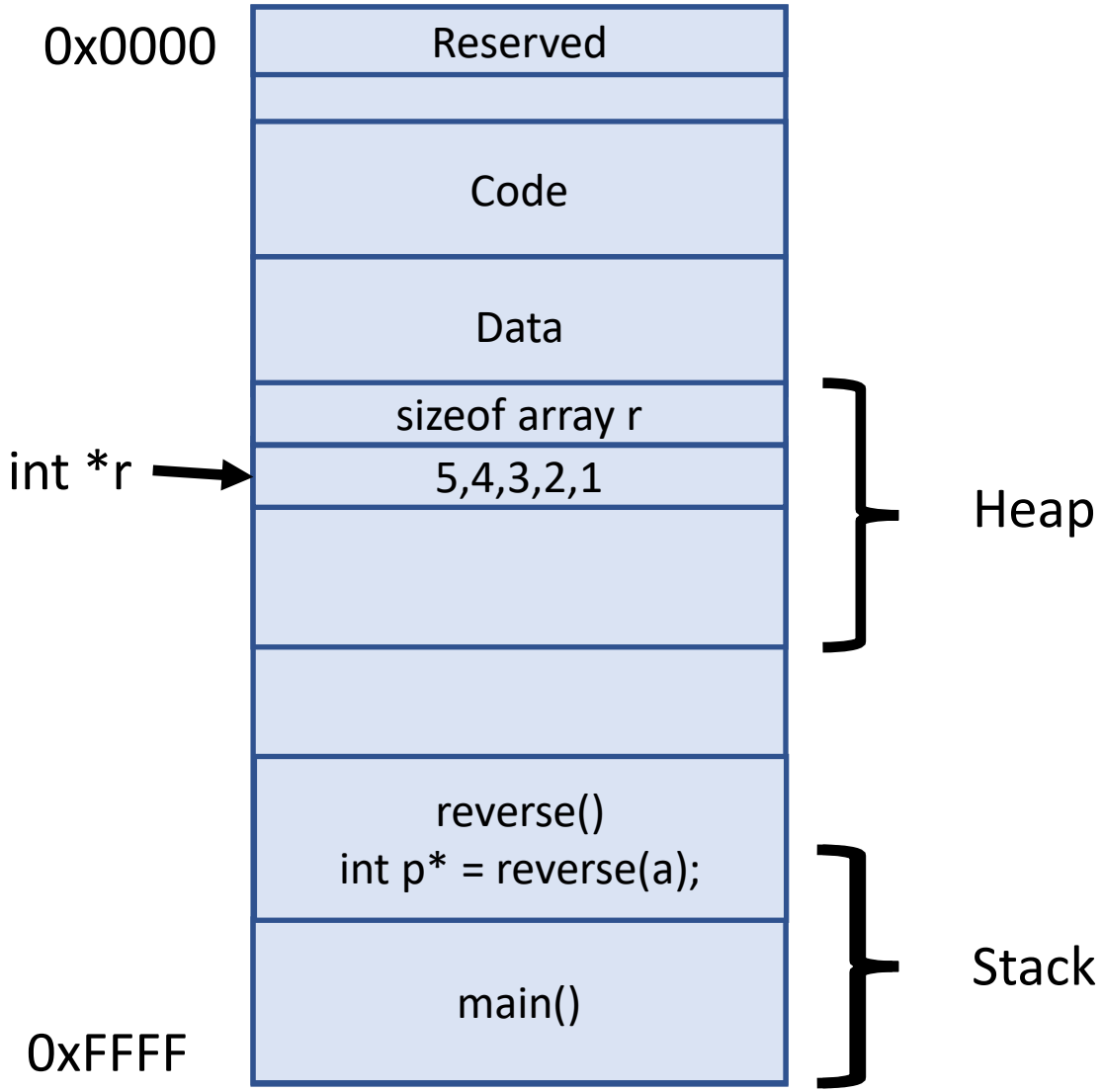
Complete Memory Address Space



Complete Memory Address Space



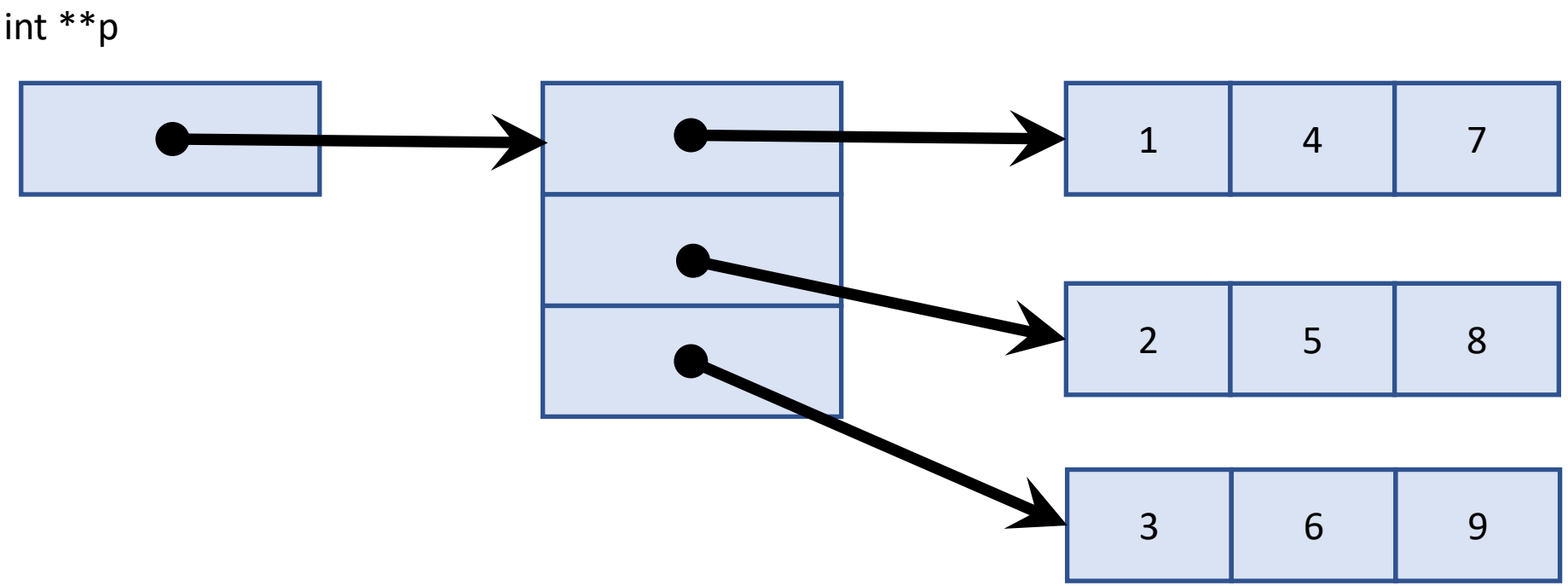
Complete Memory Address Space



Common malloc Errors

1. writing more than we allocated - out of bounds
   e.g. strlen(s) vs strlen(s) +1
2. allocating 0 sized blocks of memory
3. using pointers that have been freed
4. double free
5. freeing non-malloc pointers
6. realloc null pointers

