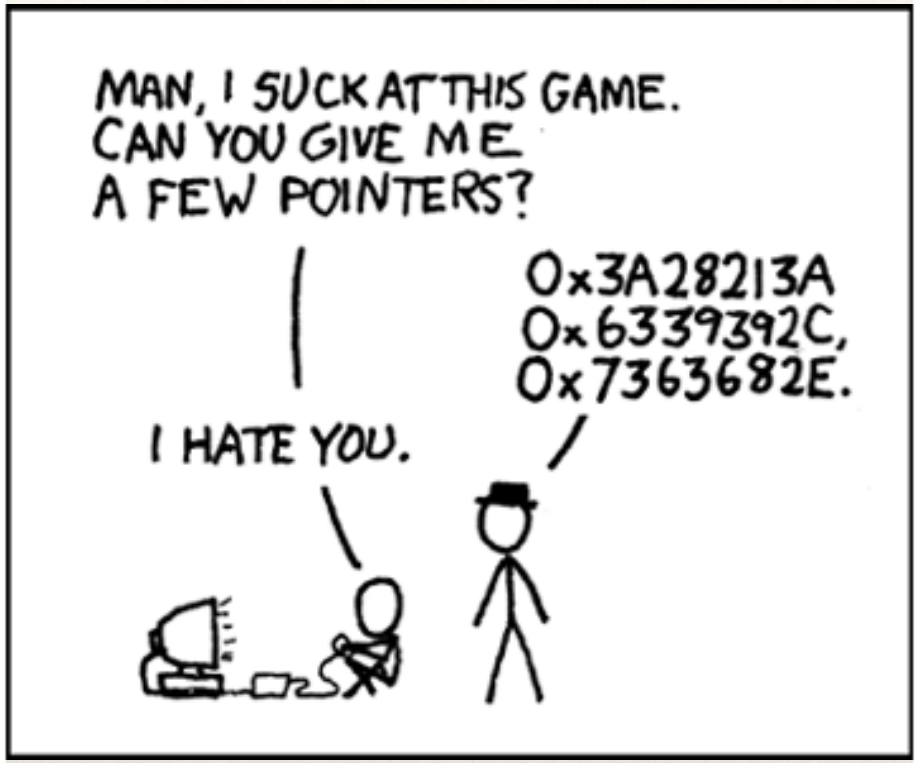
Free the malloc()s

Continuing our Look at Memory

Allocating, using and releasing memory…



https://imgs.xkcd.com/comics/pointers.png

Thinking about 2D Arrays

- You can think of the memory structure as a 2D array of characters or as a 1D array of strings

	0	1	2	3	4
0	N	a	m	e	\0
1	N	a	m	e	\0
2	N	a	m	e	\0
3	N	a	m	e	\0

sptr[i][j]
sptr[3][3] = 'e'
sptr[3] = "Name"

Dynamically Allocating a 2D Array

- Create a 10 X 15 array of integers.

```
int **iptr;
int i, j;

iptr = malloc ( sizeof(int *) * 10 );

for ( i=0; i<10; i++ ) {
   iptr[i] = malloc ( sizeof(int) * 15 );
}

for ( i=0; i<10; i++ ) {
   for ( j=0; j<15; j++ ) {
      iptr[i][j] = i + j;
   }
}
```

Dynamically Allocating a 2D Array

```
int **iptr;
int i, j;

iptr = malloc ( sizeof(int *) * 10 );

for ( i=0; i<10; i++ ) {
   iptr[i] = malloc ( sizeof(int) * 15 );
}

for ( i=0; i<10; i++ ) {
   for ( j=0; j<15; j++ ) {
      iptr[i][j] = i + j;
      printf ( "%02d ", iptr[i][j] );
   }
   printf ("\n" );
}
```

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
03 04 05 06 07 08 09 10 11 12 13 14 15 16 17
04 05 06 07 08 09 10 11 12 13 14 15 16 17 18
05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
07 08 09 10 11 12 13 14 15 16 17 18 19 20 21
08 09 10 11 12 13 14 15 16 17 18 19 20 21 22
09 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Releasing Dynamically Allocated Memory

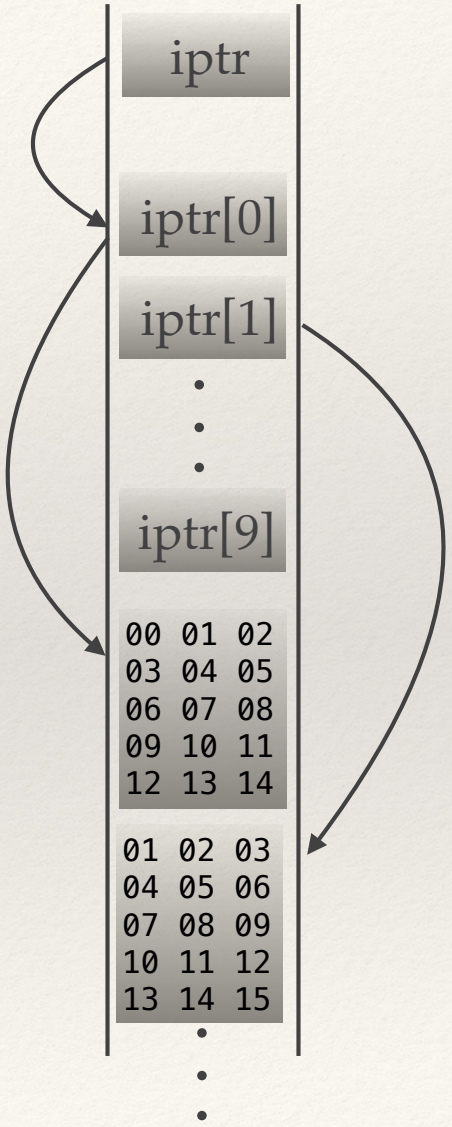
- The free() function returns dynamically allocated memory to the system.
- As soon as you do not need the memory, it should be freed.
- Call free() with the pointer as a parameter.

```
float *fptr;

fptr = malloc ( sizeof(float) * 3 );
… use fptr and the allocated memory …

free(fptr);
```

free()

- Every malloc() requires a free().
- Arrays of pointers require a loop of free()'s.

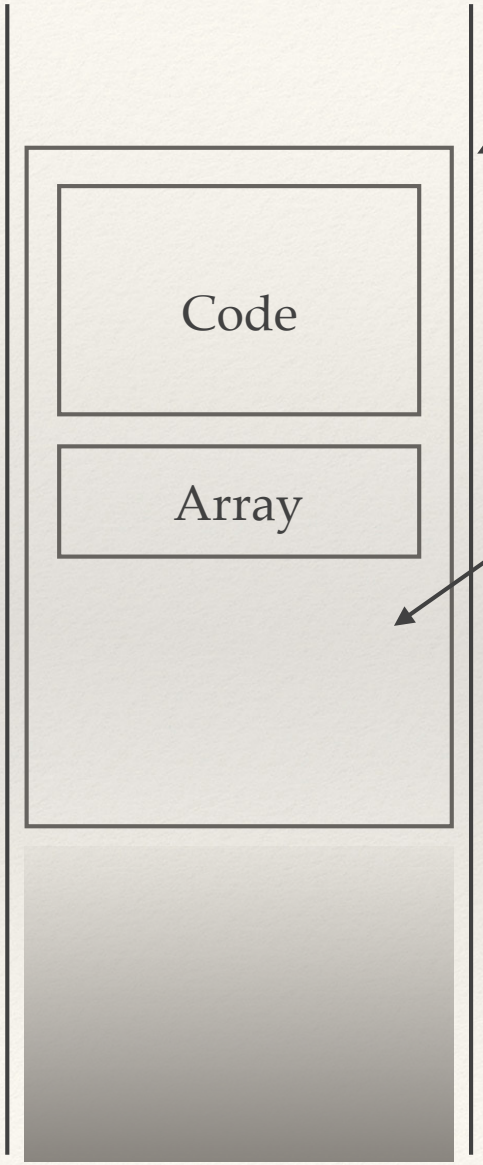
```
for ( i=0; i<10; i++ ) {
  free ( iptr[i] );
}
free ( iptr );
```



Reading or Writing Past the End of an Array

- In C, nothing stops you from using an index to an array that is larger than the size of the array.
  - float numbers[5];
  - numbers[10] = 0;
- But what happens?
  - No **apparent** problem
  - Data corruption
  - Program crash

Reading or Writing Past the End of an Array



Running program

- writing to the executable part will have unpredictable results

Unused space owned by your program

- this space can be read or written to by your code
- if it is not being used then it might seem like there is not problem
- but eventually it will be used and then there will be a problem

Memory not owned by your program

- if you write into here - program crash

Pointer Arithmetic

- A pointer can be used to step through memory using the ++ and - - operators.

```
char *name, *ptr;
int i;
name = malloc ( sizeof(char) * 10 );
strpy ( name, “abcdefg” );
ptr = name;
for ( i=0; i<4; i++ ) {
   printf ( “%c\n”, *ptr );
   ptr++;
}
```

Create string

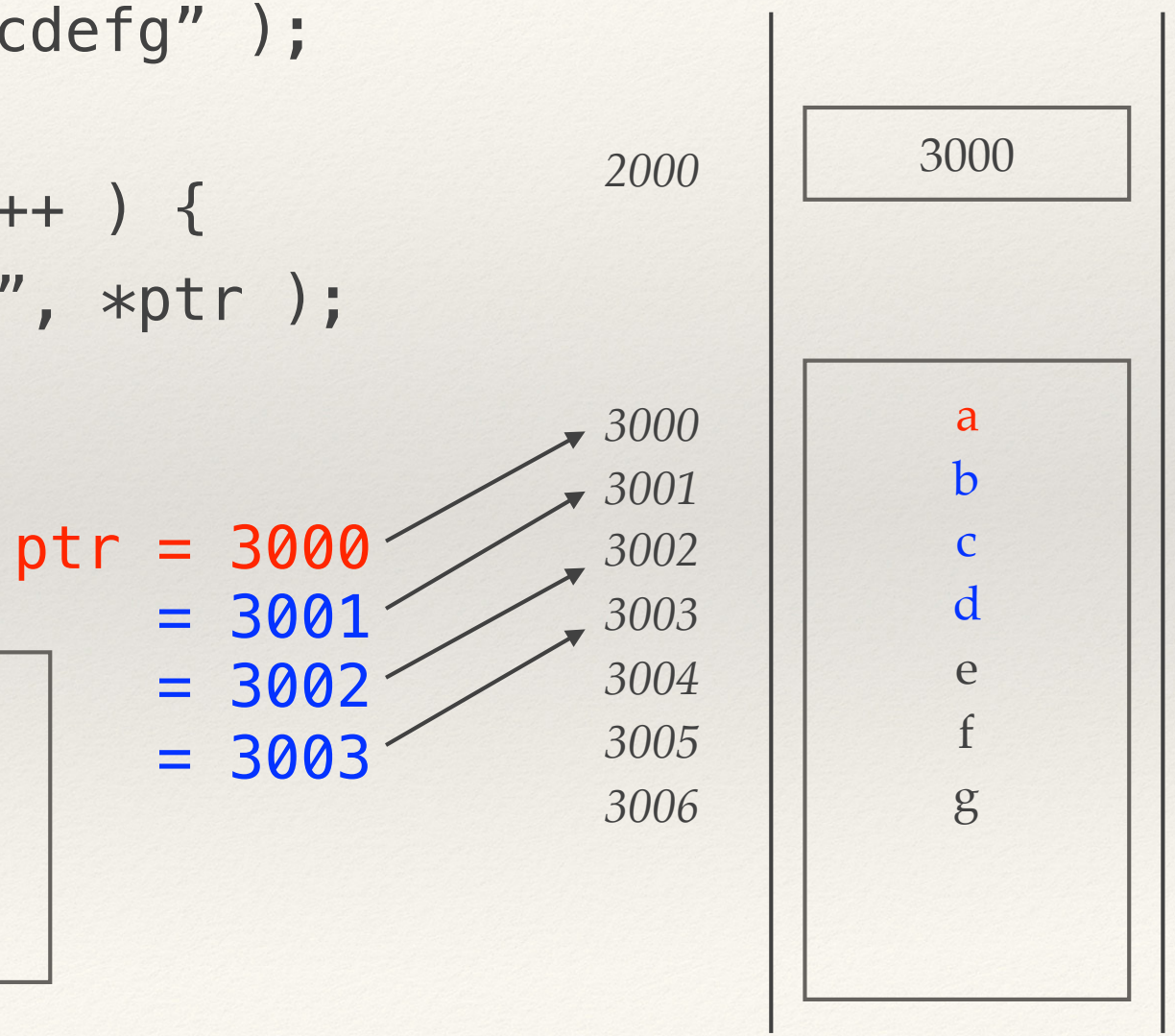
Set ptr to start of the string

Print character that ptr is pointing at

Advance ptr to the next address

Pointer Arithmetic

```
name = malloc ( sizeof(char) * 10 );
strpy ( name, “abcdefg” );
ptr = name;
for ( i=0; i<4; i++ ) {
    printf ( “%c\n”, *ptr );
    ptr++;
}
```



- This works with any type!
- `ptr` knows how big a step to take with ++ because of the **type** of the pointer

Structures in C

- Structures in C can be static or dynamically created.
- **Static** structures are described with the `struct` definition and are accessed with the `.` operator.
- You treat it like the variable but with the `struct` name in front.

```
struct data {
    int count;
    float total;
};
```

Structure Type Definition

- This describes the structure but does **not** create it.
- In our example, the **type** is `struct data`

Static Structure in C

```
int main ( )
{
   struct data {
      int count;
      float total;
   };
   struct data aStructure;
   aStructure.count = 10;
   aStructure.total = 7.5;

   printf ( “%d %f\n”, aStructure.count, aStructure.total );
}
```

this defines the structure

this creates the structure

```
$ gcc -o structExample1 structExample1.c
$ ./structExample1
10 7.500000
```

Dynamically Allocated Structures

- Dynamically allocated structures are created using `malloc` and a pointer.
- Elements in a dynamically allocated structure are accessed through the `->` operator (not the `.` operator).

