211: Computer Architecture
Fall 2019

Topic:

- C Programming

Anatomy of a C Program

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

char cMessage[] = "Hello\n";

/* Execution will start here */
int main (int argc, char **argv)
{
    int i, count;

    count = atoi(argv[1]);
    for (i = 0; i < count; i++) {
        printf("Hello %d\n", i);
    }
}
```

include files

declaration of global variables

comment

one or more function; each program starts execution at “main”

declaration of local variables

code implementing function

Pointers

A pointer is just an address

Can have variables of type pointer

- Hold addresses as values
- Used for indirection

When declaring a pointer variable, need to declare the type of the data item the pointer will point to

- int *p;   /* p will point to a int data item */

Pointer operators

- De-reference: *
  - *p gives the value stored at the address pointed to by p
- Address: &
  - &v gives the address of the variable v

Arrays

Arrays are contiguous sequences of data items

- All data items are of the same type
- Declaration of an array of integers: “int a[20];”
- Access of an array item: “a[15]”

Array index always start at 0

The C compiler and runtime system do not check array boundaries

- The compiler will happily let you do the following:
  - int a[10]; a[11] = 5;

Arrays & Pointers

Given

```
char word[10];
char *cptr;
cptr = word;
```

Each row in following table gives equivalent forms

`cptr`	`word`	`&word[0]`
`cptr + n`	`word + n`	`&word[n]`
`*cptr`	`*word`	`word[0]`
`*(cptr + n)`	`*(word + n)`	`word[n]`

Pointer Arithmetic

Be careful when you are computing addresses

Address calculations with pointers are dependent on the size of the data the pointers are pointing to

Examples:

- `int *i; …; i++;      /* i = i + 4 */`
- `char *c; …; c++;     /* c = c + 1 */`
- `double *d; …; d++;   /* d = d + 8 */`

Another example:

```
double x[10];
double *y = x;
*(y + 3) = 13;   /* x[3] = 13 */
```

Strings: Arrays of Characters

Allocate space for a string just like any other array:

```
char outputString[16];
```

Each string should end with a ‘\0’ character

Special syntax for initializing a string:

```
char outputString[16] = "Result";
```

…which is the same as:

```
outputString[0] = 'R';
outputString[1] = 'e';
...
outputString[6] = ‘\0’;
```

The ‘\0’ allows functions like strlen() to work on arbitrary strings

Defining a Struct

We first need to define a new type for the compiler and tell it what our struct looks like.

```
struct flightType {
        char flightNum[7];    /* max 6 characters */
        int altitude;         /* in meters */
        int longitude;        /* in tenths of degrees */
        int latitude;         /* in tenths of degrees */
        int heading;          /* in tenths of degrees */
    double airSpeed;          /* in km/hr */
};
```

This tells the compiler how big our struct is and how the different data items are laid out in memory

- But it does not allocate any memory
- Memory is only allocated when a variable is declared

Declaring and Using a Struct

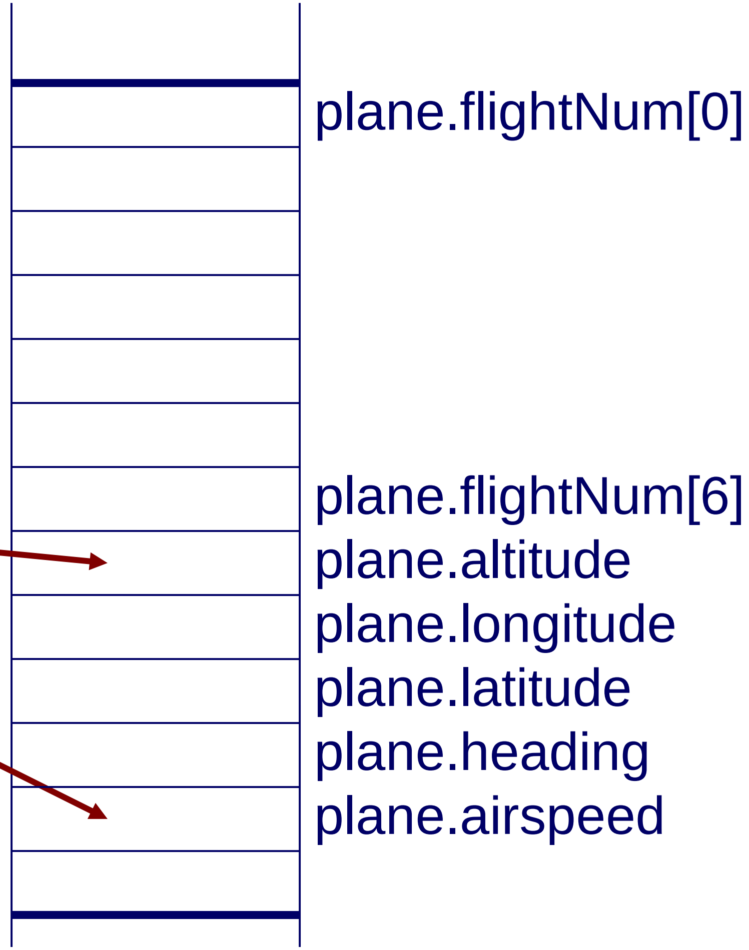
To allocate memory for a struct, we declare a variable using our new data type.