Transpose

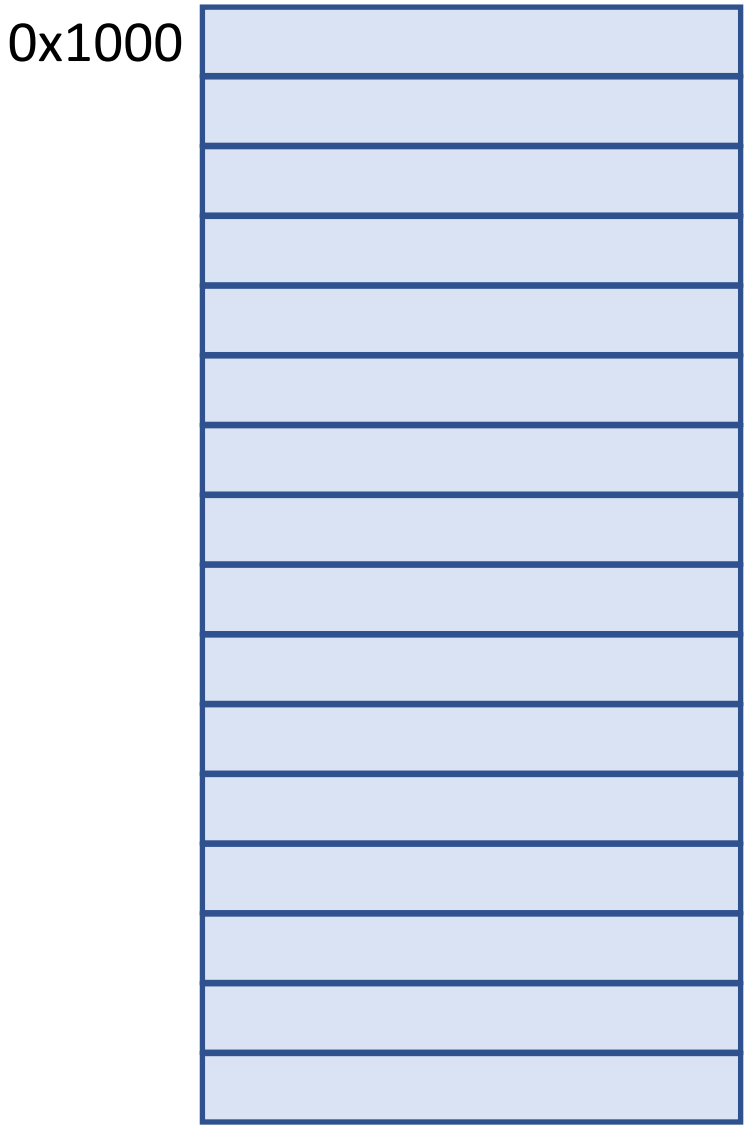


2d arrays and 1d memory

```
int a[3][5];

// initialize
for (int i=0;i<3;i++)
for (int j=0;j<5;j++)
a[i][j] = 10*i + j
```

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24



2d arrays and 1d memory

```
int a[3][5];

// initialize
for (int i=0;i<3;i++)
for (int j=0;j<5;j++)
a[i][j] = 10*i + j
```

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
	01
	02
	03
	04
	10
	11
	12
	13
	14
	20
	21
	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

// initialize
for (int i=0;i<3;i++)
for (int j=0;j<5;j++)
a[i][j] = 10*i + j

void print_array(int a[][]) {…}
```

Consider just the line
printf(“%d”,a[1][2]);
How does compiler know how to do this?

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
	01
	02
	03
	04
	10
	11
	12
	13
	14
	20
	21
	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

// initialize
for (int i=0;i<3;i++)
for (int j=0;j<5;j++)
a[i][j] = 10*i + j

void print_array(int a[][]) {…}
```

Consider just the line
printf(“%d”,a[1][2]);
How does compiler know how to do this?
What is the address of this element?

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
	01
	02
	03
	04
	10
	11
	12
	13
	14
	20
	21
	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

// initialize
for (int i=0;i<3;i++)
for (int j=0;j<5;j++)
a[i][j] = 10*i + j

void print_array(int a[][]) {…}
```

Consider just the line

```
    printf("%d",a[1][2]);
```

How does compiler know how to do this?
What is the address of this element?

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
```

Consider just the line

```
    printf("%d",a[1][2]);
```

How does compiler know how to do this?

For 1d array

*(a + i*sizeof(int))

Write a similar formula for a 2d array.

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
```

Consider just the line

```
    printf("%d",a[1][2]);
```

How does compiler know how to do this?

For 1d array

*(a + i*sizeof(int))

Write a similar formula for a 2d array.

Asume sizeof(int) = 4

*(a + (5 * 1 * 4) + (2 * 4))

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
    printf("%d",a[1][2]);
```

Asume sizeof(int) = 4

*(a + (5 * 1 * 4) + (2 * 4))

20 + 8 = 28 --> 0x1C

= *(0x1000 + 0x1C)

= *(0x101C)

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
    printf("%d",a[1][2]);
```

Asume sizeof(int) = 4
*(a + (5 * 1 * 4) + (2 * 4)
20 + 8 = 28 --> 0x1C
= *(0x1000 + 0x1C)
= *(0x101C)

Does the compiler know there are 3 rows and 5 columns?

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
    printf("%d",a[1][2]);
```

Asume sizeof(int) = 4

*(a + (5 * 1 * 4) + (2 * 4)

20 + 8 = 28 --> 0x1C

= *(0x1000 + 0x1C)

= *(0x101C)

Does the compiler know there are 3 rows and 5 columns?