Dynamically Allocated Structures

```
int main ( )
{
   struct data {
      int count;
      float total;
   }
   struct data *aPtr;
   aPtr = malloc ( sizeof(struct data) );

   (*aPtr).count = 10;
   (*aPtr).total = 3.5;

   printf ( "%d %f\n", aPtr->count, aPtr->total );
}
```

pointer to structure of type `struct data`

allocate enough memory to hold a structure of type `data`

pointer references element in structure

Dynamically Allocated Structures

```
int main ( )
{
   struct data {
      int count;
      float total;
   }
   struct data *aPtr;
   aPtr = malloc ( sizeof(struct data) );

   aPtr -> count = 10;
   aPtr -> total = 3.5;

   printf ( "%d %f\n", aPtr->count, aPtr->total );
}
```

pointer to structure of type `struct data`

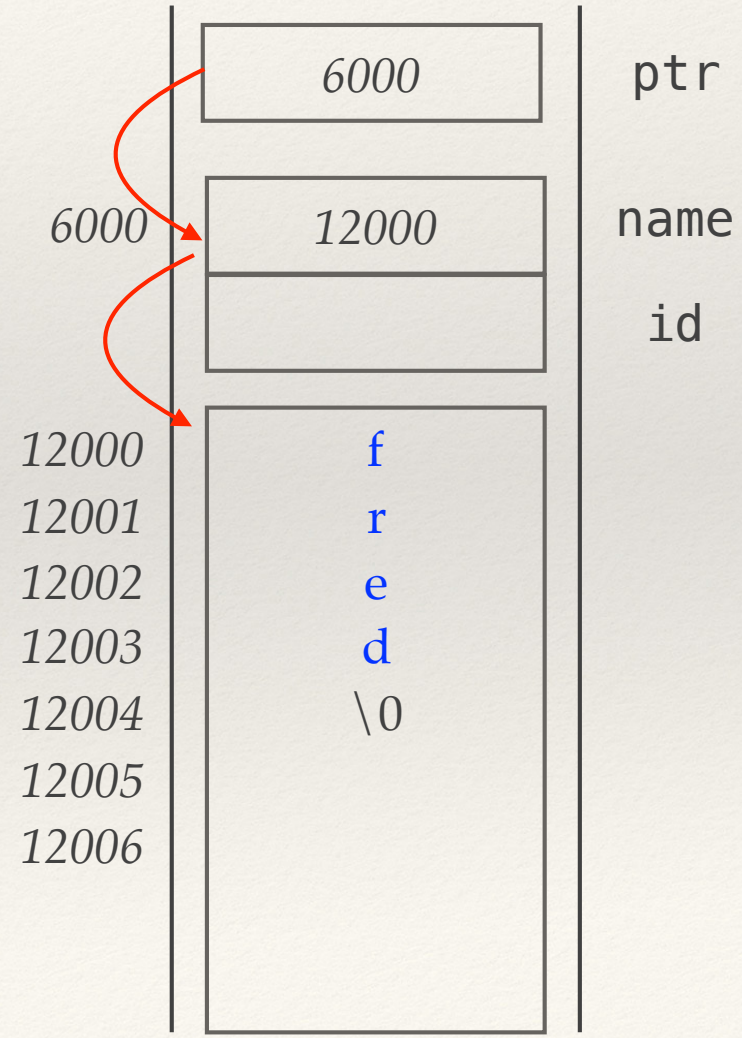
allocate enough memory to hold a structure of type `data`

pointer references element in structure

Dynamically Allocated Structures

- The contents of a structure can be of any valid C type.

```
struct data {
   char *name;
   int id;
};
struct data *ptr;

                                        allocate structure
ptr = malloc ( sizeof(struct data) );

                                          allocate string
ptr->name = malloc ( sizeof(char)*20 );


strcpy ( ptr->name, "fred" );
```

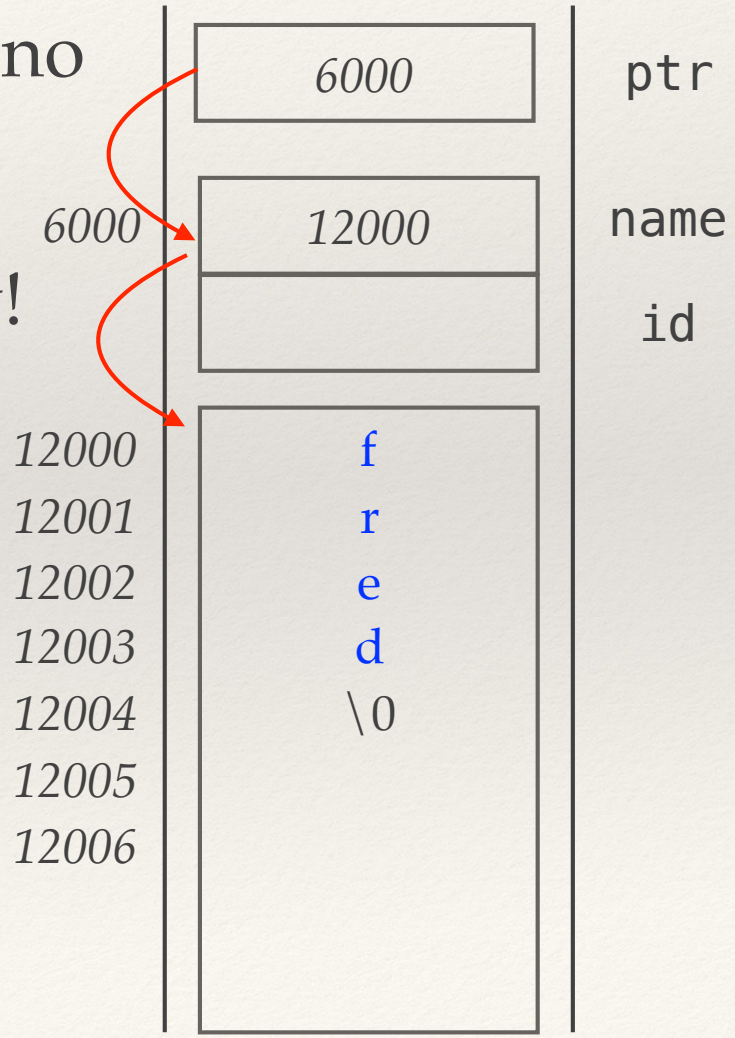


Freeing Memory

- Memory that has been allocated should be returned to the system when you no longer need it.
- Do not ignore the freeing of memory!
- Freeing the structure `data`:

```
free ( ptr );
```

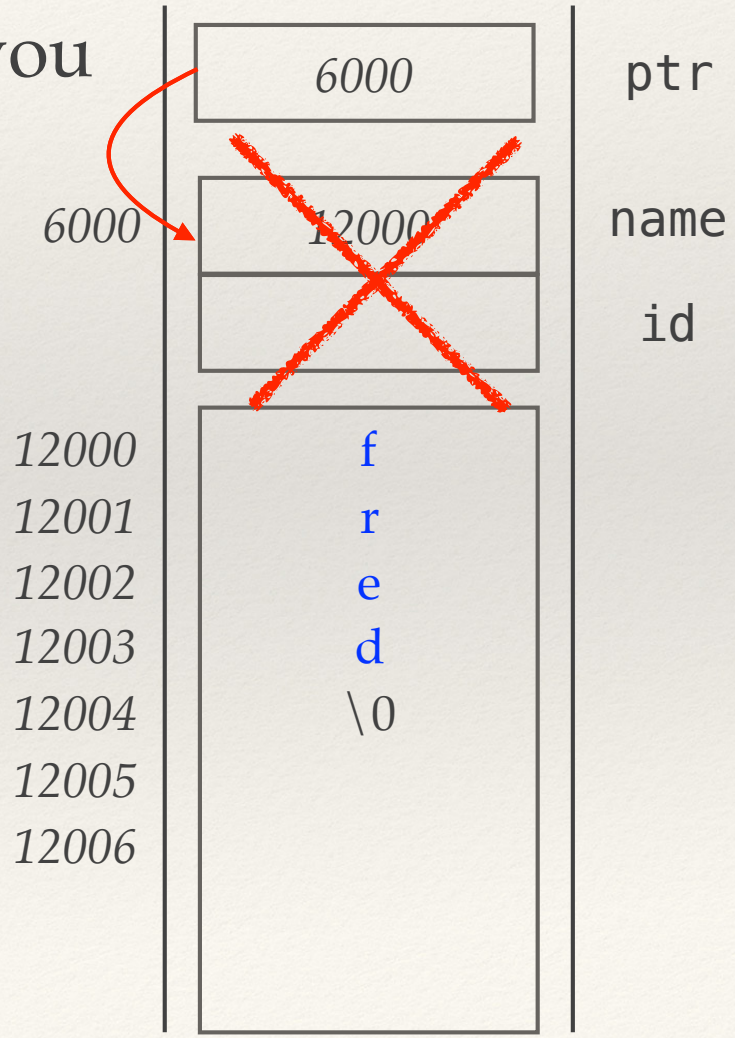
- But this will create a problem!
  - **Why???**



Freeing Memory

- If you free `ptr` first then the pointer to the string name will disappear and you will not be able to free `ptr->name`.
- Free `ptr->name` first.

```
free ( ptr->name );
free ( ptr );
```



NULL Pointers

- `NULL` pointers are used to show that no memory is allocated on that pointer.
- `malloc()` returns `NULL` if it fails to allocate memory.
- It is good to test if a pointer is `NULL` after a `malloc()`.

```
ptr = malloc ( sizeof(char) * 10 );
if ( ptr == NULL ) {
   … error handling …
}
```

NULL Pointers

- It is also good policy to set pointers to `NULL` once they have been freed.

```
free ( ptr );
ptr = NULL;
```

- After the `free()` the pointer still points at the memory location.

Back to Arrays

`array[][]` has multiple meanings

- can be used to reference elements

```
x = a[3][5]
```

- can be used to declare the array in local memory

```
int a[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}
```

- can be used as a parameter declaration to pass array as a variable

passing locally defined arrays

```
int foo(int a[][4]){
    /* your code here */
}
```

passing dynamically defined arrays

```
int foo(int a[][]){
    /* your code here */
}
```

Back to Arrays

Different compiler directives set up different arrays

- a locally defined array — 2 * 3 * sizeof(double) = 2 * 3 * 8 = 48 bytes local

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

- an array of 2 pointers

```
double *array[2]
```

2 * sizeof(double *) = 2 * 4 = 8 bytes local

2 * (3 * sizeof(double)) = 2 * 3 * 8 = 48 bytes dynamic

- a pointer to a pointer

```
double **array
```

1 * sizeof(double **) = 1 * 4 = 4 bytes local

2 * sizeof(double *) + 2 * (3 * sizeof(double))
= 2 * 4 + 2 * 3 * 8 = 8 + 48 = 56 bytes dynamic

all of the above assumed to create the same 2 x 3 array

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

`array[1][2]` actually converted to

```
*( array + 1 * sizeof(double) * 3
         + 2 * sizeof(double) )
```

Local Arrays are not the same as Dynamic Arrays

Create a local array

compiler remembers dimension size from here

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

`array[1][2]` actually converted to

```
*( array + 1 * sizeof(double) * 3
         + 2 * sizeof(double) )
```

Local Arrays are not the same as Dynamic Arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

`array[1][2]` actually converted to

```
*( array + 1 * sizeof(double) * 3
         + 2 * sizeof(double) )
```

compiler remembers element type from here

Local 2^d arrays are **not** the same as dynamic 2^d arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

`array[1][2]` actually converted to

```
*( 2000  + 40 )
```

Local 2^d arrays are **not** the same as dynamic 2^d arrays

Create a local array

```
double array[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}}
```

We think of it as:

	0	1	2
0	1.1	2.2	3.3
1	4.4	5.5	6.6

C thinks of it as:

2000	2008	2016	2024	2032	2040
1.1	2.2	3.3	4.4	5.5	6.6

`array[1][2]` actually converted to

`*( 2040 )` → `6.6`

Back to Arrays

`array[][]` has multiple meanings

- can be used to reference elements

```
x = a[3][5]
```

- can be used to declare the array in local memory

```
int a[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}
```

- can be used as a parameter to pass array as a variable

Compiler need this information to perform the pointer arithmetic when accessed

passing locally defined arrays

```
int foo(int a[][4]){
    /* your code here */
}
```

passing dynamically defined arrays

```
int foo(int a[][]){
    /* your code here */
}
```

Returning Dynamic Arrays

- Creating and returning a 1^d array that uses dynamic memory

```
double * foo( int length ) {
    double *foo_array = malloc(length * sizeof(double));

    /* check if foo_array == NULL, error handling if yes */

    /* initialize foo_array here */

    return foo_array;
}
```

Returning Dynamic Arrays

- Creating and returning a 2^d array that uses dynamic memory

```
double ** bar( int row, int col ) {
    double **bar_array = malloc(row * col * sizeof(double));

    /* check if bar_array == NULL, error handling if yes */

    /* initialize bar_array here */

    return bar_array;
}
```

Returning Dynamic Structures

- Creating and returning a structure that uses dynamic memory

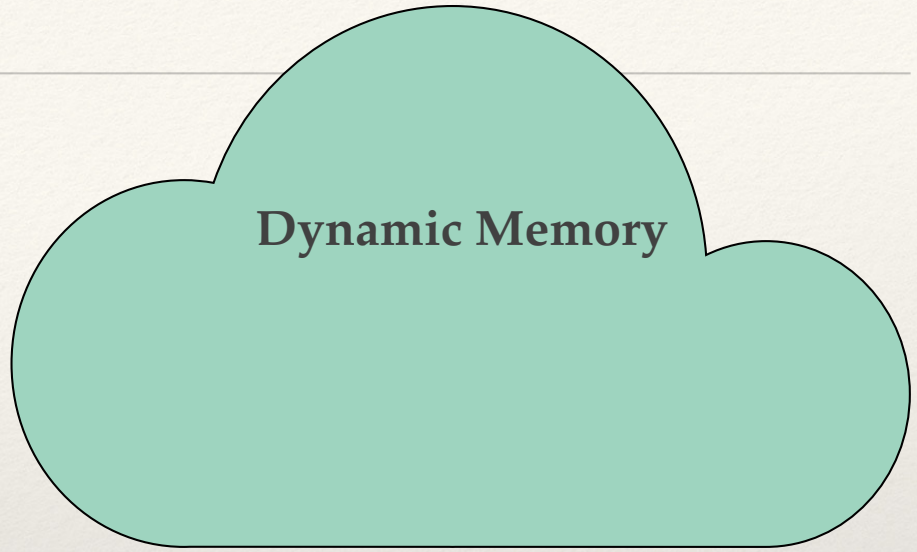
```
struct point {
    double x_coord;
    double y_coord
};

struct point * create_location( double x, double y ) {
    struct point *loc = malloc(2 * sizeof(double));

    /* check if loc == NULL, error handling if yes */

    loc -> x_coord = x;
    loc -> y_coord = y;

    return loc;
}
```

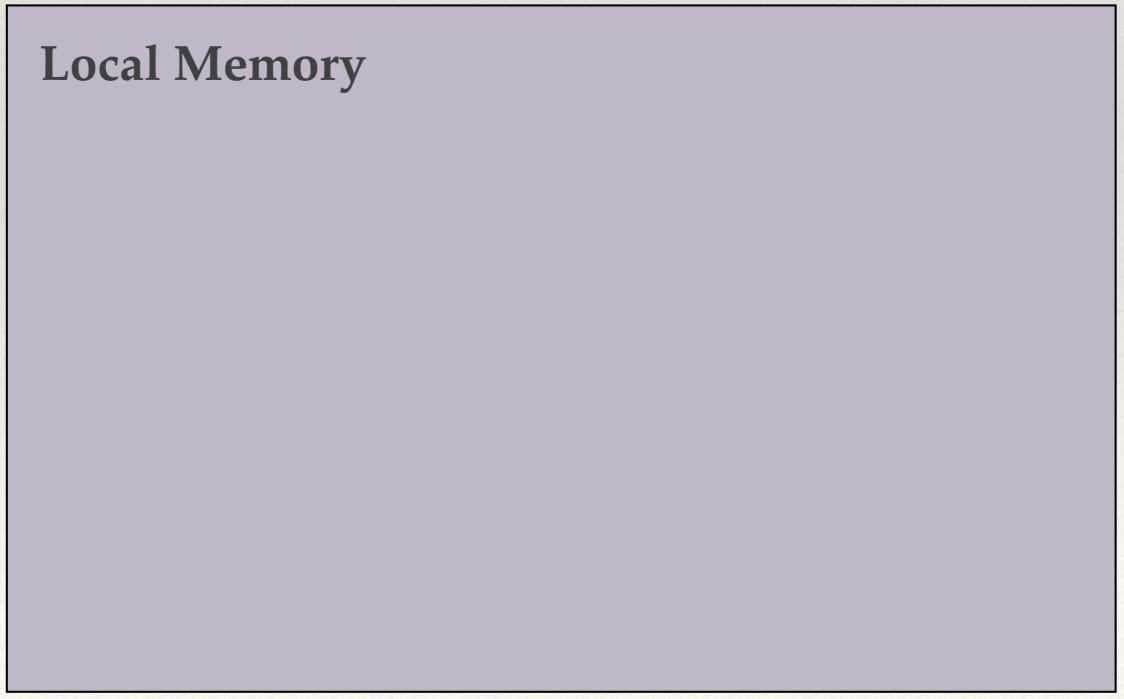
Should be declared in a .h file
so code that use create_location
has access to the struct definition