```
struct flightType plane;
```

Memory is allocated, and we can access individual members of this variable:

```
plane.altitude = 10000;
plane.airSpeed = 800.0;

foo(&(plane.airSpeed);
/* pass the address of
    plane.airSpeed */
```



Array of Structs

Can declare an array of struct items:

- struct flightType planes[100];

Each array element is a struct item of type "struct flightType"

To access member of a particular element:

- planes[34].altitude = 10000;

Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:

- (planes[34]).altitude = 10000;

Pointer to Struct

We can declare and create a pointer to a struct:

```
struct flightType *planePtr;
planePtr = &planes[34];
```

To access a member of the struct addressed by dayPtr:

```
(*planePtr).altitude = 10000;
```

Because the . operator has higher precedence than *, this is NOT the same as:

```
*planePtr.altitude = 10000;
```

C provides special syntax for accessing a struct member through a pointer:

```
planePtr->altitude = 10000;
```

Dynamic Allocation

What if we want to write a program to handle a variable amount of data?

- E.g., sort an arbitrary set of numbers
- Can’t allocate an array because don’t know how many numbers we will get
  - Could allocate a very large array
  - Inflexible and inefficient

Answer: dynamic memory allocation

- Similar to “new” in Java

Memory Management 101

When a function call is performed in a program, the run-time system must allocate resources to execute it

- Memory for any local variables, arguments, and result

The same function can be called many times (Example: recursion)

- Each instance will require some resources

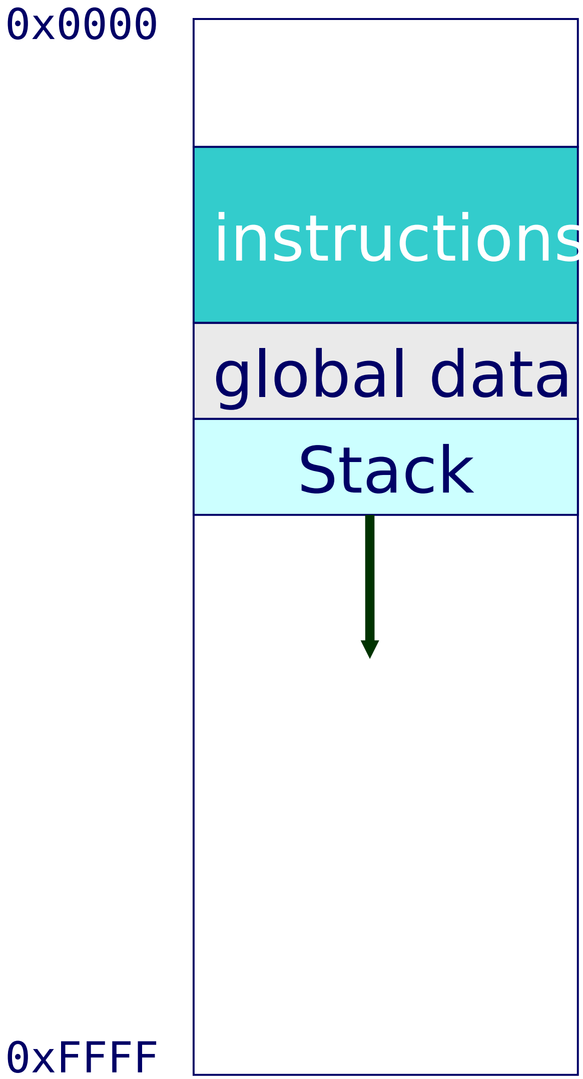
The state associated with a function is called an activation record

Allocating Space for Variables

Activation records are allocated on a call stack

- Function calls leads to a new activation record pushed on top of the stack
- Activation record is popped off the stack when the function returns

Let’s see an example



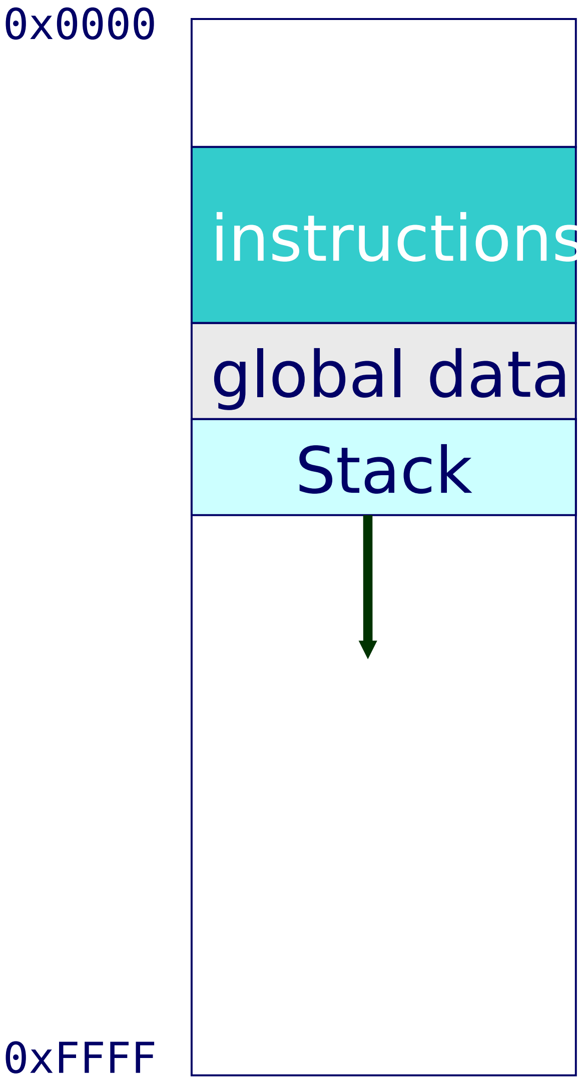
Allocating Space for Variables

Compute the sum of number from 1 to N