NO!!!

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][]) {…}
    printf("%d",a[1][2]);
```

Compiler only knows:
a → 0x1000
Type →int

It's missing the 5 columns!!!

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[][5]) {…}
    printf("%d",a[1][2]);
```

Compiler only knows:
a → 0x1000
Type →int

It's missing the 5 columns!!!

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[3][5]) {…}
    printf("%d",a[1][2]);
```

Compiler only knows:
a → 0x1000
Type →int

The 3 rows is optional. The compiler will ignore this if we put it in.

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[3][5]) {…}
    printf("%d",a[1][2]);
```

Common question
Where does the 5 come from?

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

```
int a[3][5];

void print_array(int a[3][5]) {…}
    printf("%d",a[1][2]);
```

Common question
Where does the 5 come from?

The programmer is responsible for knowing the size of the array.

```
#define MAXROW 3
#define MAXCOL 5
```

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

2d arrays and 1d memory

Multidimensional arrays require we provide functions with all but the first dimension.

```
char c[3][5][3];

// initialize

Print_Array(c);

void Print_Array(char c[][5][3]) {}
```

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

0x1000	00
0x1004	01
0x1008	02
0x100C	03
0x1010	04
0x1014	10
0x1018	11
0x101C	12
0x1020	13
0x1024	14
0x1028	20
0x102C	21
0x1030	22
	23
	24