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

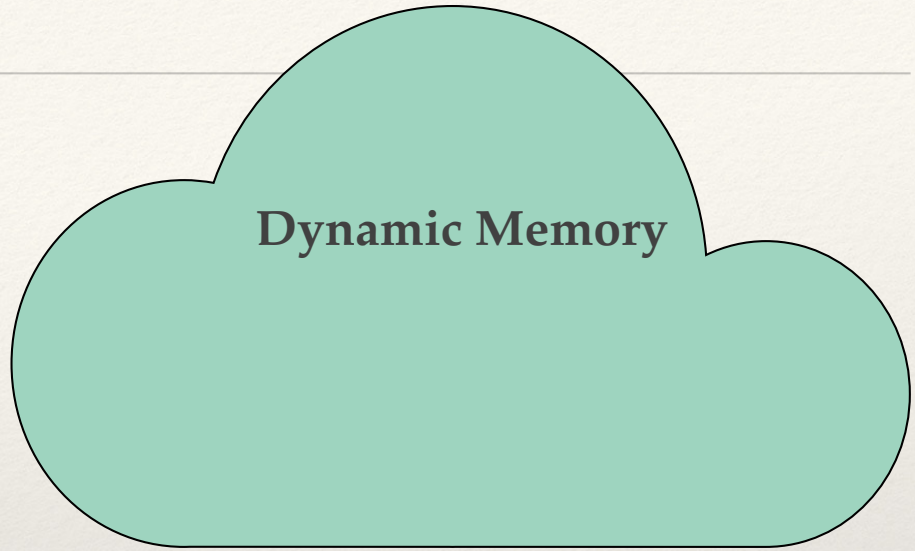




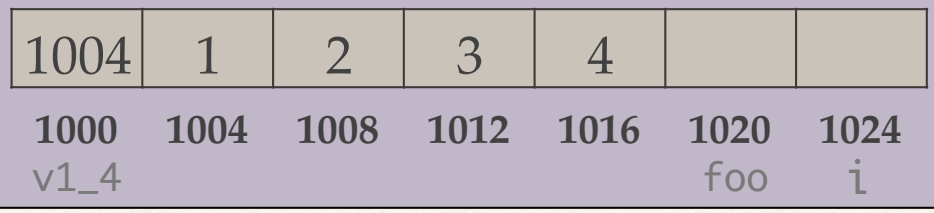
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



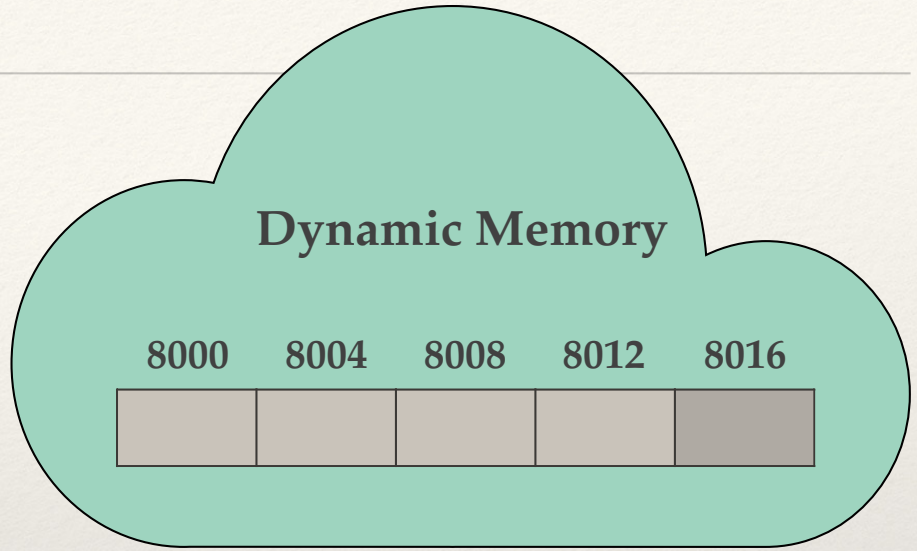
Local Memory



Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



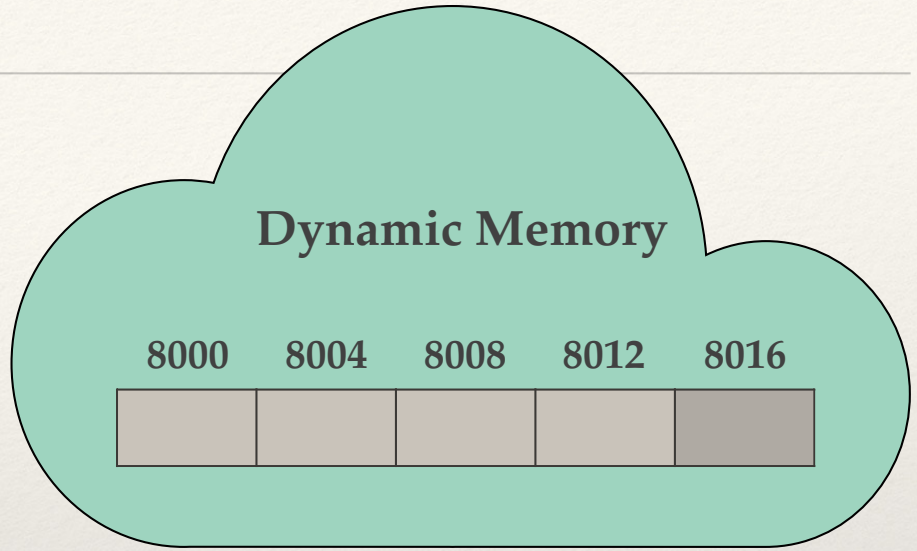
Local Memory

1004	1	2	3	4	8000	
1000	**1004**	**1008**	**1012**	**1016**	**1020**	**1024**
v1_4					foo	i

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



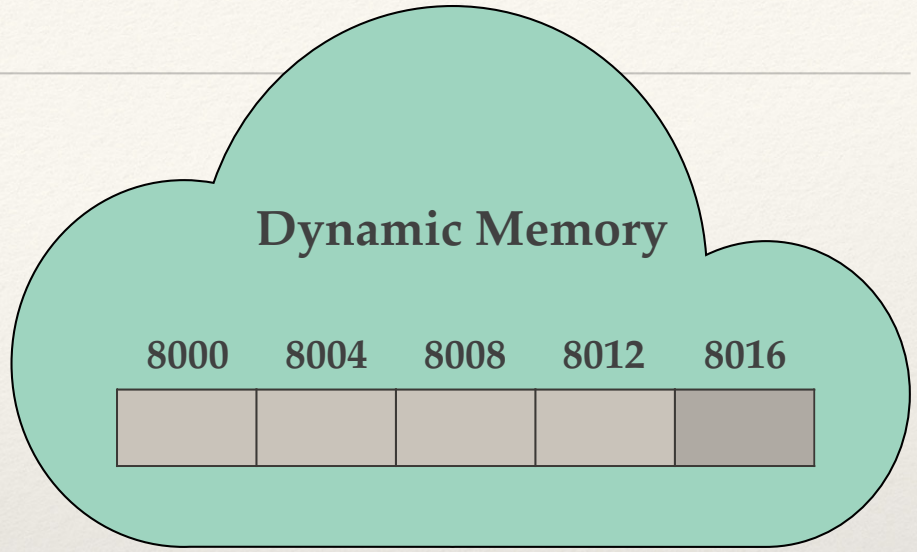
Local Memory

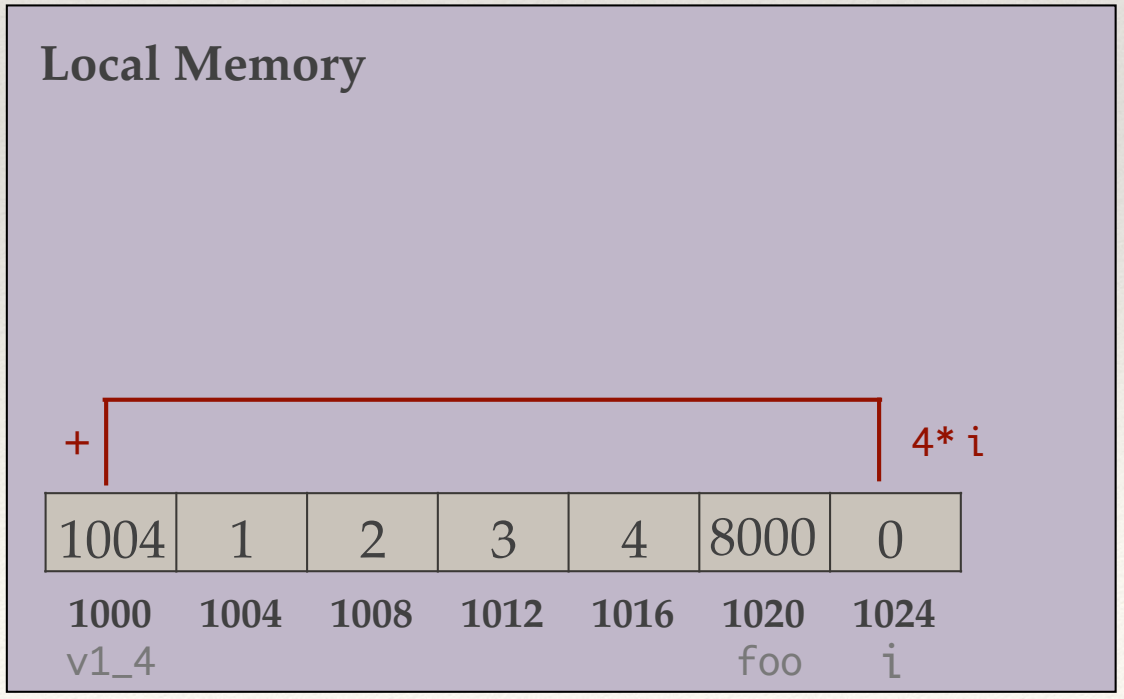
1004	1	2	3	4	8000	0
1000	1004	1008	1012	1016	1020	1024
v1_4					foo	i