```
int summation(int n){
  if(n == 0) return 0
  else return n + summation(n-1);
}
…
summation(5);
```

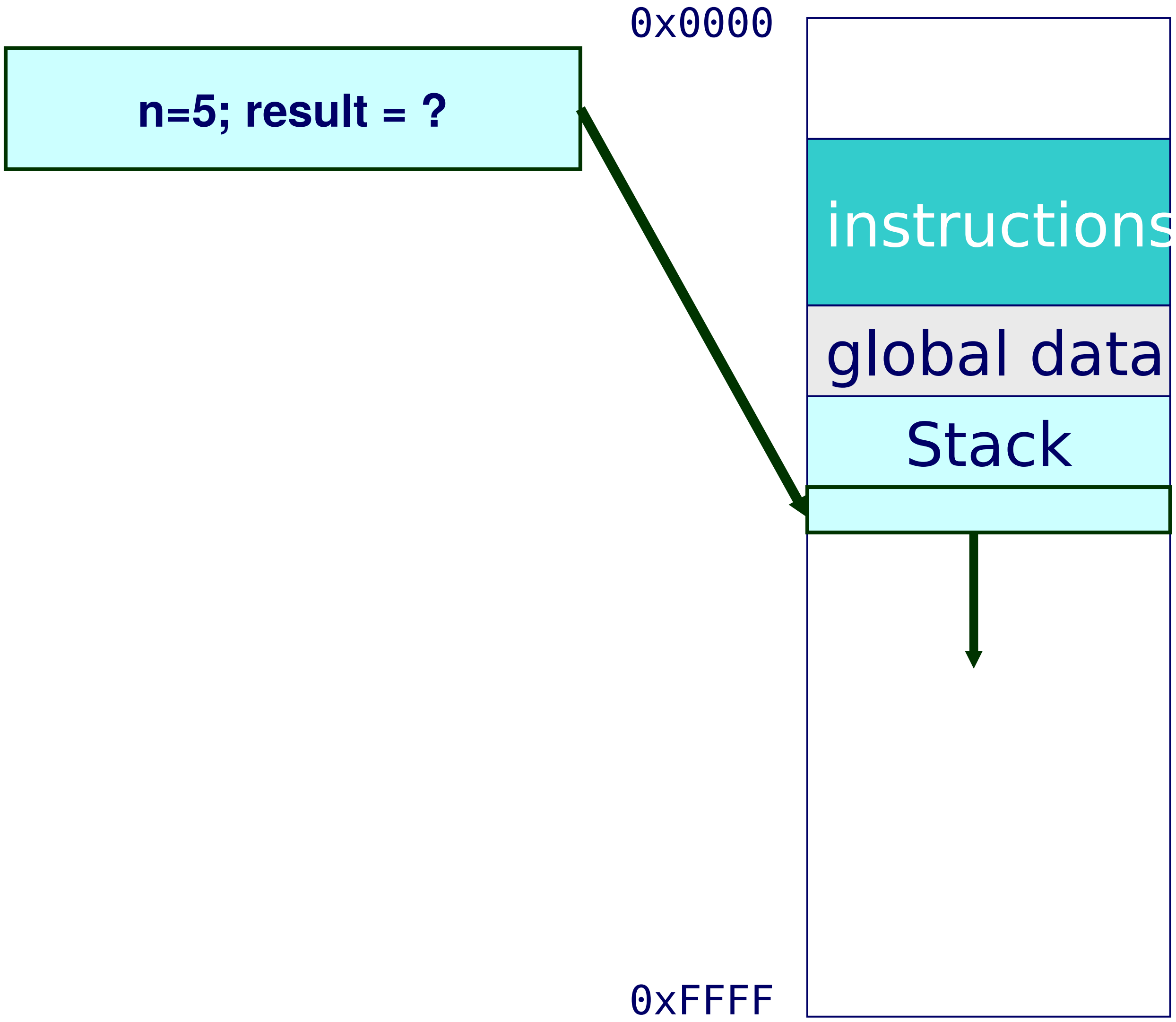
Recall that the activation