CS 354
Machine Organization and Programming

Lecture 06B

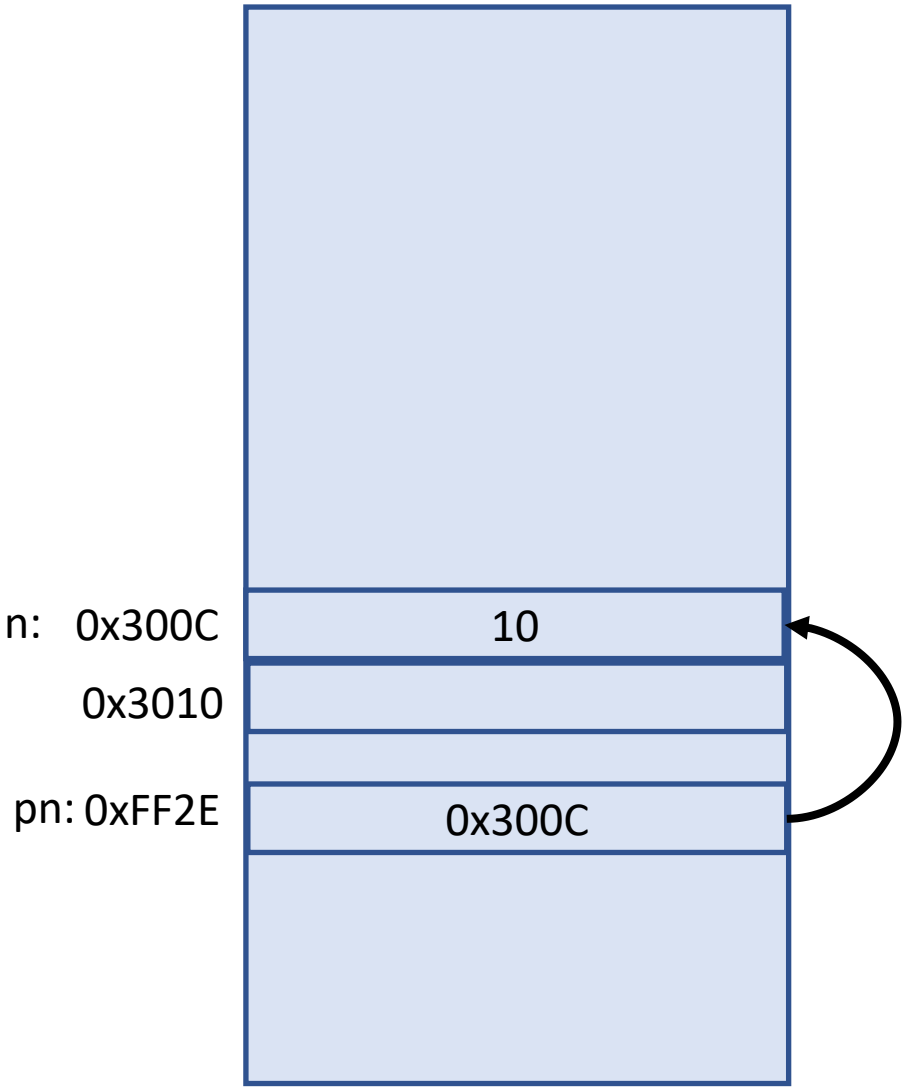
Michael Doescher
Summer 2020

Pointers to Pointers
Command Line Arguments

Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;
```

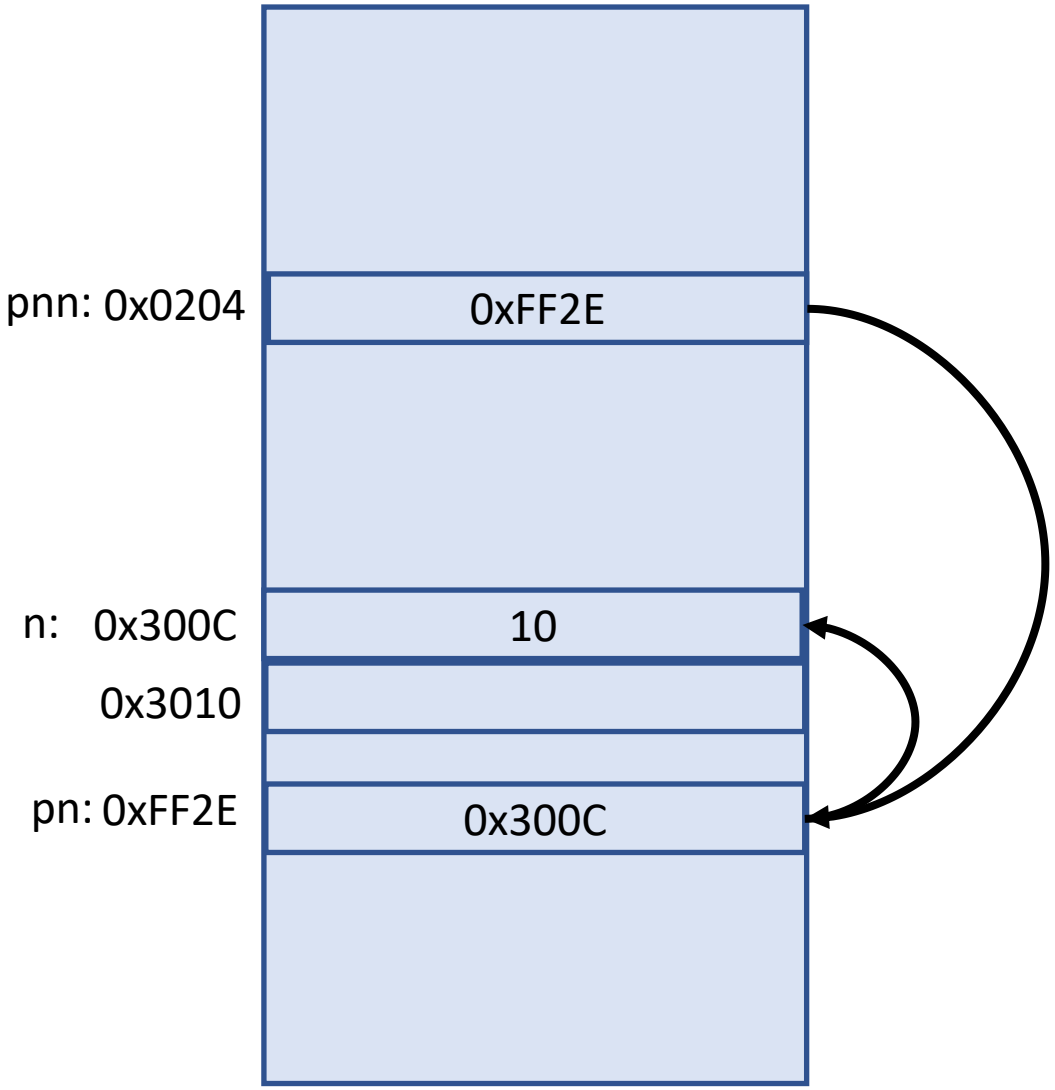


Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn
```



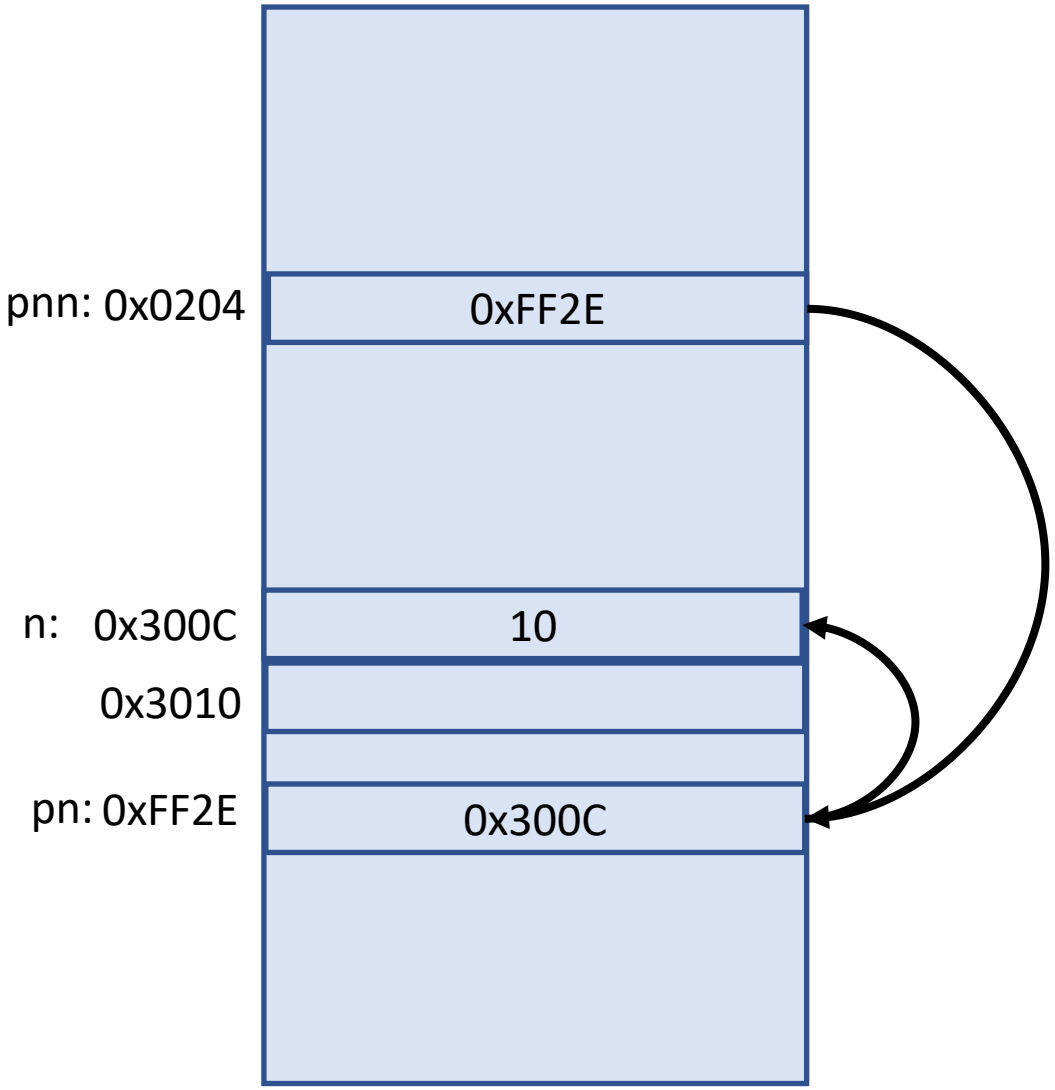
Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn