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

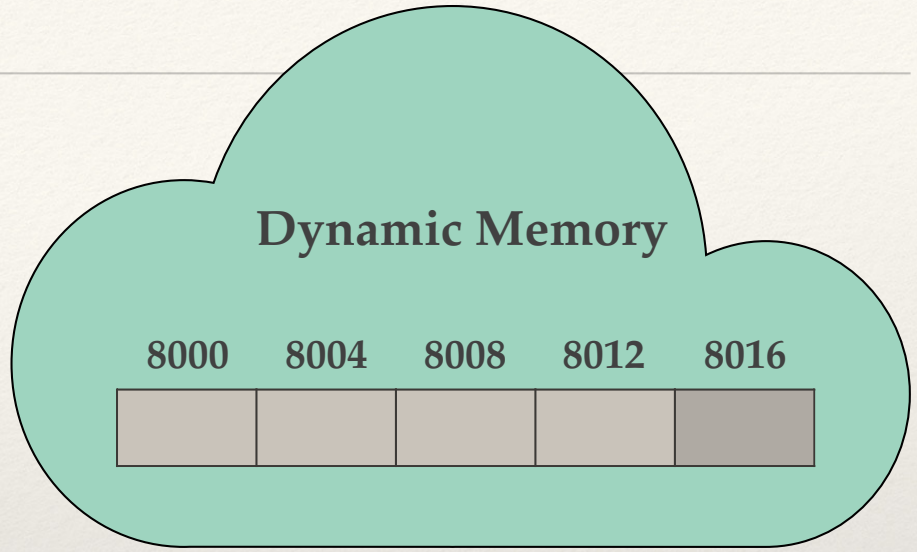


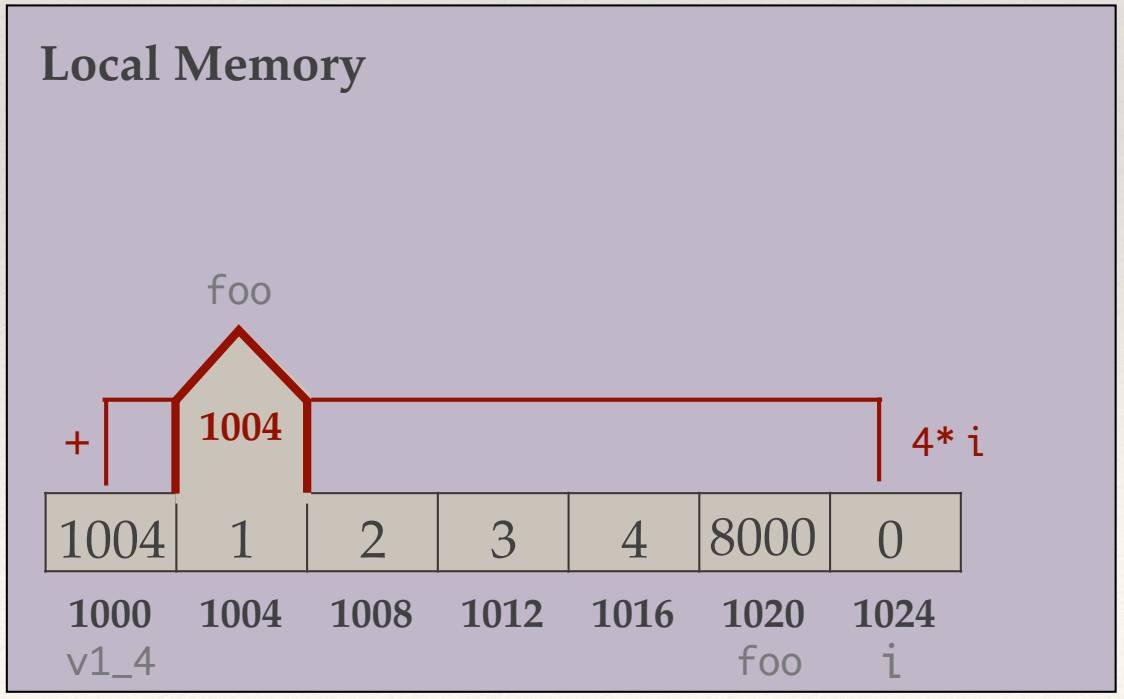


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

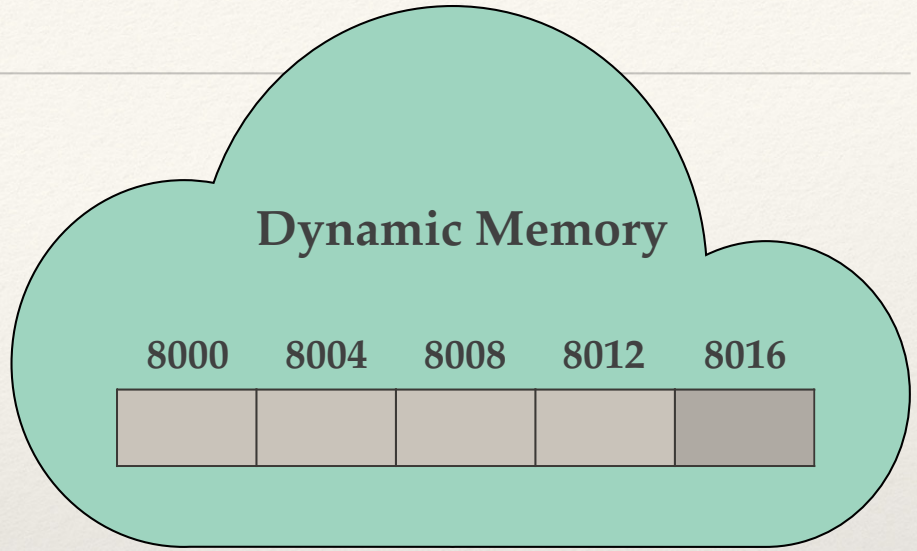


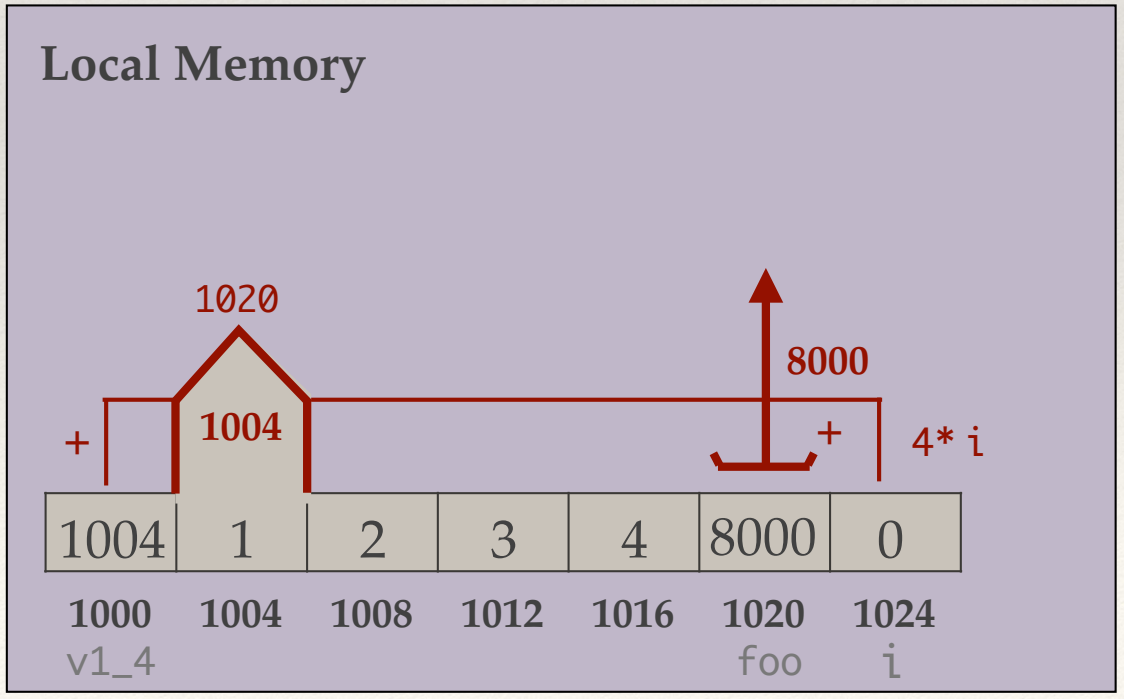


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

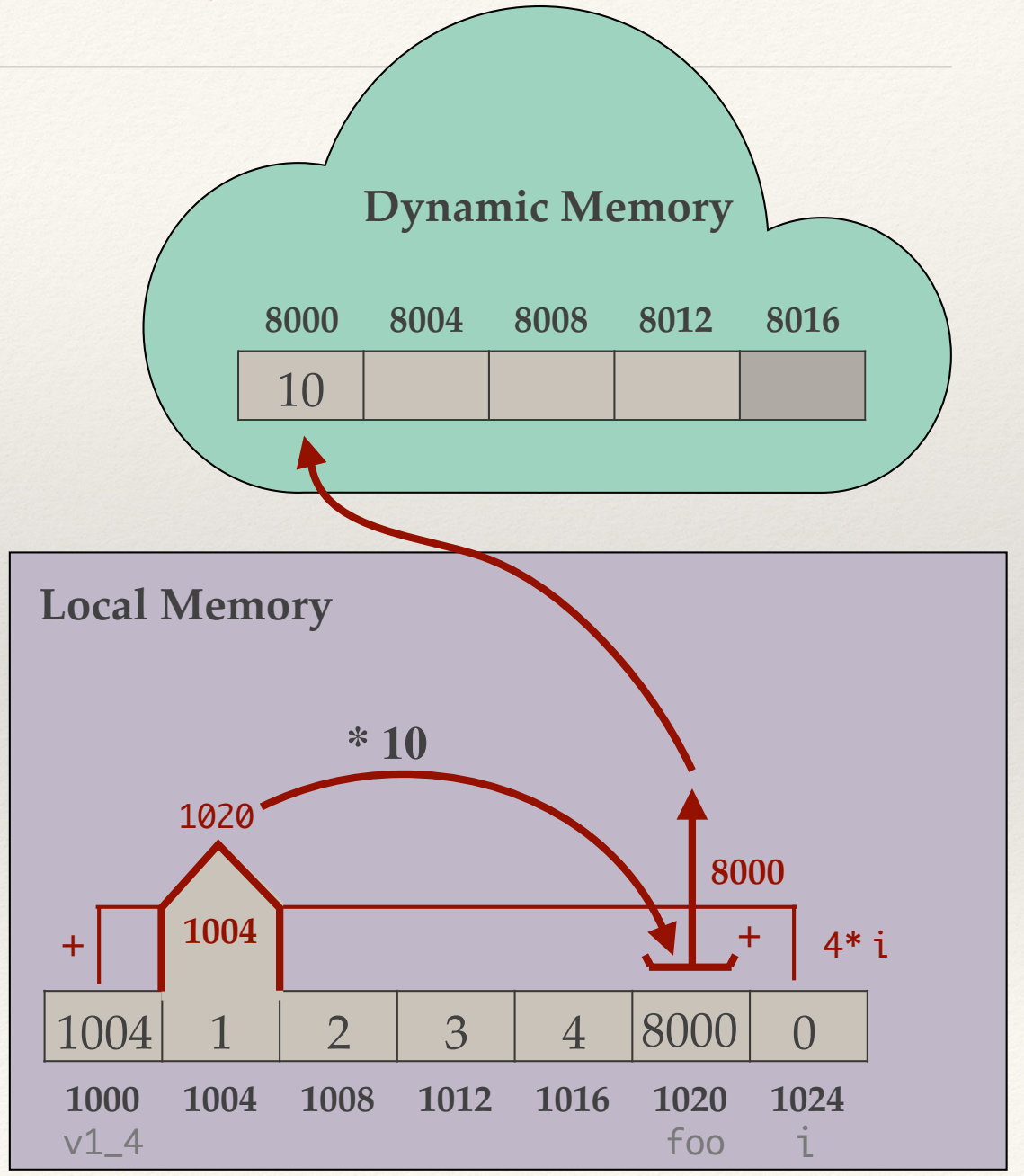




Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

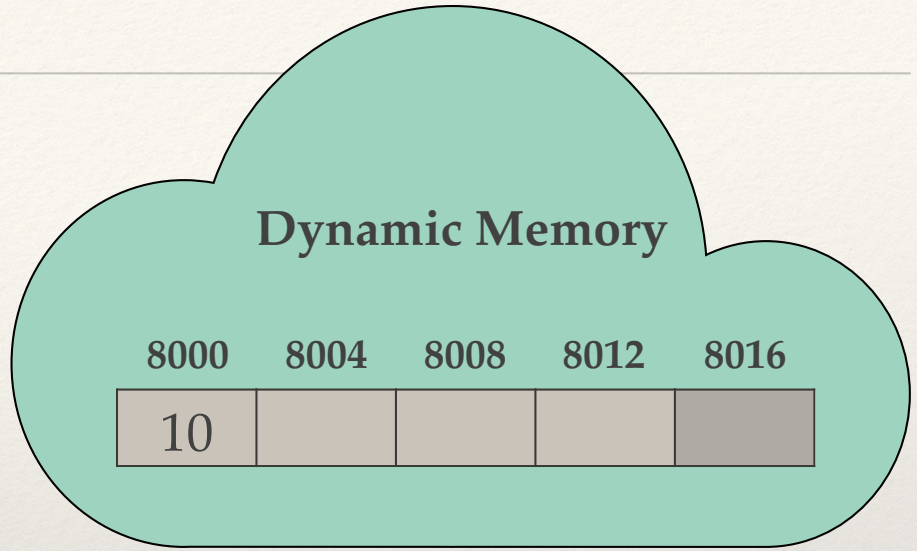
```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

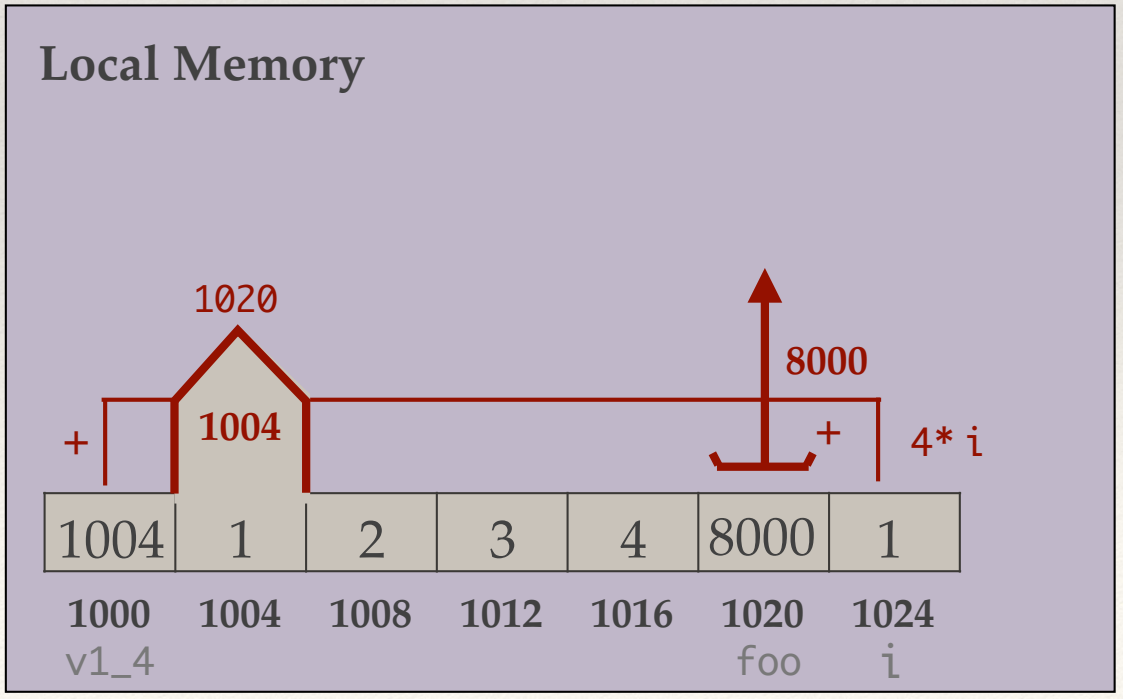


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

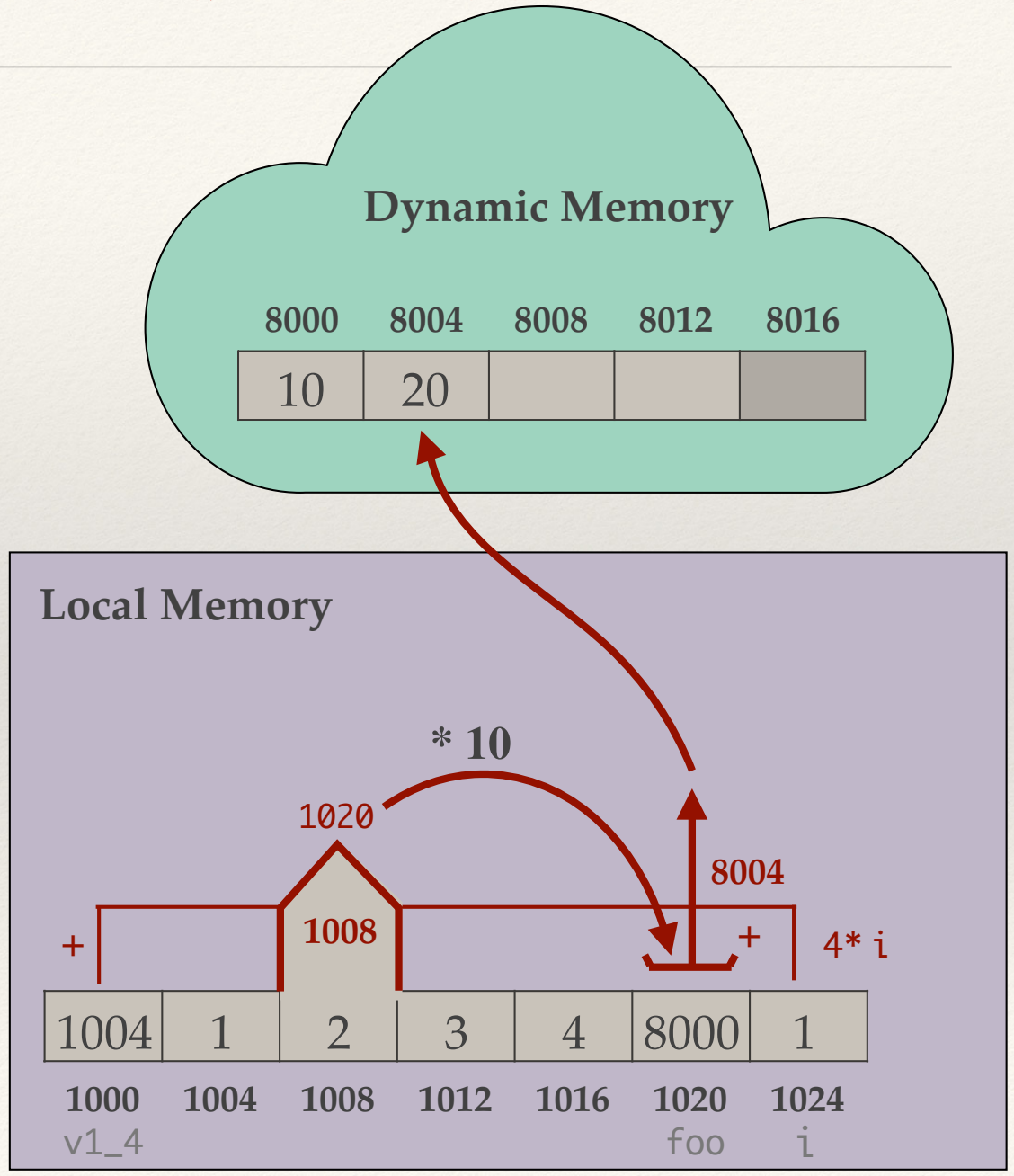




Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

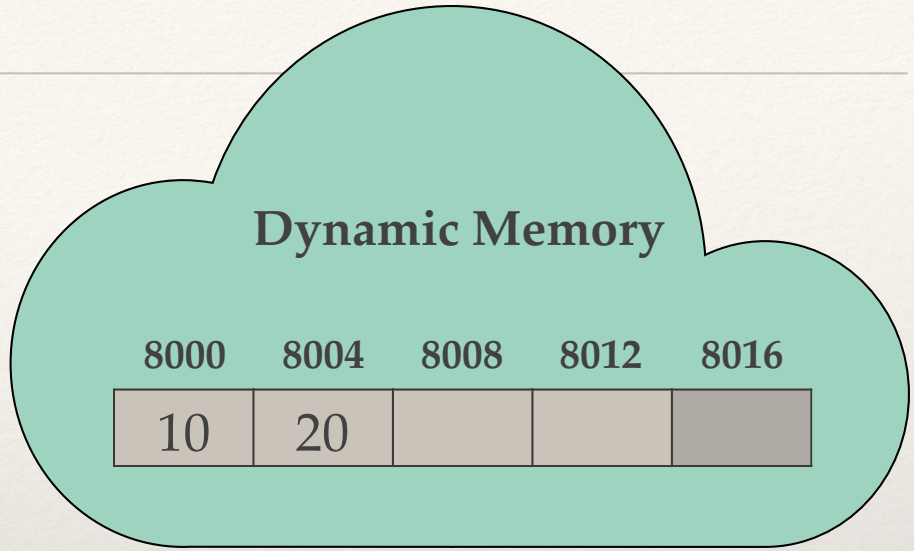
```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