record for a function contains state for all arguments, local variables, and result

int n; int result;

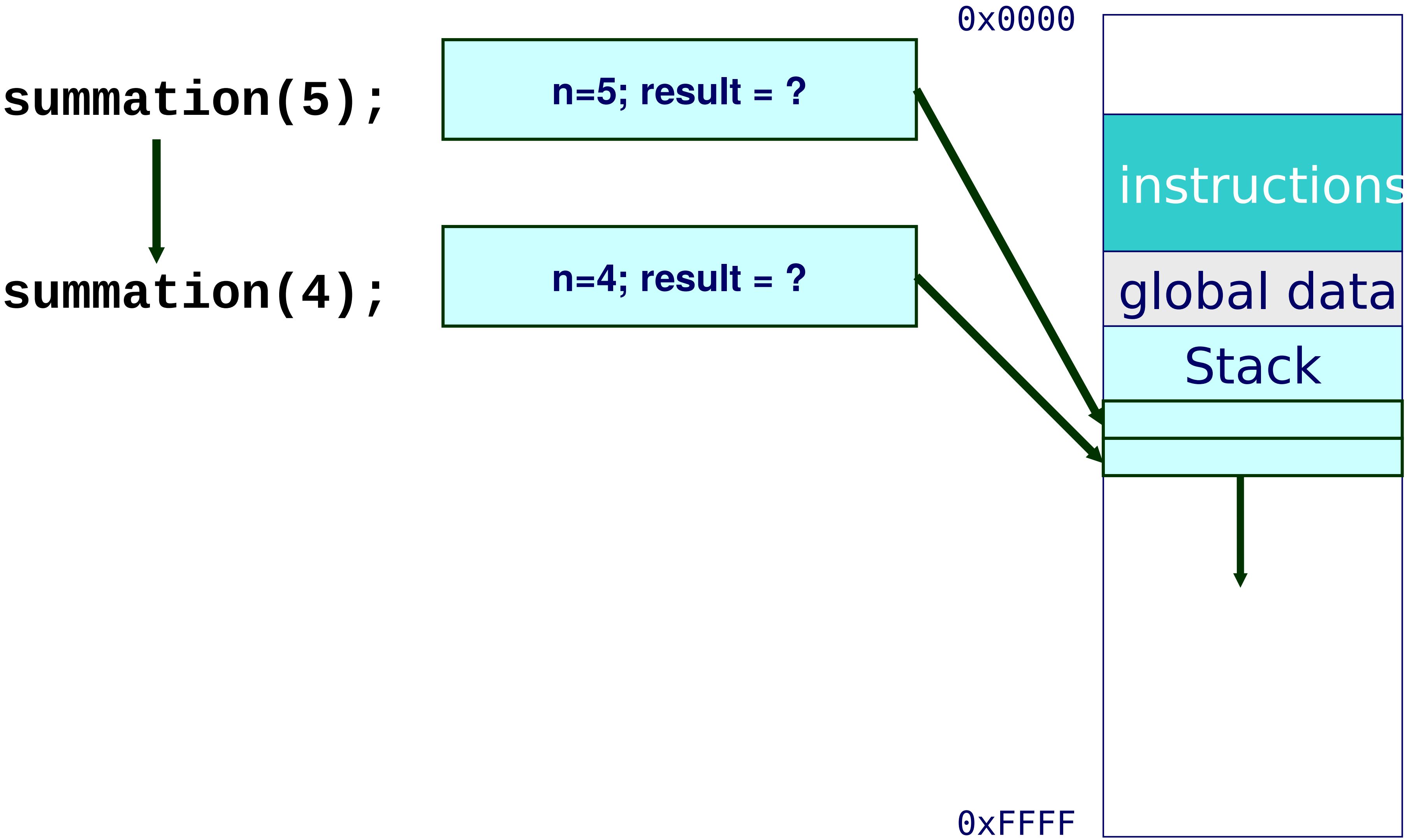


Allocating Space for Variables