// Access n
n = 3;
*pn = 3;
**ppn = 3;
```



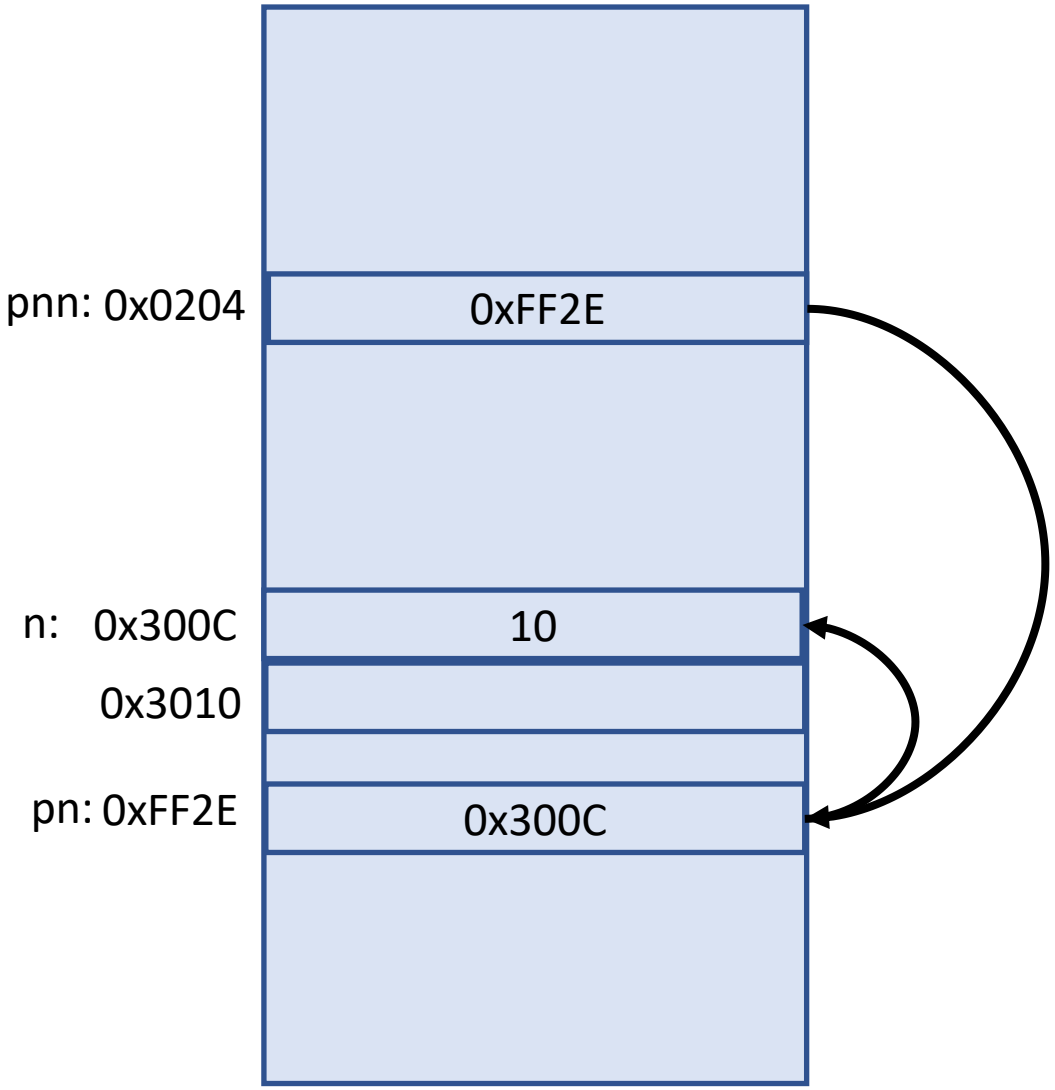
Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn


// print address of n (0x300C)
printf("%p",   &n);
printf("%p",   pn);
printf("%p",   *ppn);
```



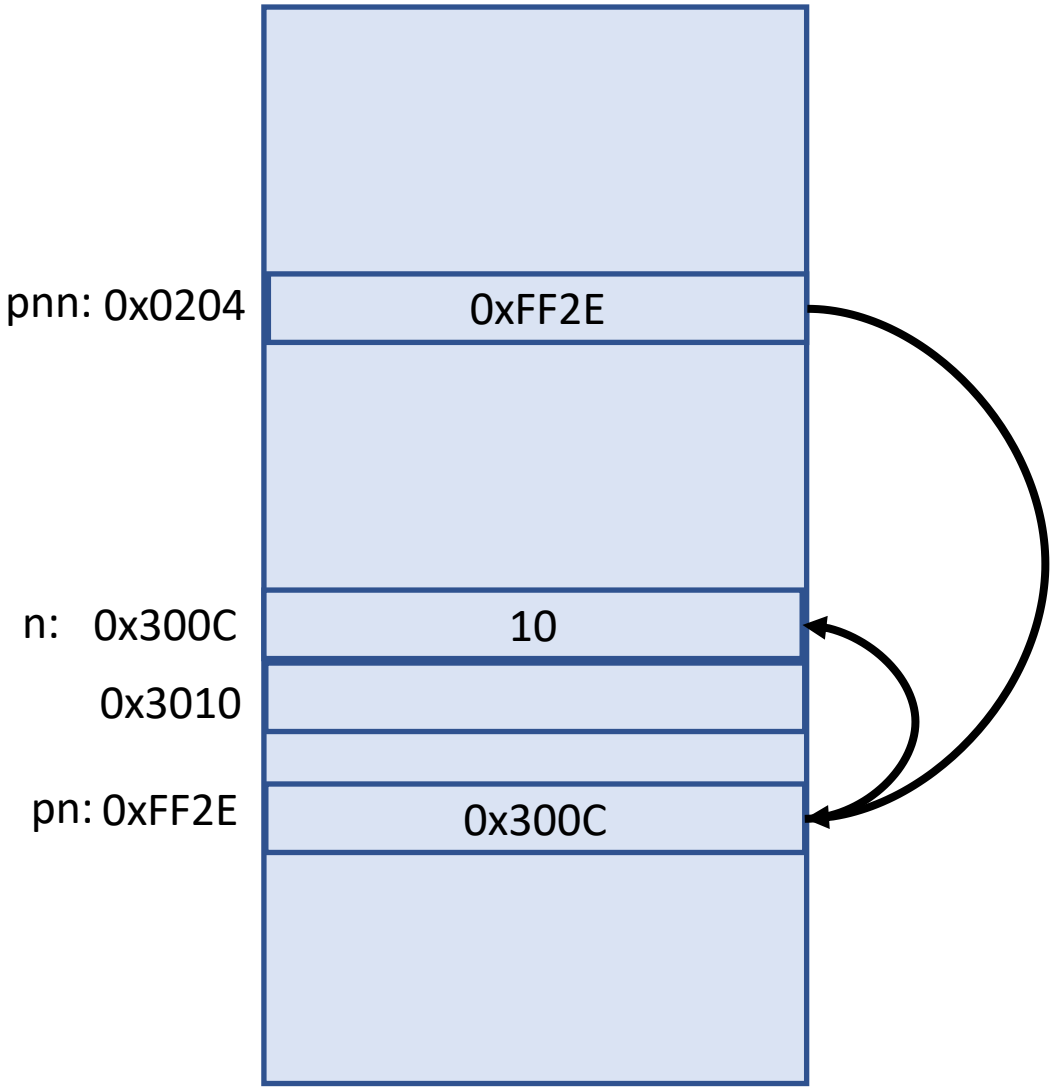
Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn