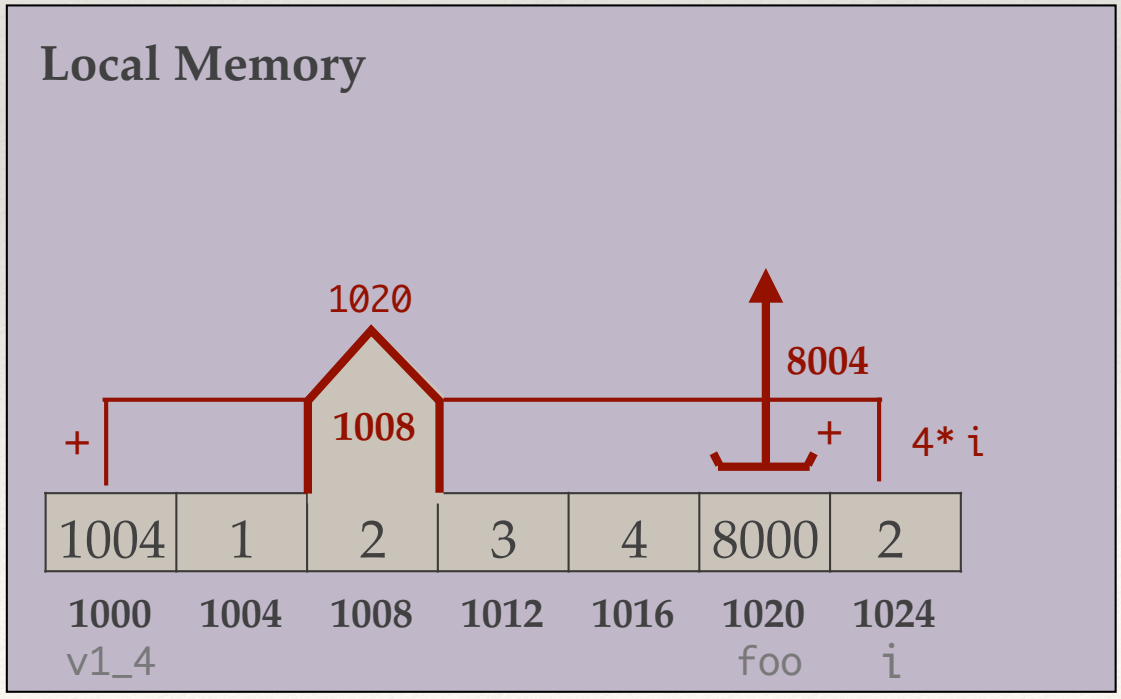


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

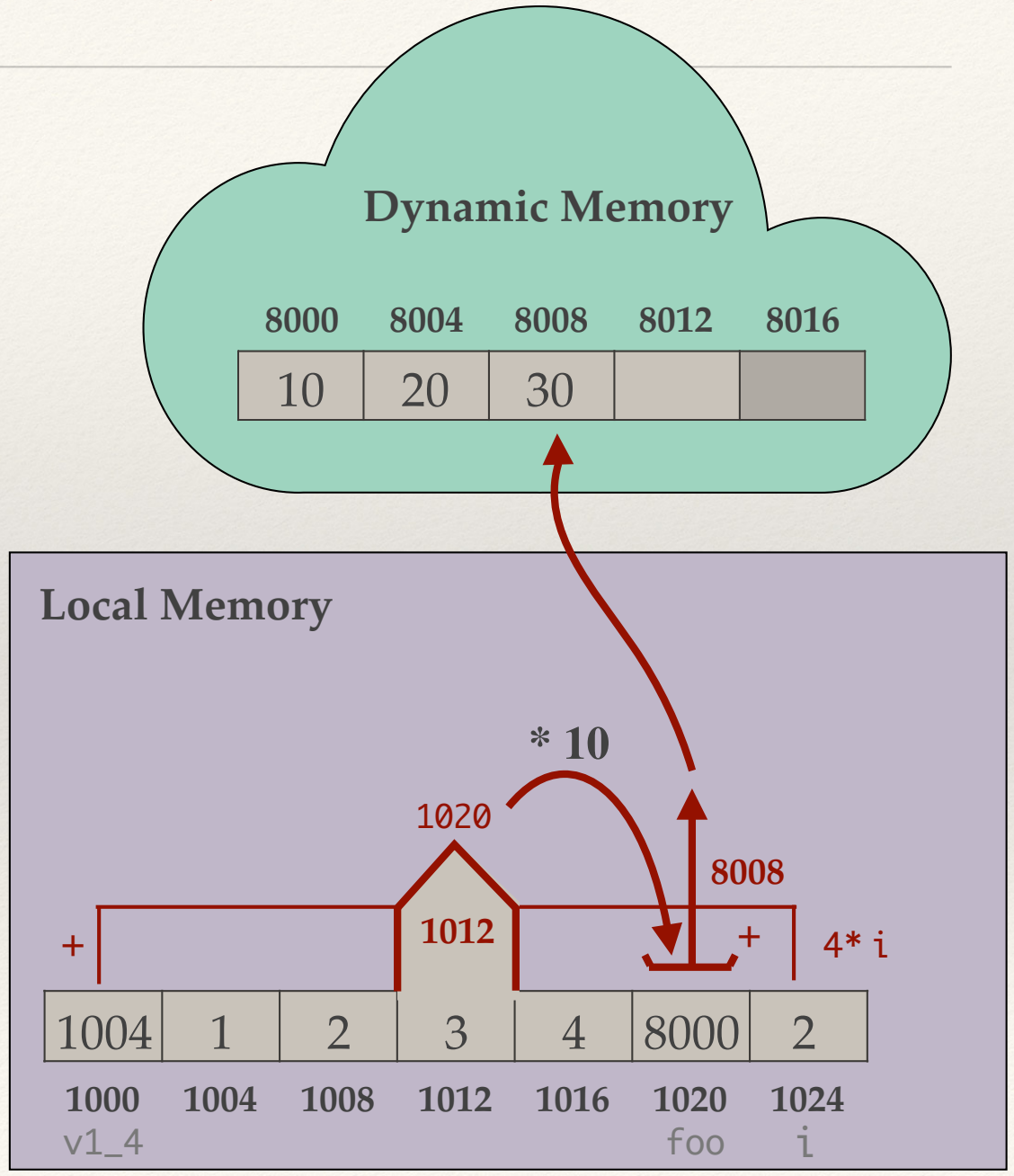




Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

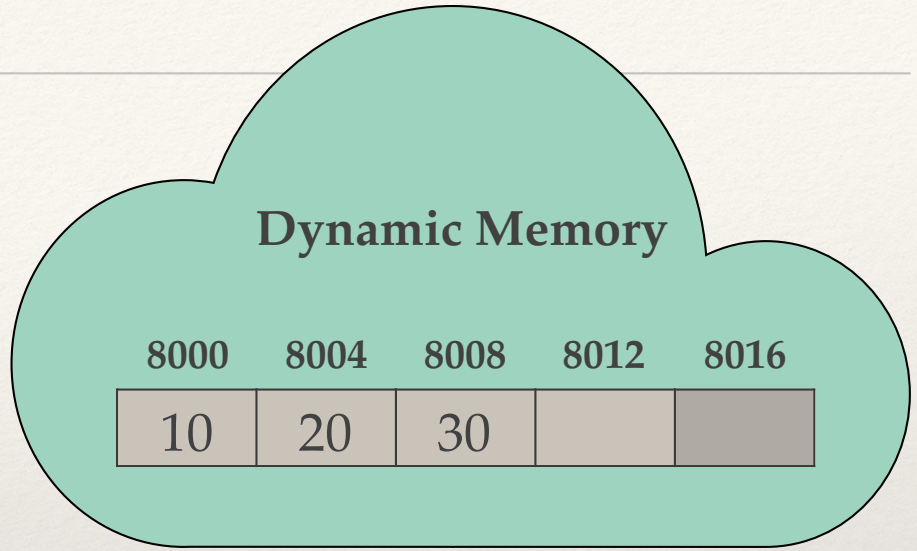
```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

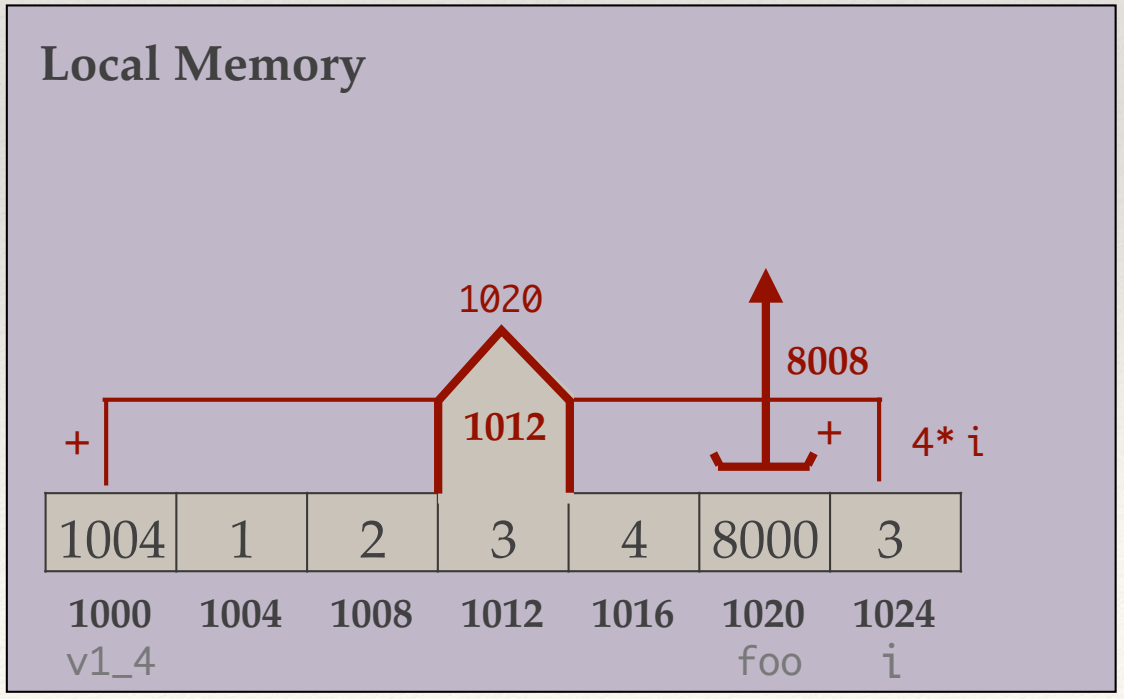


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

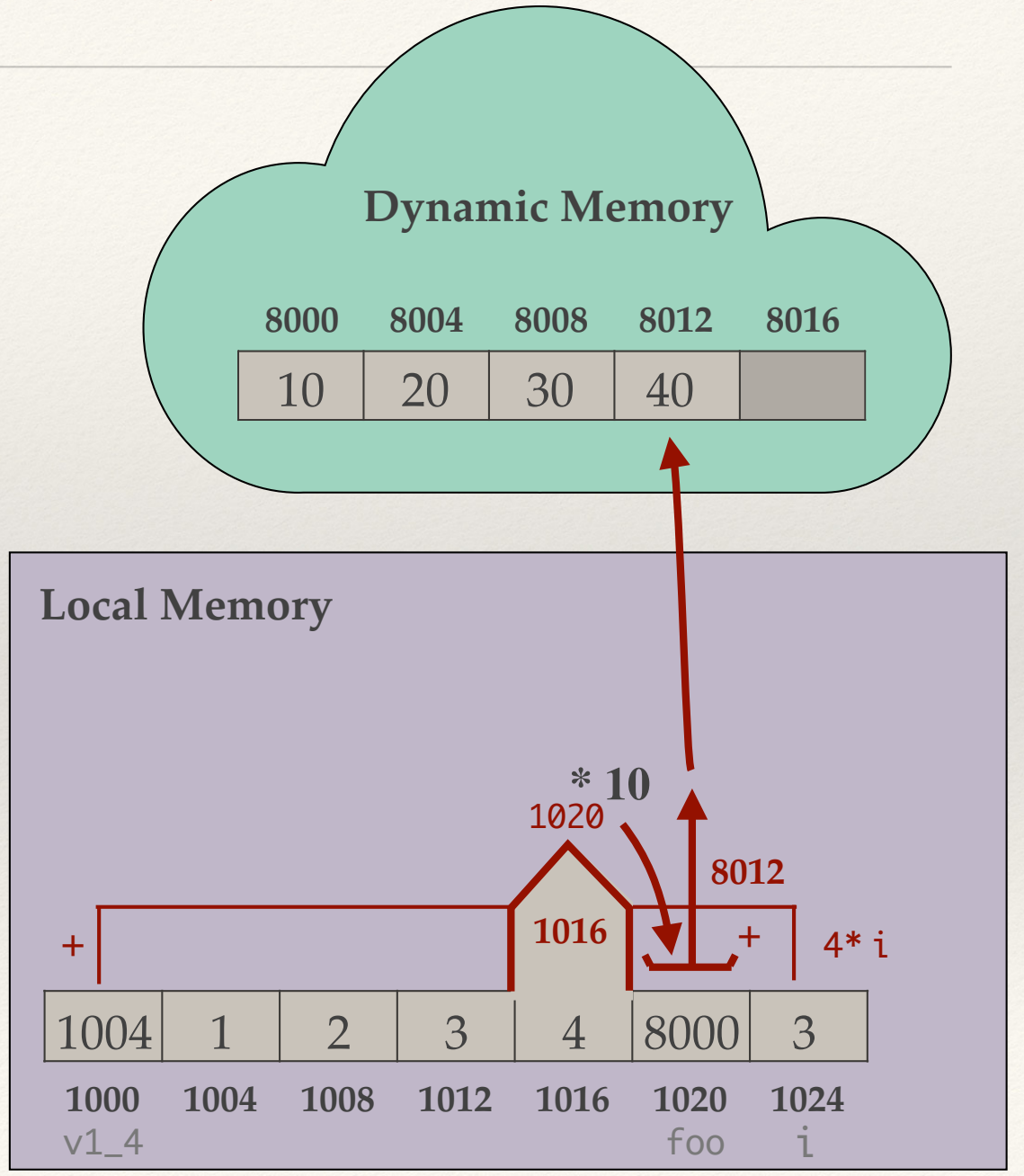




Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

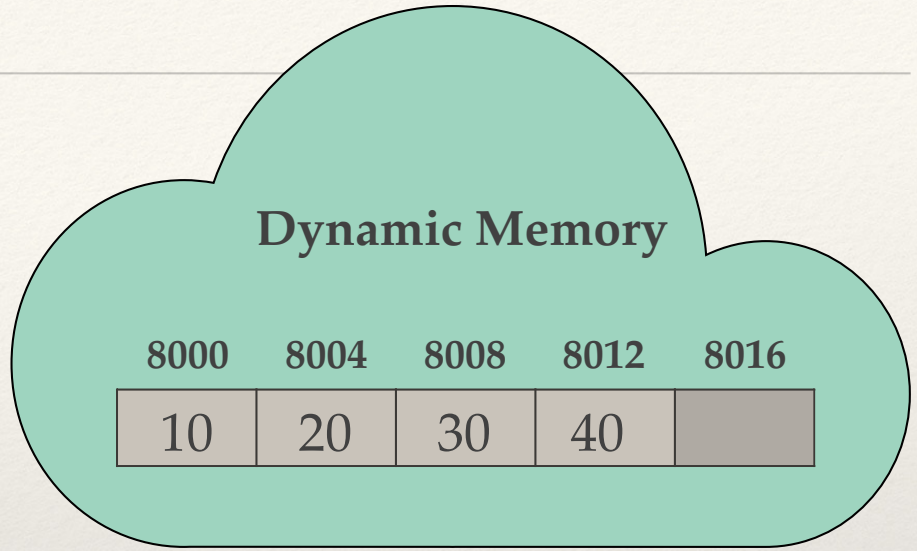
```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