```
summation(5);
```



Allocating Space for Variables



```
summation(5);
```

n=5; result = ?

```
summation(4);
```

n=4; result = ?

0x0000

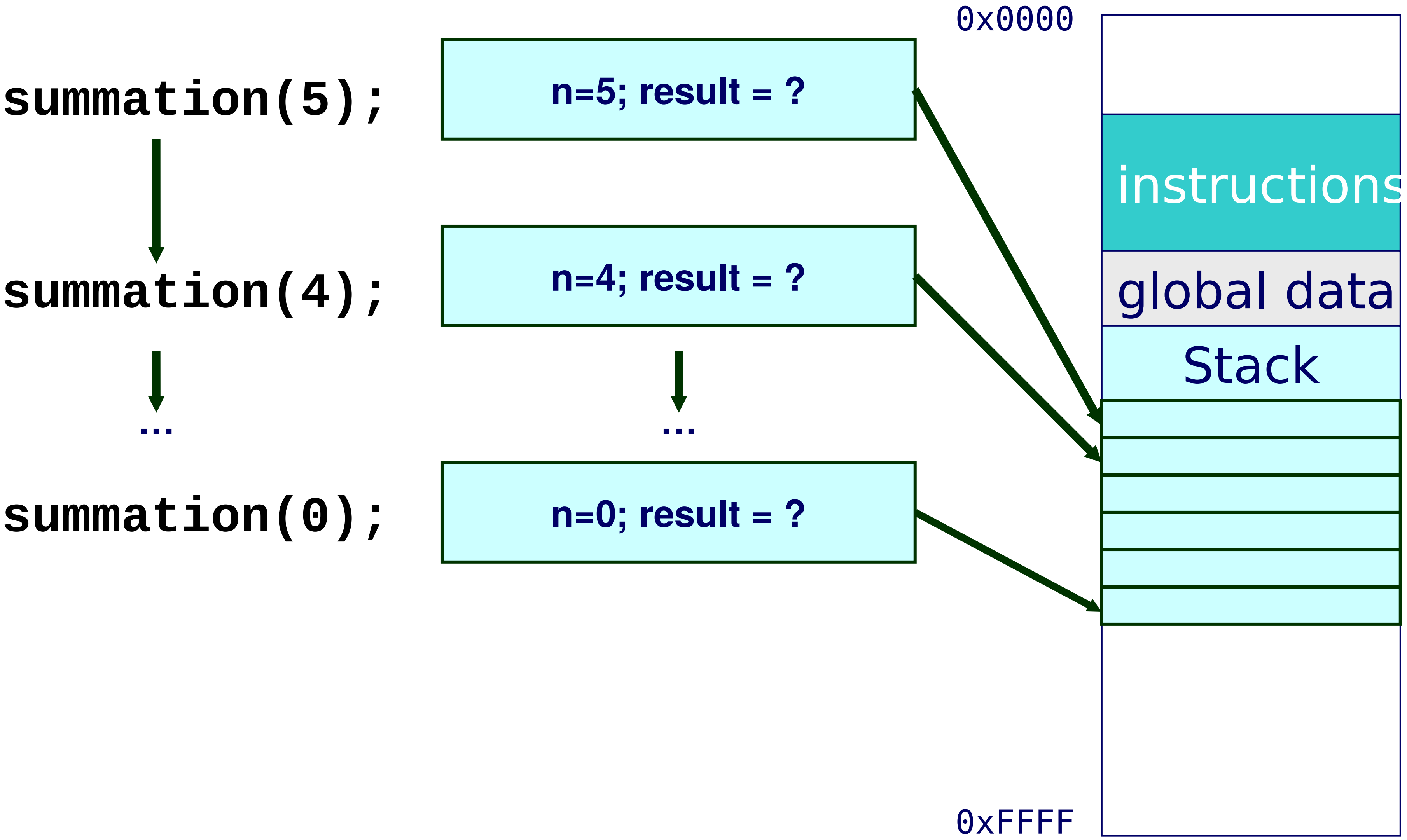
instructions

global data

Stack

0xFFFF

Allocating Space for Variables



```
summation(5);
```

n=5; result = ?