// print address of pn (0xFF2E)
printf("%p",   ?n);  // Not possible
printf("%p",   &pn);
printf("%p",   ppn);
```



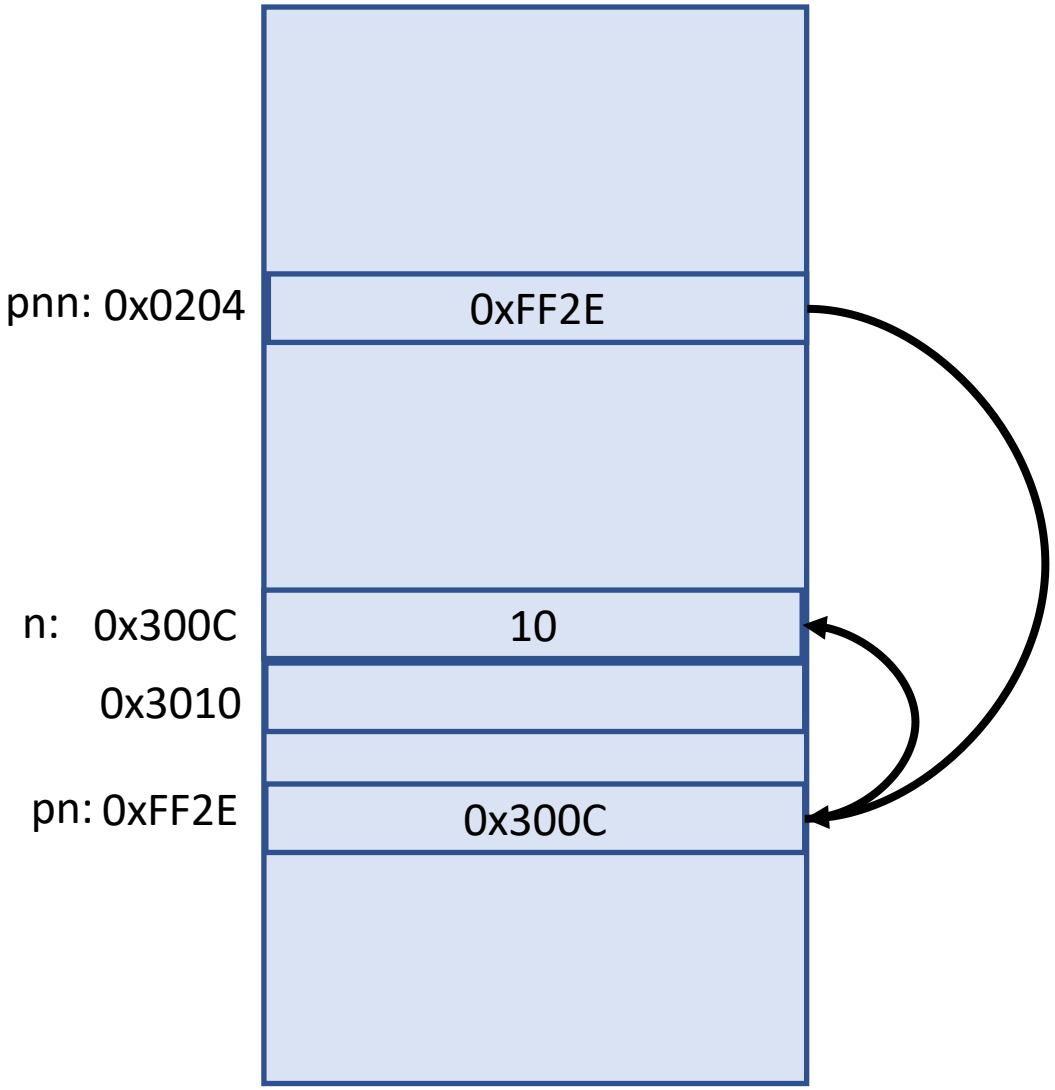
Pointers to Pointers

Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn


// print address of ppn (0x0204)
printf("%p",   ?n);  // Not possible
printf("%p",   ?pn); // Not possible
printf("%p",   &ppn);
```