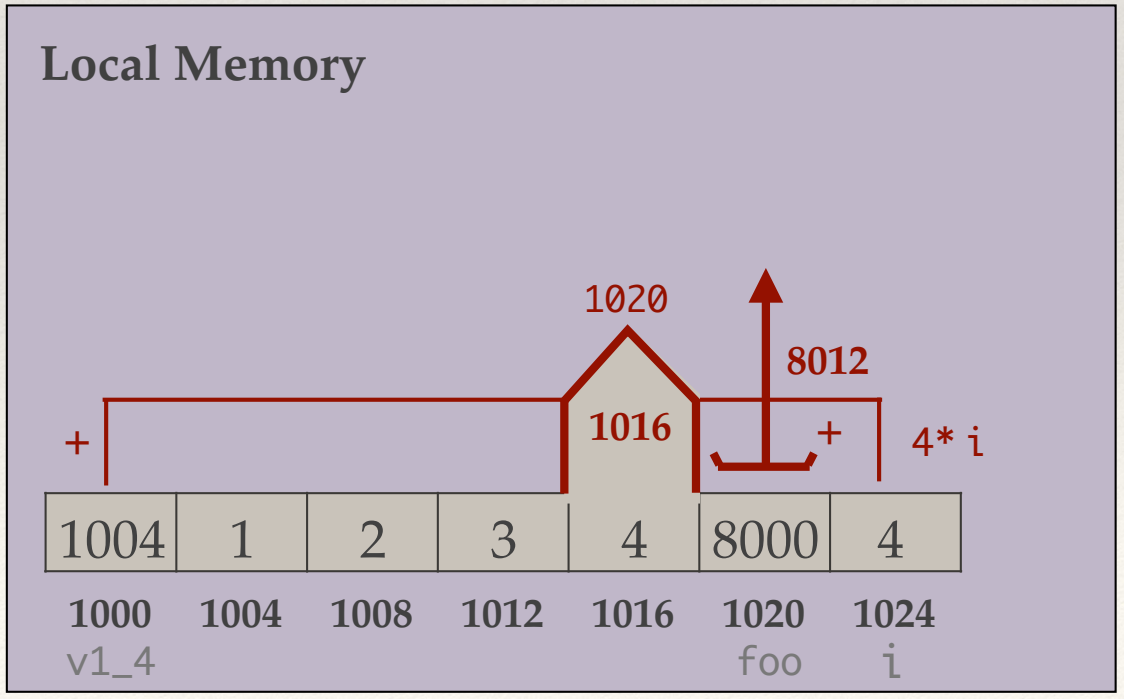


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

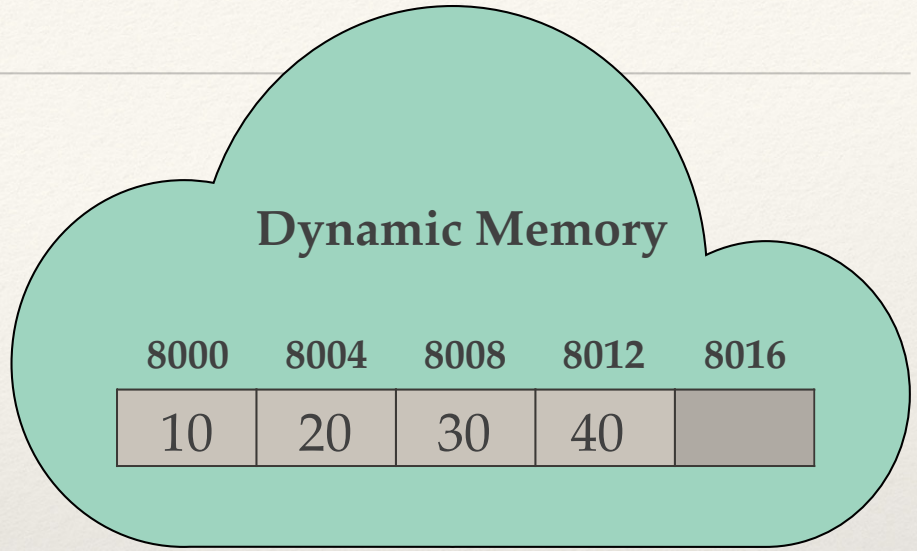


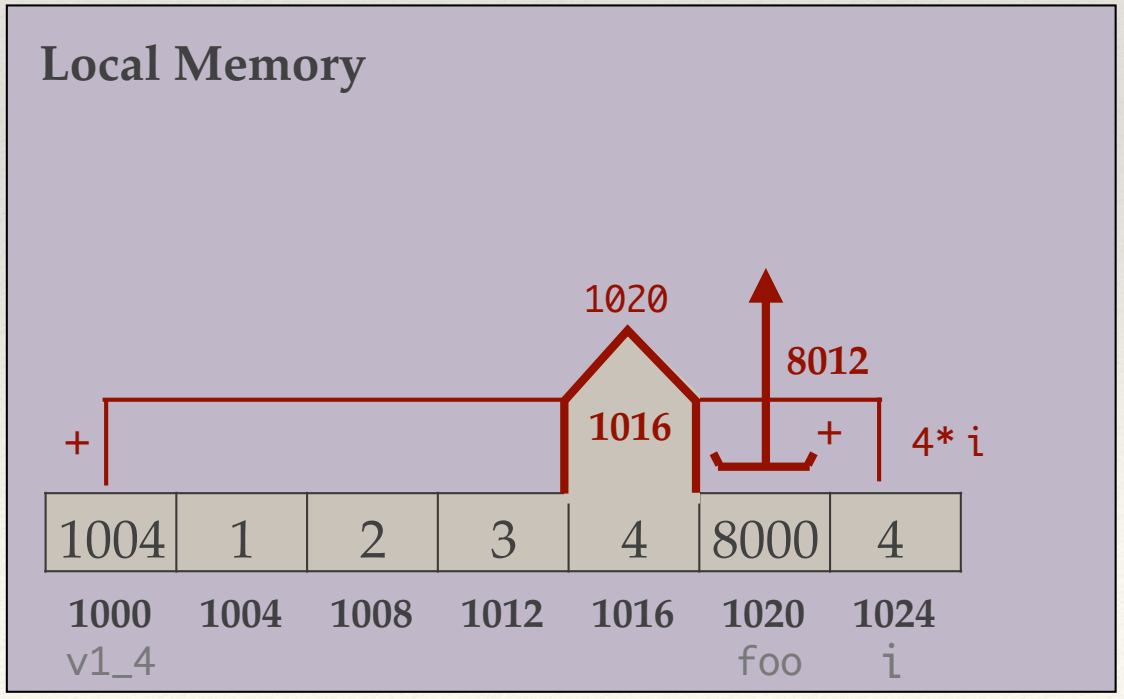


Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



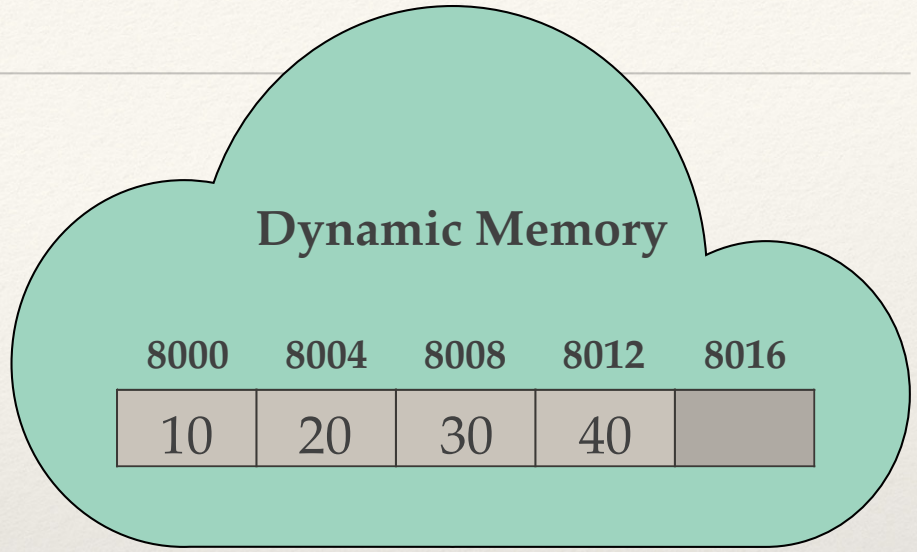


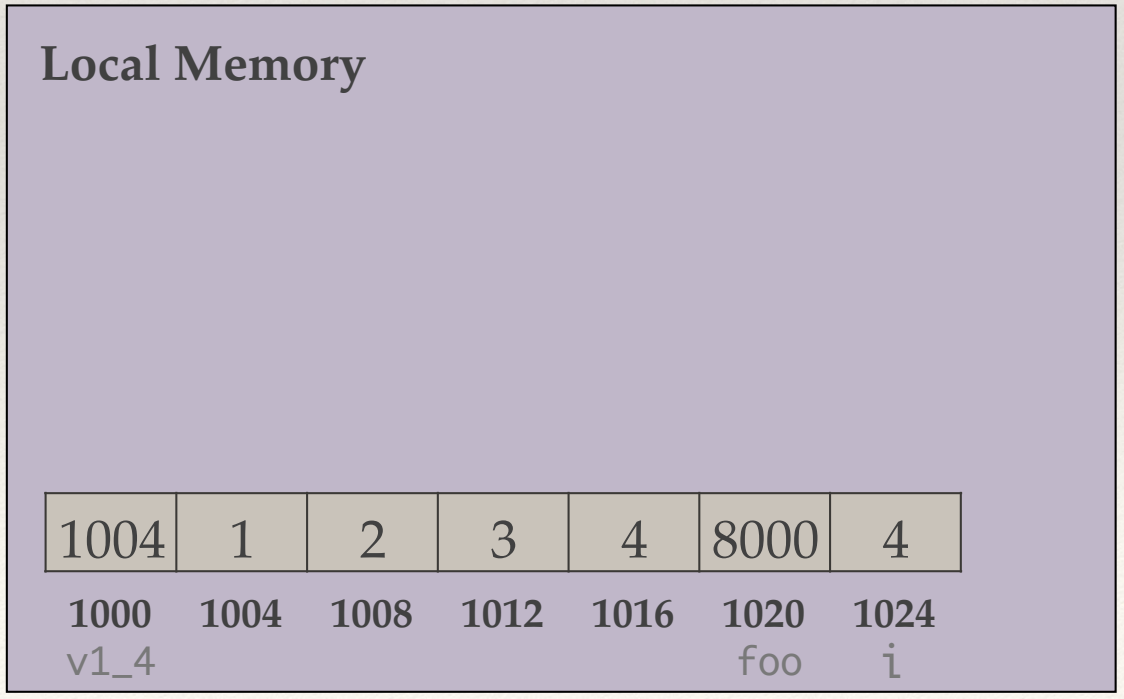
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



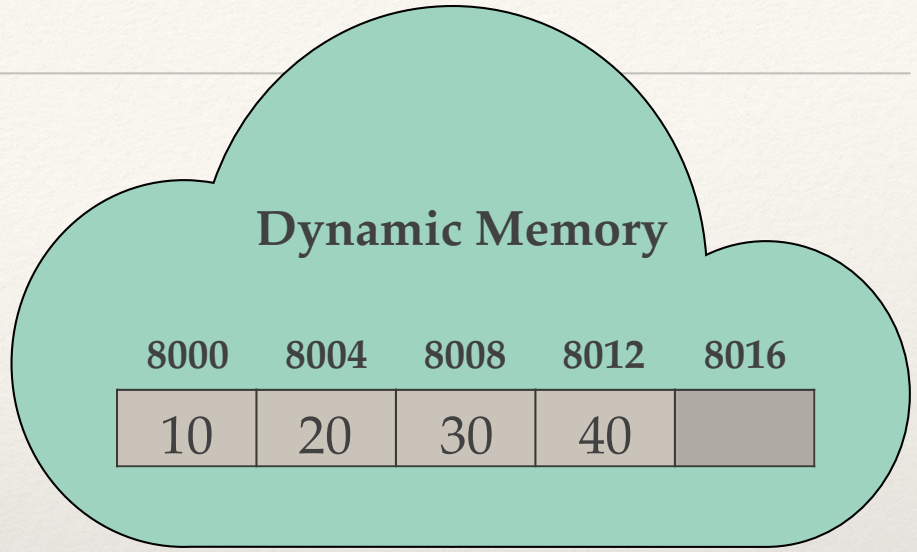


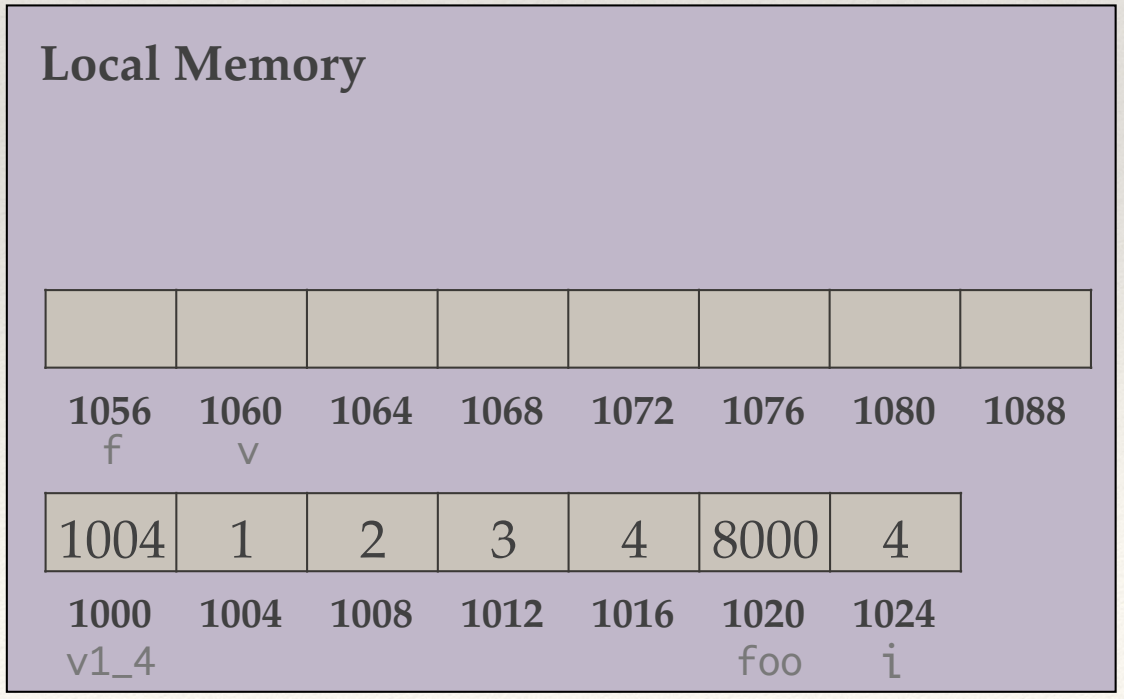
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



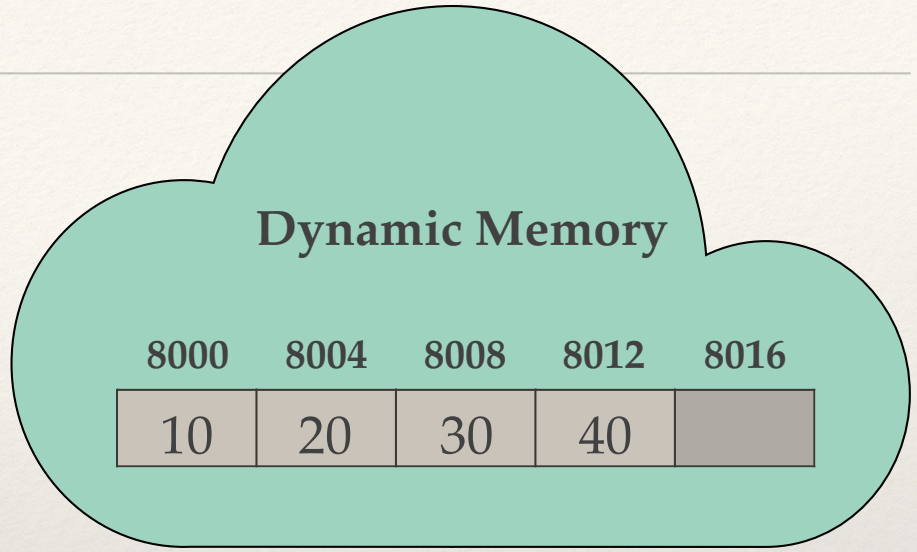


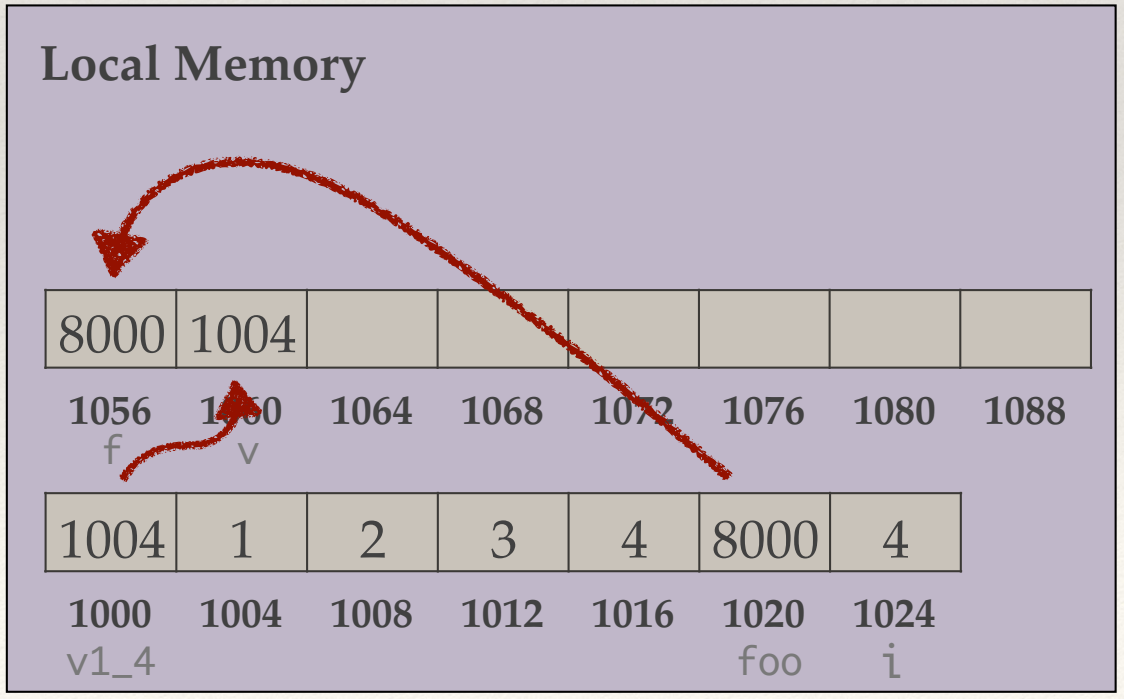
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