```
summation(4);
```

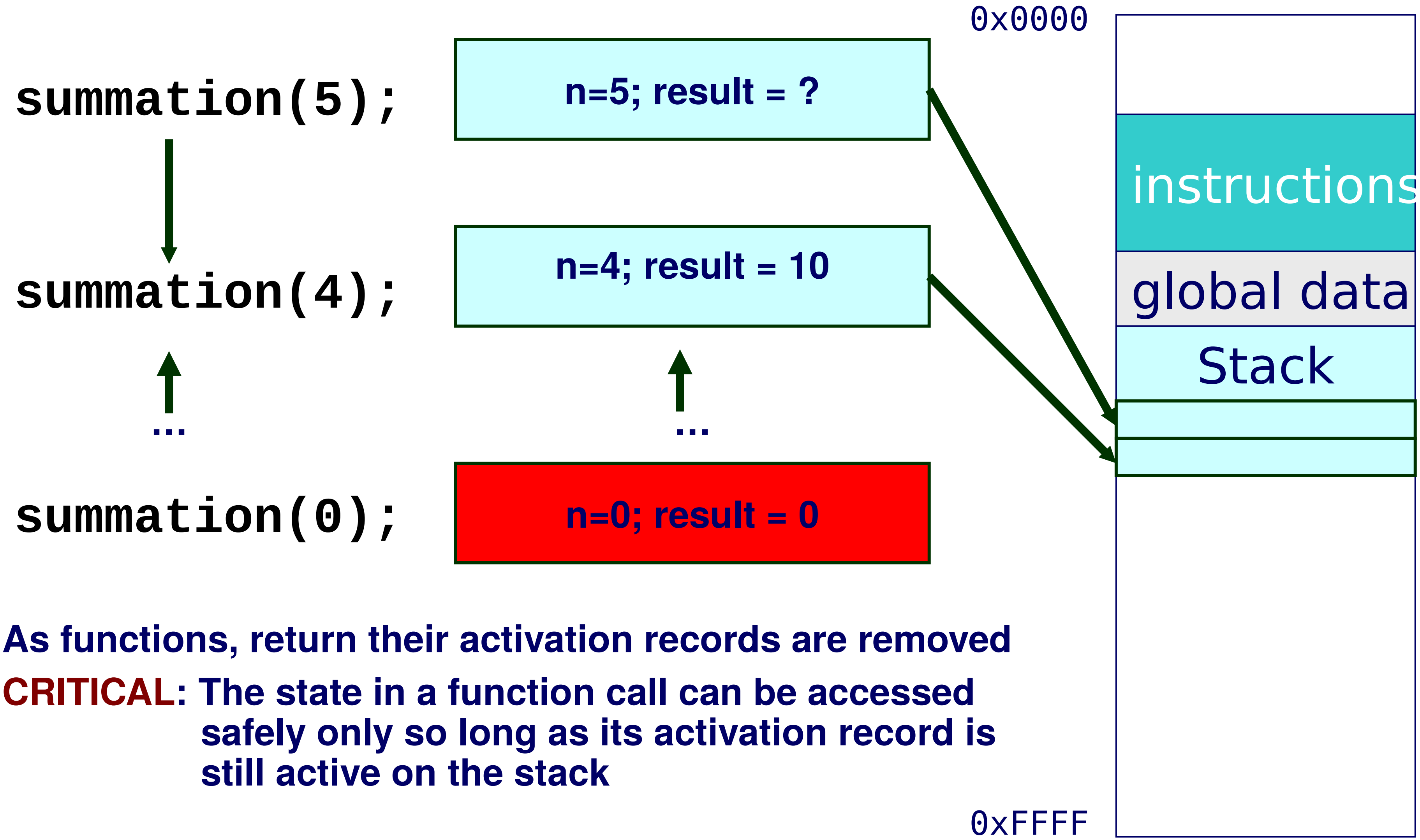
n=4; result = ?

...