Pointers to Pointers

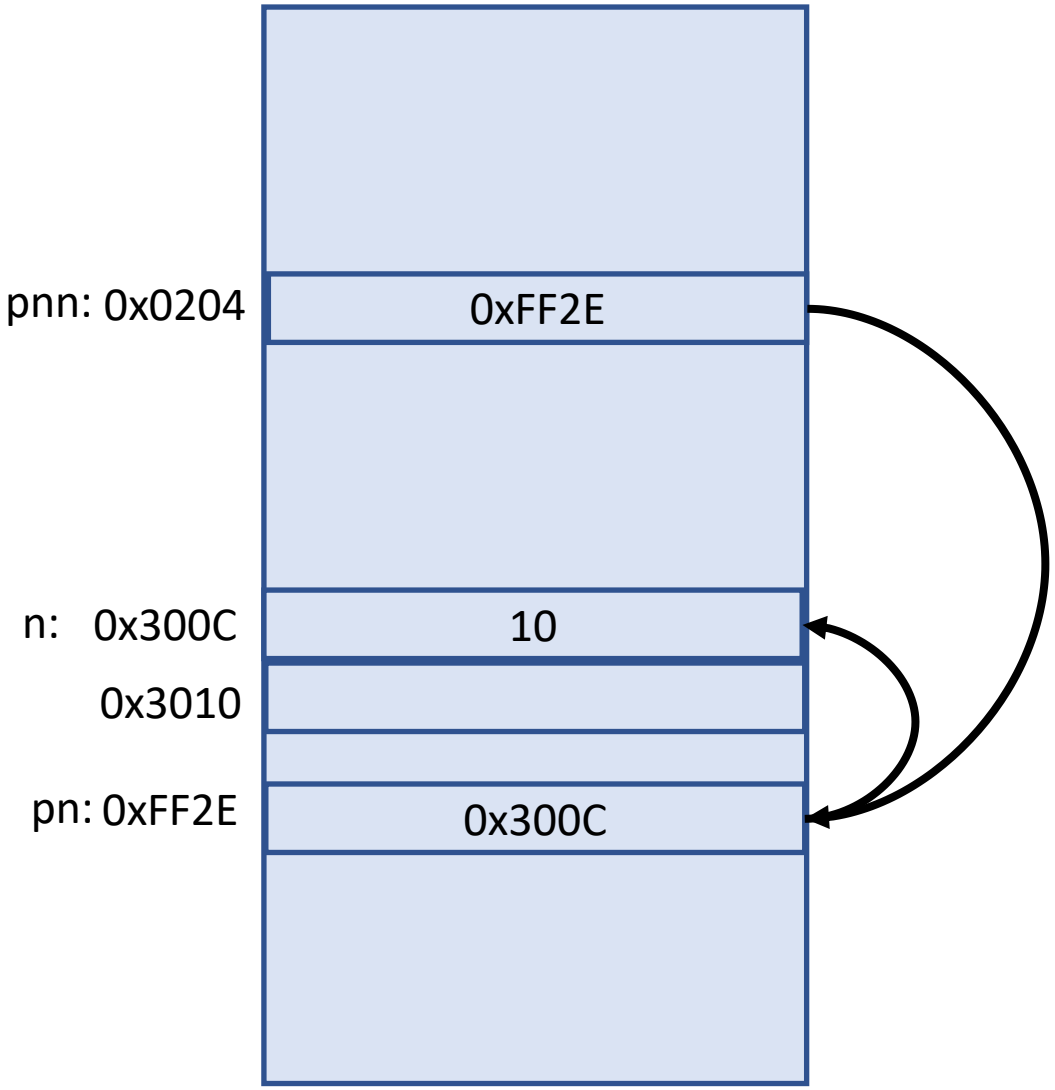
Pointer Review

```
int n = 10;
int *pn = &n;

int **ppn = &pn
```

```
// print address of ppn (0x0204)
printf("%p",   ?n);  // Not possible
printf("%p",   ?pn); // Not possible
printf("%p",   &ppn);
```

No practical limit to how many levels of indirection we can use.



Pointers to Pointers

Why would I ever want to to this?

Pointers to Pointers

Why would I ever want to to this?

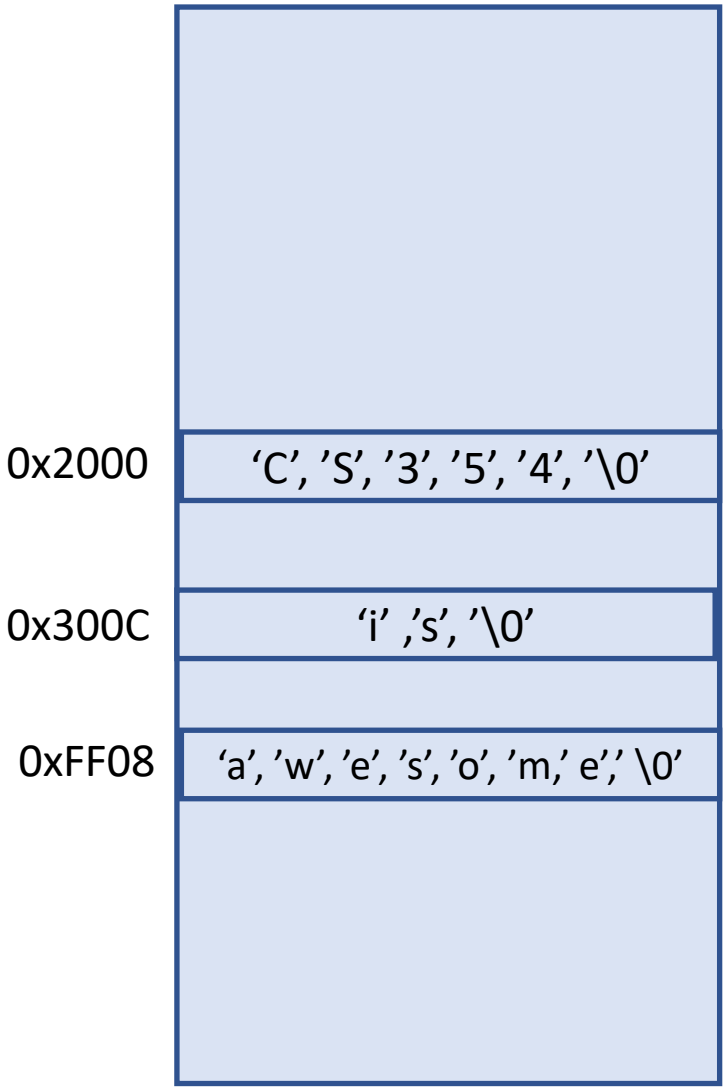
Store an array of strings
Strings are character arrays

Pointers to Pointers

Why would I ever want to to this?