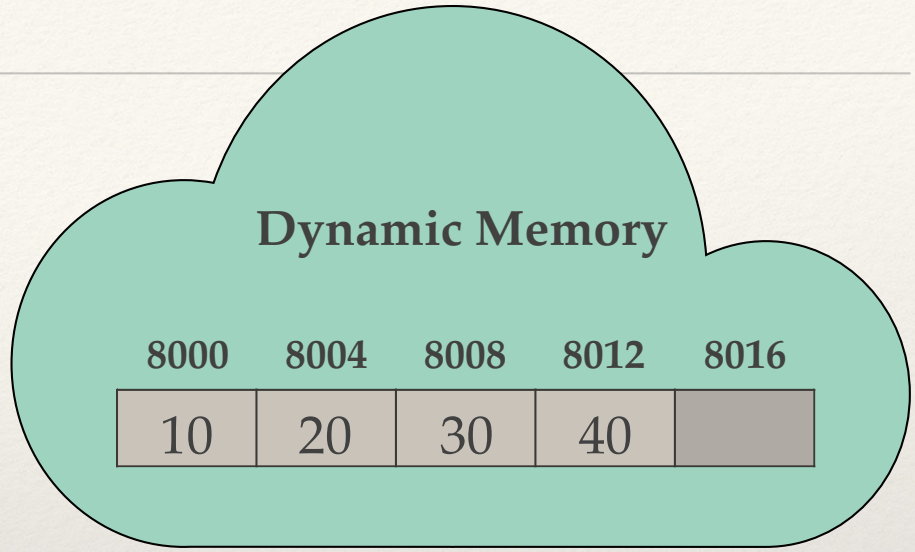


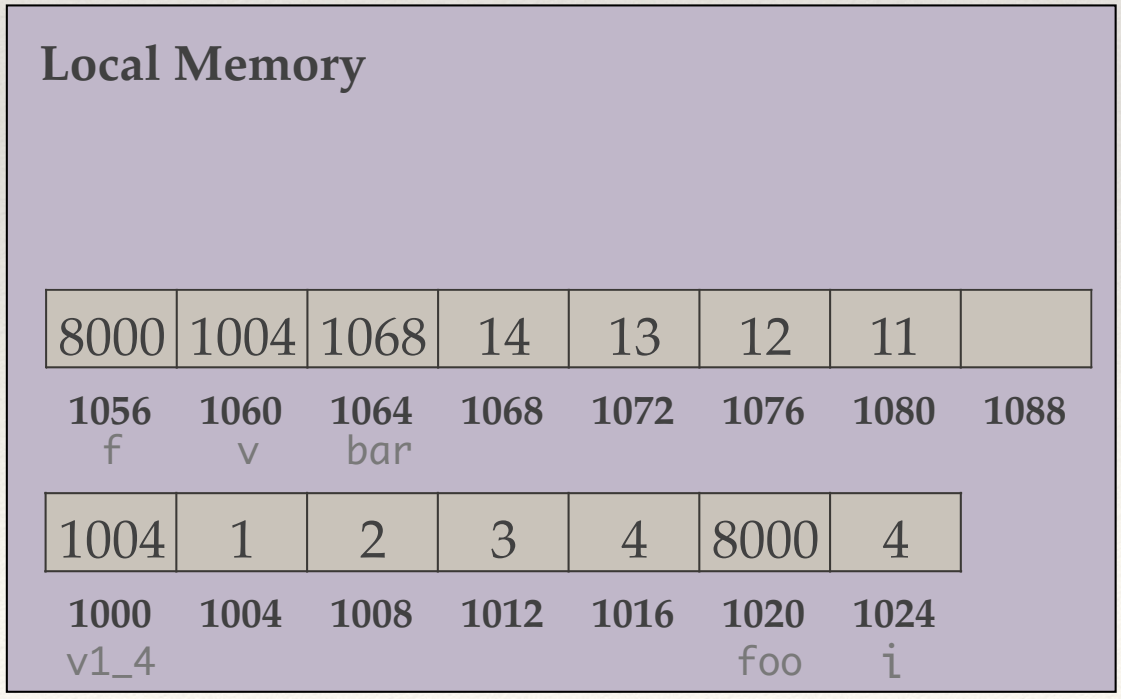
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



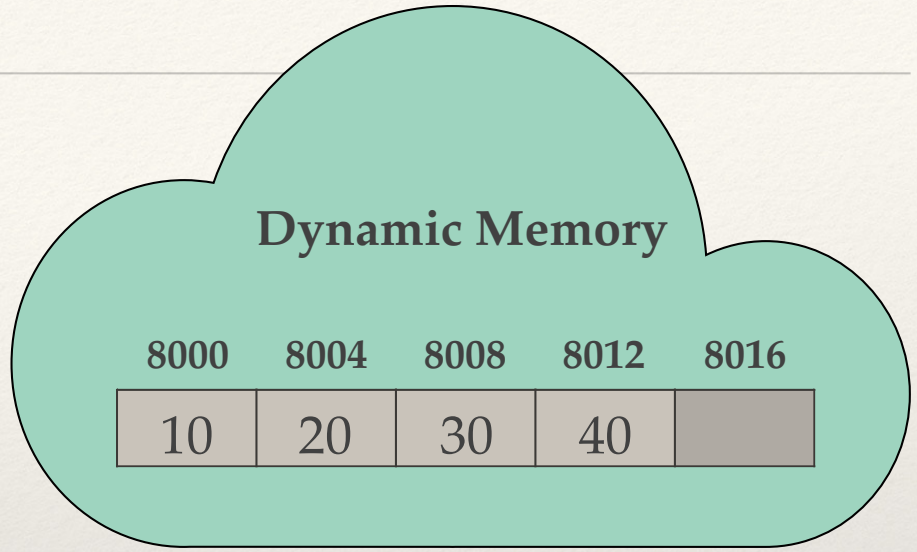


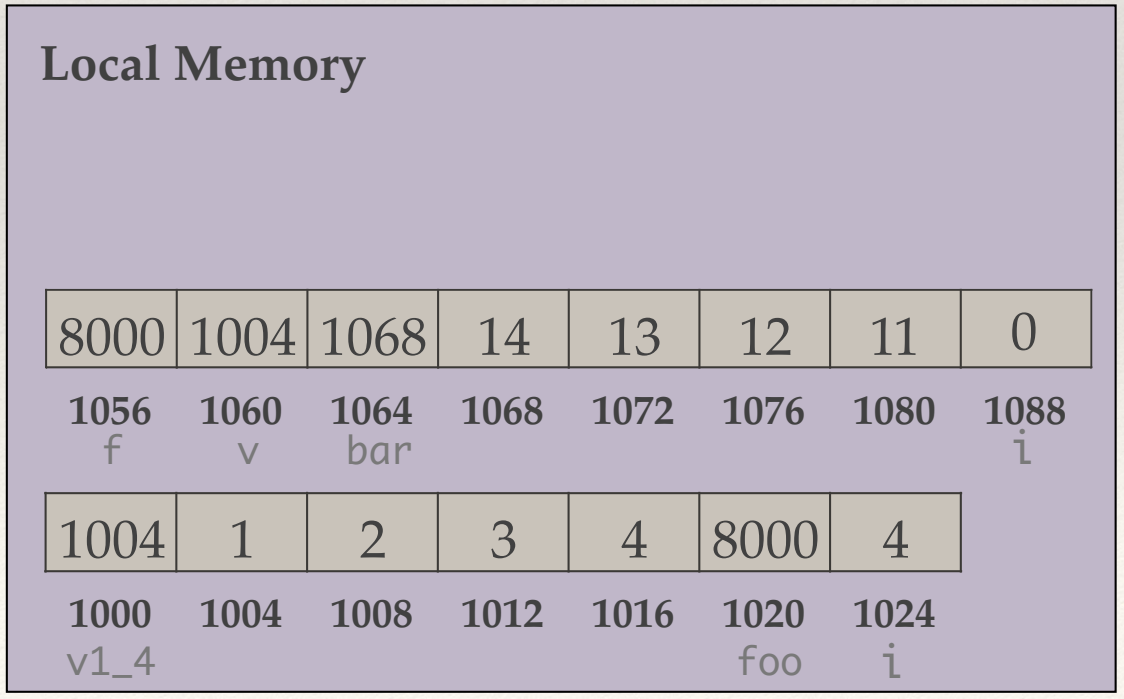
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



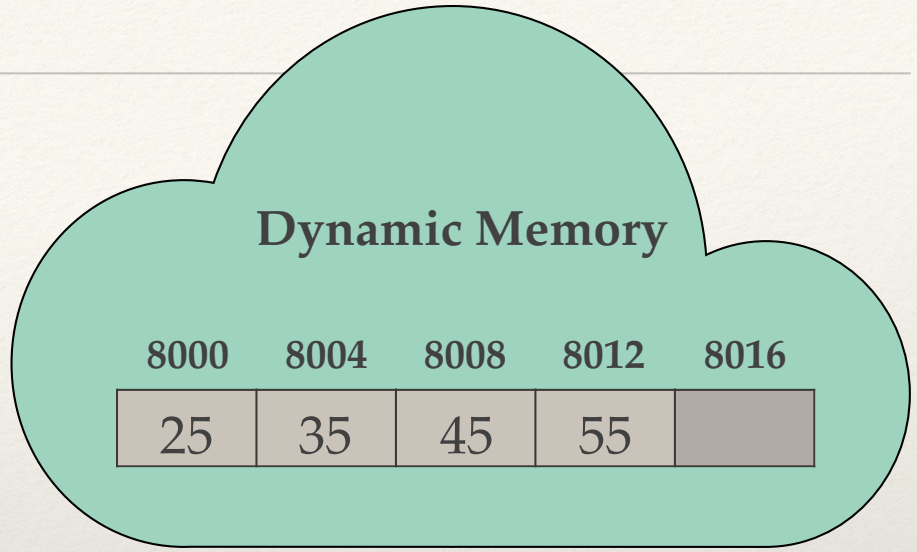


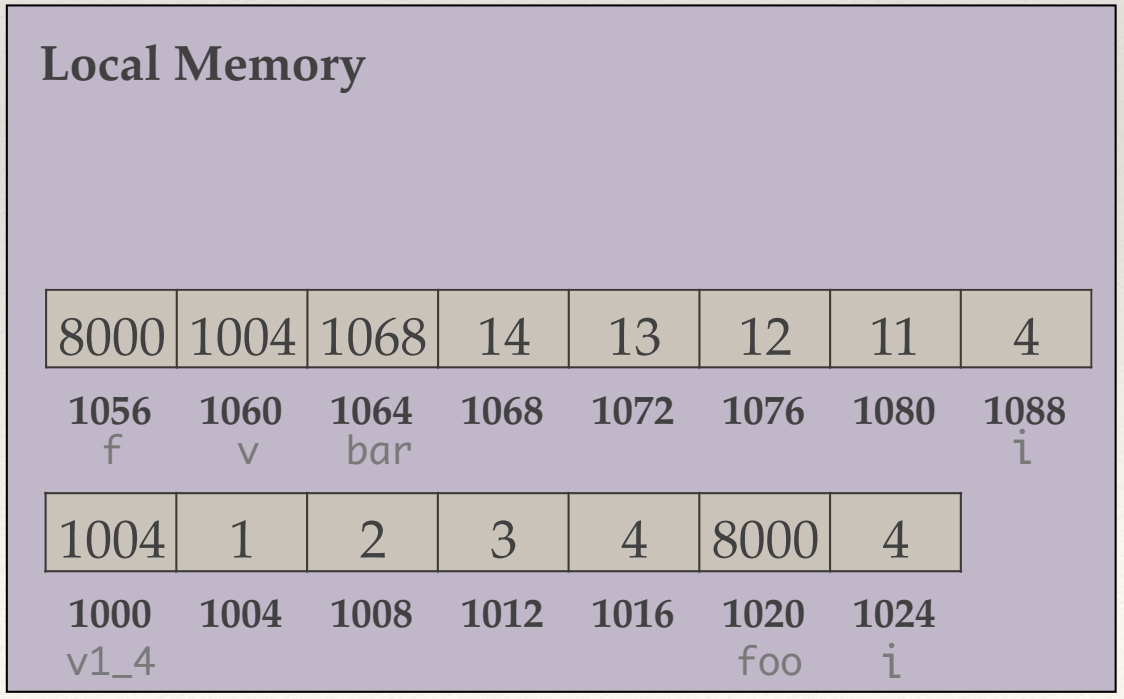
Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