```
summation(0);
```

n=0; result = ?

Allocating Space for Variables



As functions, return their activation records are removed

CRITICAL: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Dynamic Allocation

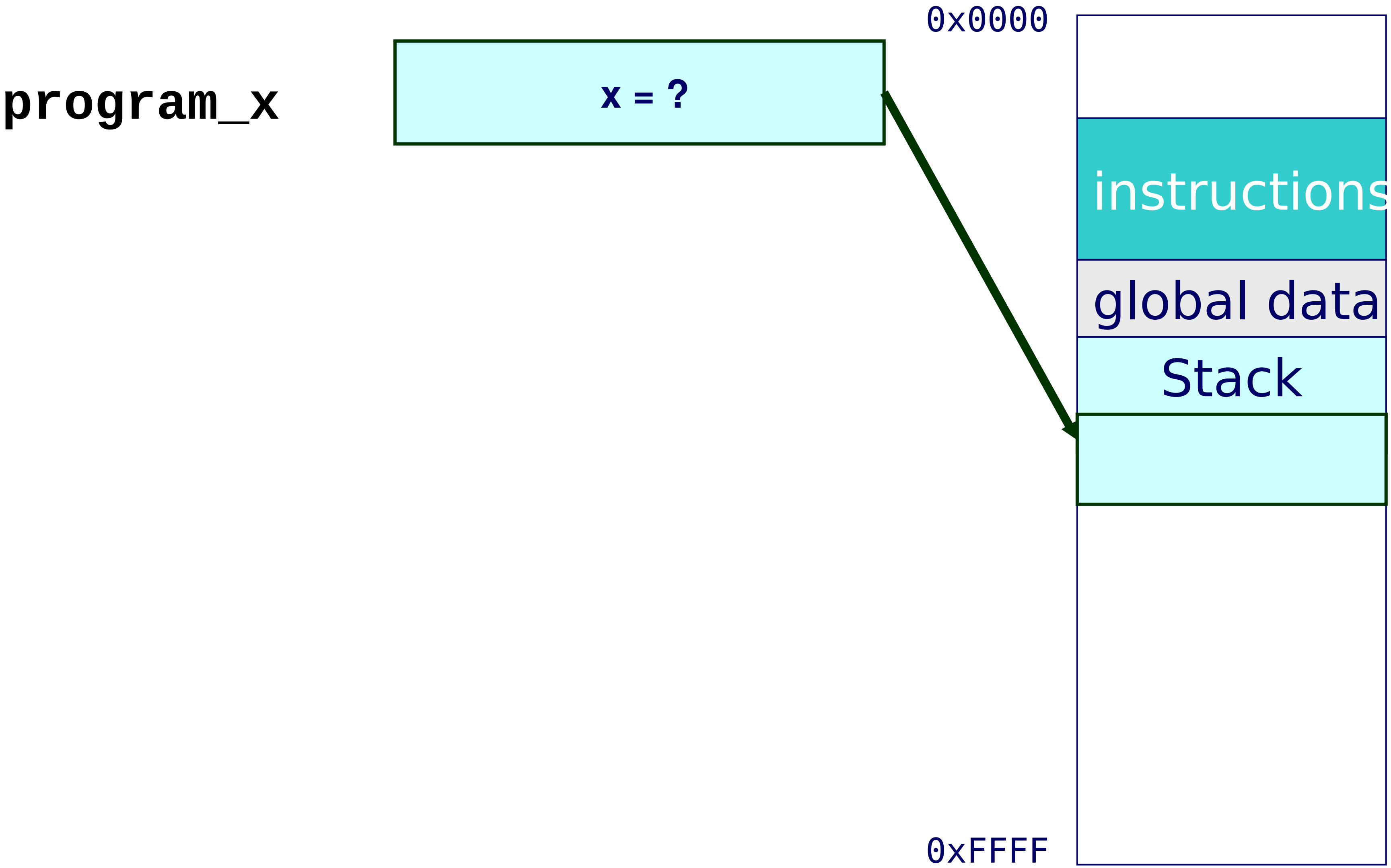
What if we want

- Memory area whose lifetime does not match any particular function?
- Memory area whose size is not known at compile time?

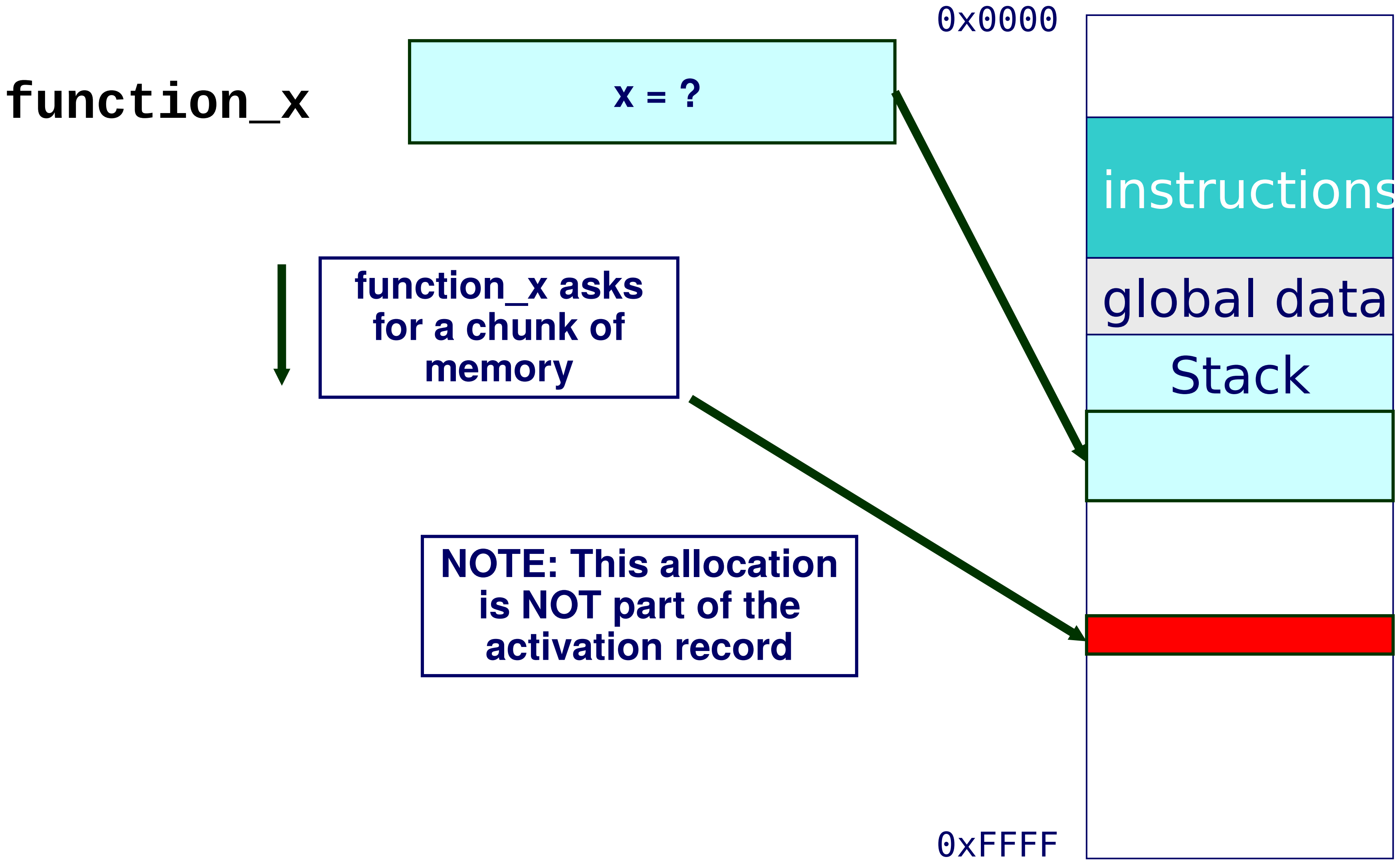
Two ways to “get memory”

- Declare a variable
  - Placed in global area or stack
  - Either “lives” forever or “live-and-die” with containing function