Store an array of strings
Strings are character arrays

Consider storing these three words
“CS354”
“is”
“awesome”



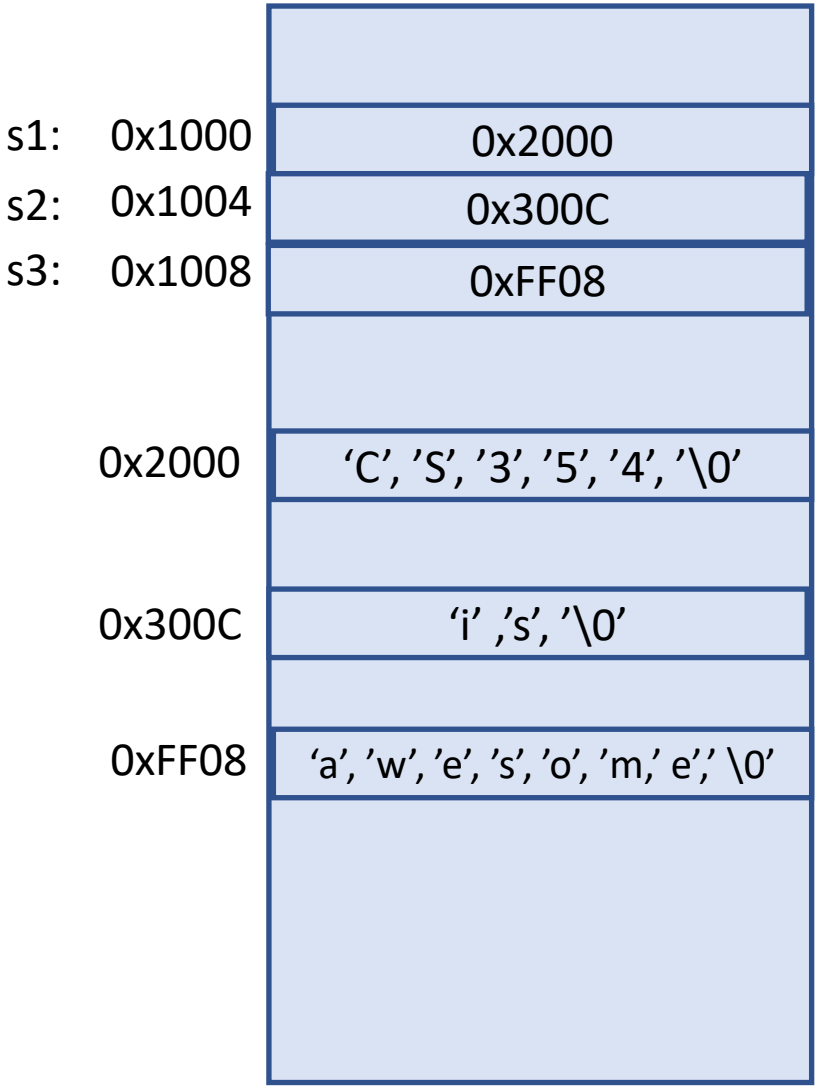
Pointers to Pointers

Why would I ever want to to this?

Store an array of strings
Strings are character arrays

Consider storing these three words
“CS354”
“is”
“awesome”

Declare pointer to pointer as
char **ppc



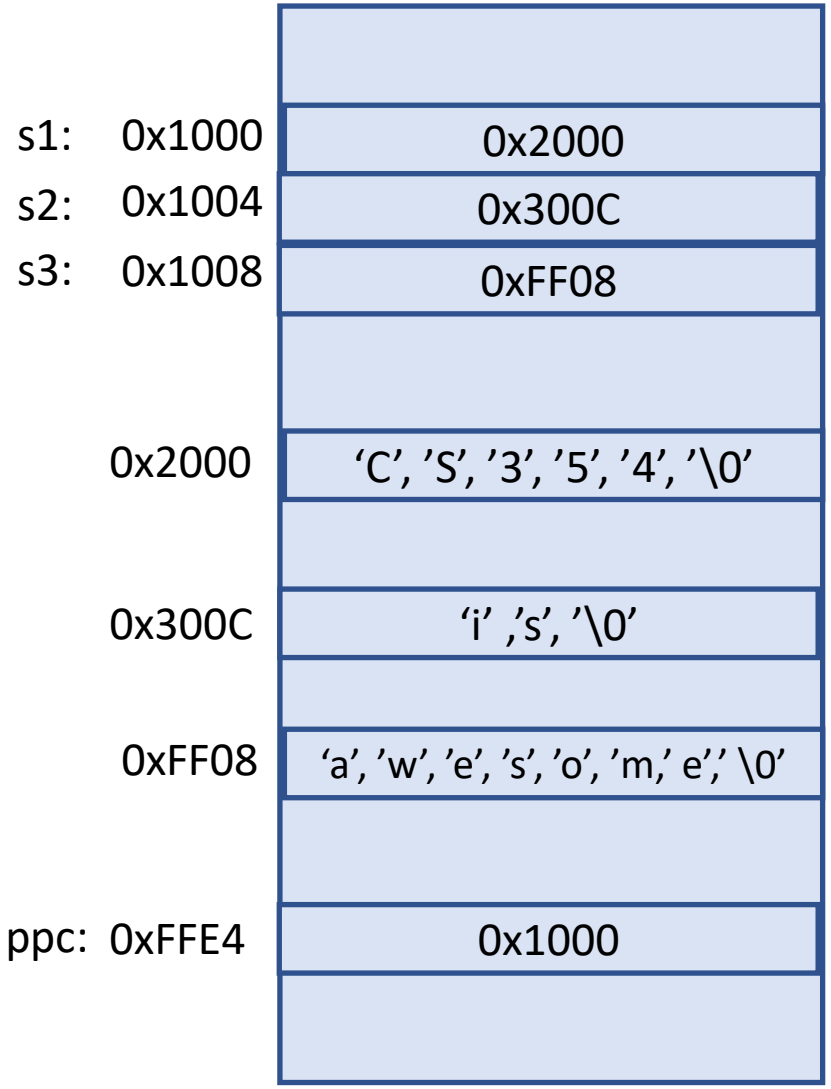
Pointers to Pointers

Why would I ever want to to this?

Store an array of strings
Strings are character arrays

Consider storing these three words
“CS354”
“is”
“awesome”

Declare pointer to pointer as
char **ppc



CS 354
Machine Organization and Programming

Lecture 06C

Structures

Michael Doescher

Summer 2020

Structures

STUDENT
    name
    id

Structures

STUDENT
name
id

```
struct STUDENT {
    char *name;
    int id;
};
```

Structures

STUDENT
name
id

```
struct STUDENT {
    char *name;
    int id;
};
```

Structures

STUDENT
name
id

```
struct STUDENT {
    char *name;
    int id;
};

struct STUDENT s1;
s1.name = "Mike"
s1.id = 123456;
```