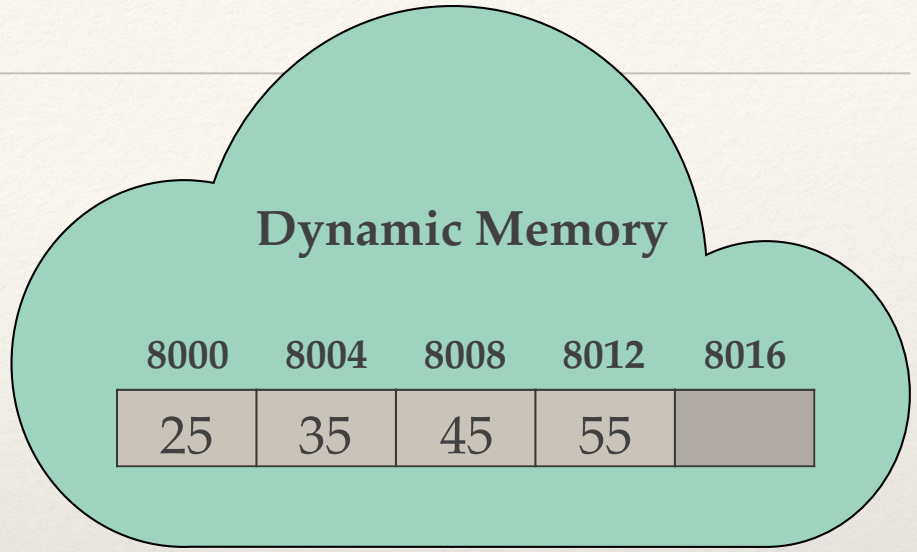




Memory Location and Function Calls

```
int int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



Local Memory

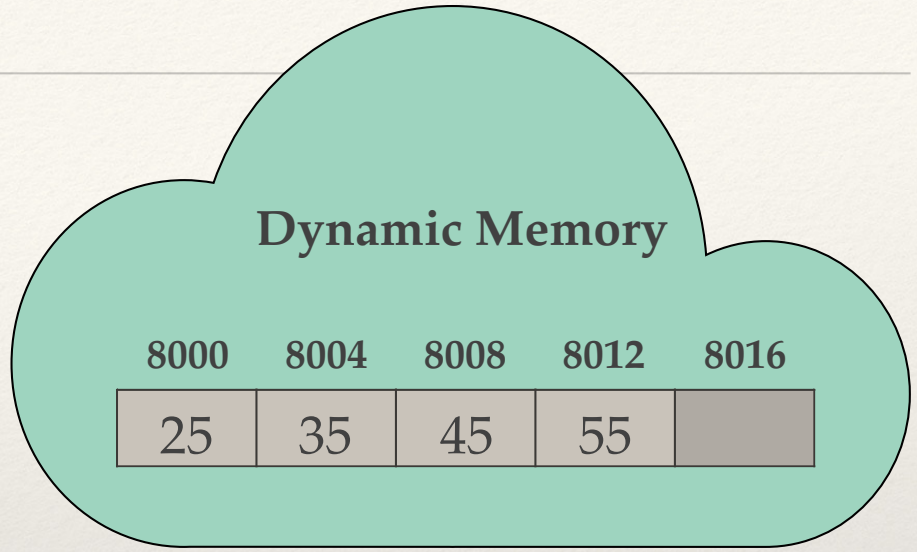
8000	1004	1068	14	13	12	11	4
1056 f	**1060** v	**1064** bar	**1068**	**1072**	**1076**	**1080**	**1088** i

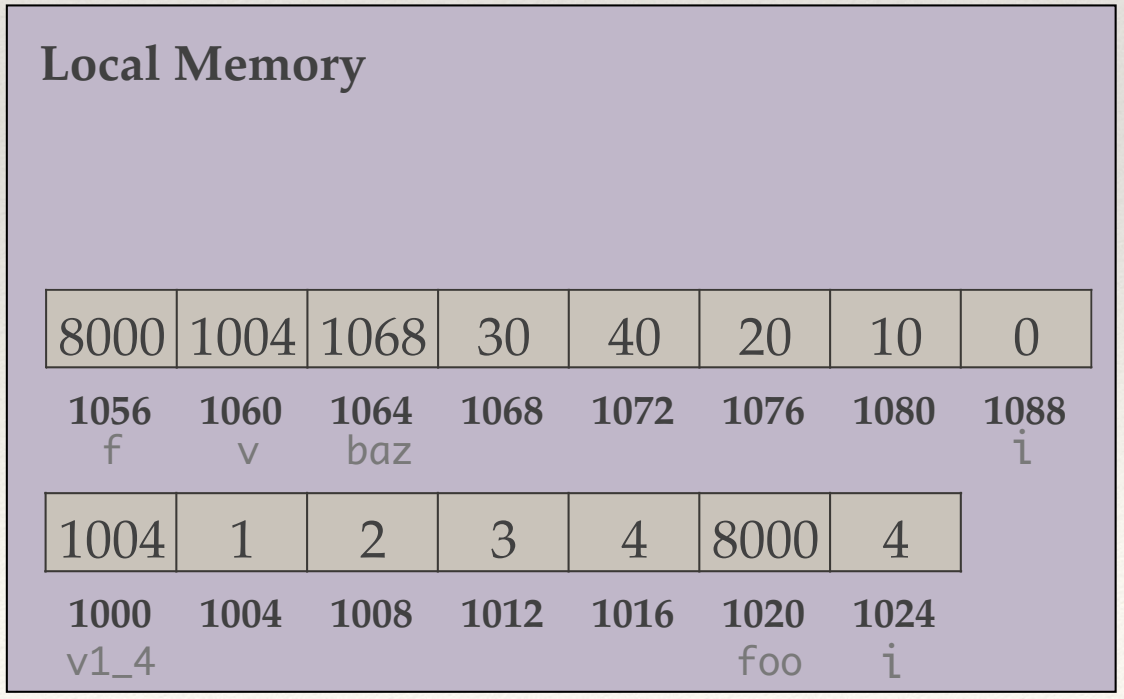
1004	1	2	3	4	8000	4
1000 v1_4	**1004**	**1008**	**1012**	**1016**	**1020** foo	**1024** i

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}

void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```

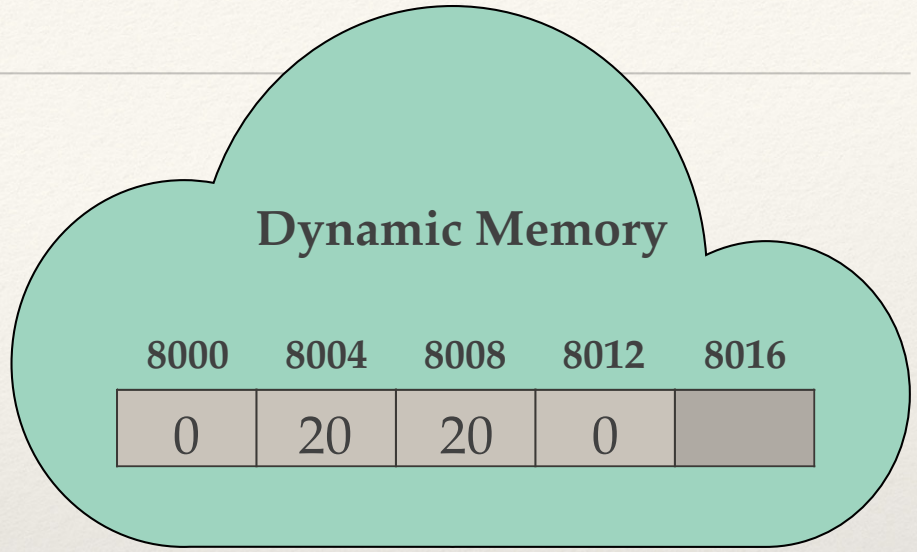




Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}

void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}
```

```
void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



Local Memory

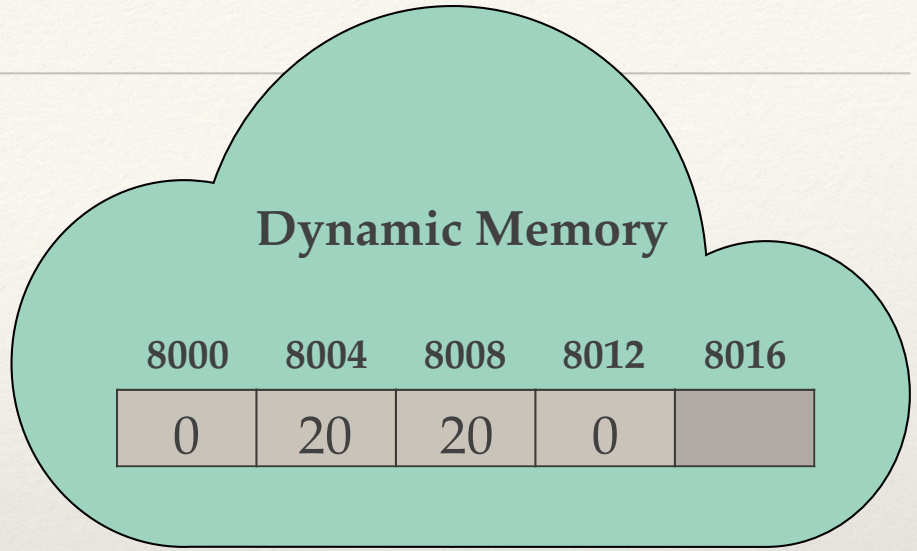
8000	1004	1068	30	40	20	10	4
1056 f	**1060** v	**1064** baz	**1068**	**1072**	**1076**	**1080**	**1088** i

1004	1	2	3	4	8000	4
1000 v1_4	**1004**	**1008**	**1012**	**1016**	**1020** foo	**1024** i

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```



Local Memory

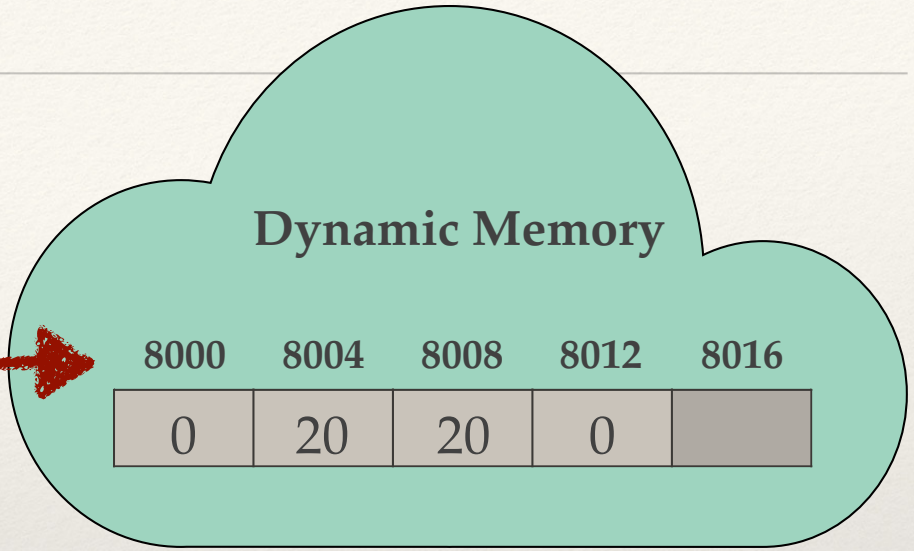
8000	1004	1068	30	40	20	10	4
1056	1060	1064	1068	1072	1076	1080	1088
f	v	baz					i

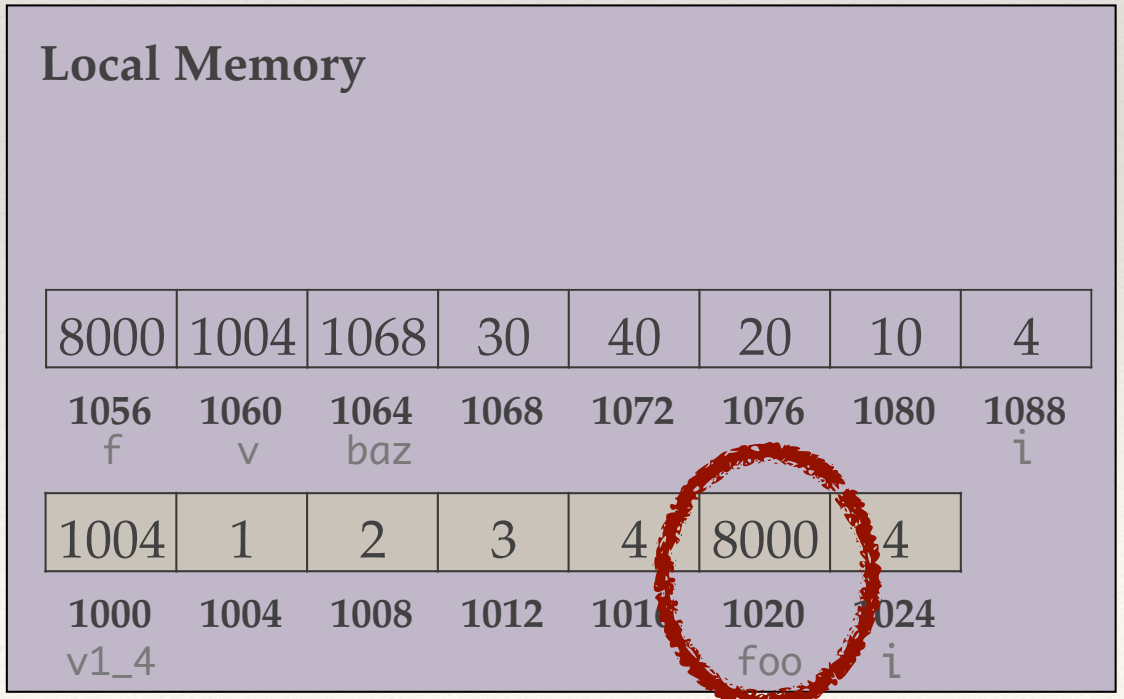
1004	1	2	3	4	8000	4
1000	1004	1008	1012	1016	1020	1024
v1_4					foo	i

Memory Location and Function Calls

```
int * example() {
    int v1_4[4] = {1, 2, 3, 4};
    int *foo = malloc(4 * sizeof(int));

    for(int i = 0; i < 4; i++)
        foo[i] = 10 * v1_4[i];

    addbar(foo, v1_4);
    modbaz(foo, v1_4);

    return foo;
}
```

```
void addbar(int f[], int v[]) {
    int bar[4] = {14, 13, 12, 11};

    for(int i = 0; i < 4; i++)
        f[i] += v[i] + bar[i];
}

void modbaz(int f[], int v[]) {
    int baz[4] = {30, 40, 20, 10};

    for(int i = 0; i < 4; i++)
        f[i] = (f[i] * v[i]) % baz[i];
}
```





Reallocating Memory

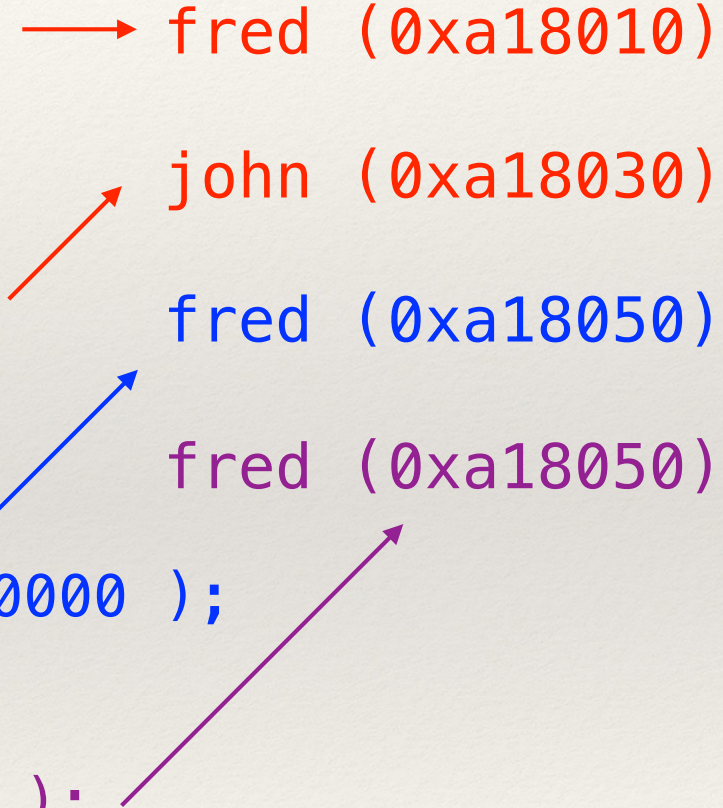
- It is possible to increase or decrease the size of the allocated memory using the `realloc()` function.

```
realloc ( void *ptr, size_t size )
```

- If there is not enough room to enlarge the memory allocation pointed to by `ptr`, `realloc()`
  - creates a new allocation,
  - copies as much of the old data pointed to by `ptr` as will fit to the new allocation,
  - frees the old allocation, and
  - returns a pointer to the allocated memory.

realloc()

```
char *ptr, *new;
ptr = malloc ( sizeof(char) * 10 );
strcpy ( ptr, “fred” );
printf ( “%s (%p)\n”, ptr, ptr );

new = malloc ( sizeof(char) * 10 );
strcpy ( new, “john” );
printf ( “%s (%p)\n”, new, new );

ptr = realloc ( ptr, sizeof(char)*20000 );
printf ( “%s (%p)\n”, ptr, ptr );

ptr = realloc ( ptr, sizeof(char)*2 );
printf ( “%s (%p)\n”, ptr, ptr );
```



```
char *ptr, *new;
ptr = malloc ( sizeof(char) * 10 );
strcpy ( ptr, “fred” );
printf ( “%s (%p)\n”, ptr, ptr );

new = malloc ( sizeof(char) * 10 );
strcpy ( new, “john” );
printf ( “%s (%p)\n”, new, new );

ptr = realloc ( ptr, sizeof(char)*20000 );
printf ( “%s (%p)\n”, ptr, ptr );

ptr = realloc ( ptr, sizeof(char)*2 );
printf ( “%s (%p)\n”, ptr, ptr );
```

→ fred (0xa18010)

→ john (0xa18030)

fred (0xa18050)

- *first* `realloc` *moves pointer to new location since memory needed is* ***too large*** *for the original location*

fred (0xa18050)

- *second* `realloc` *is* ***smaller*** *than the currently allocated memory so the pointer does not need to move and extra memory is freed.*

Question

- Why does the string still have 5 characters (“fred\0”) in it when the last `realloc()` only created space for 2 characters?
- Answer: `realloc()` does not clear the memory it frees. The end of the string (“ed\0”) is still in memory but it is **NOT** allocated to the program.
- A later `malloc()` may overwrite these characters.

Be Careful!!

- The previous examples are designed to demonstrate properties of realloc and are not examples of safe code.
- e.g. consider the statement that we had used

```
ptr = realloc ( ptr, sizeof(char)*20000 );
```

  - realloc can return NULL if the OS doesn't have the memory
  - if this happens, ptr has now been set to NULL
  - in this situation realloc doesn't free the ptr memory
    - this gives you an opportunity to ask for less space with a different realloc, or just go on with the space you have with destroying the contents in ptr
  - you haven't yet freed the ptr memory either
  - but as ptr is now NULL you no longer have the original address!

Be Careful!!

- The previous examples are designed to demonstrate properties of realloc and are not examples of safe code.
- e.g. conside

```
ptr = re
```

  - realloc ca ory
  - if this ha
  - in this situation realloc doesn't free the ptr memory
    - this gives you an opportunity to ask for less space with a different realloc, or just go on with the space you have with destroying the contents in ptr
  - you haven't yet freed the ptr memory either
  - but as ptr is now NULL you no longer have the original address!

This means…

1. You have created a memory leak!
2. You have lost your data!!!

Safe Code

```
char *ptr, *temp;
ptr = malloc ( sizeof(char) * 10 );
strcpy ( ptr, “fred” );
printf ( “%s (%p)\n”, ptr, ptr );

temp = realloc ( ptr, sizeof(char)*20000 );

if temp != NULL {
    ptr = temp;
} else {
    /* error handling code goes here */
}

printf ( “%s (%p)\n”, ptr, ptr );
```

fred (0xa18010)

fred (0xa18050)