  - Size must be known at compile time
- Ask the run-time system for a “chunk” of memory dynamically

Allocating Space for Variables



Allocating Space for Variables



function_x

x = ?

function_x asks for a chunk of memory

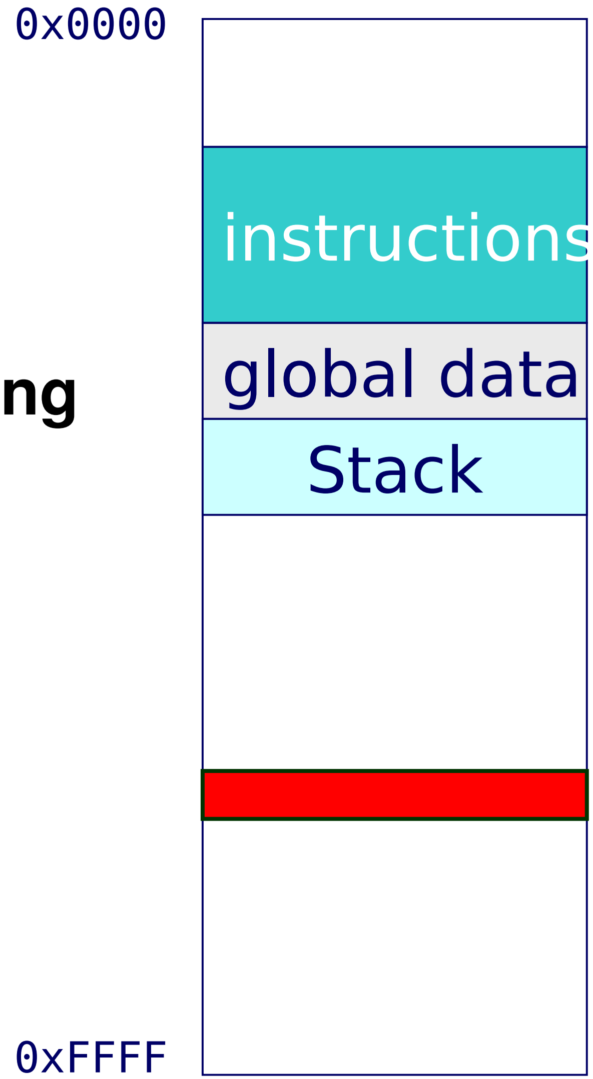
NOTE: This allocation is NOT part of the activation record

Allocating Space for Variables

After function returns, memory is still allocated

Request for dynamic chunks of memory performed using a call to the underlying runtime system (a system call).

- Commands: malloc and free

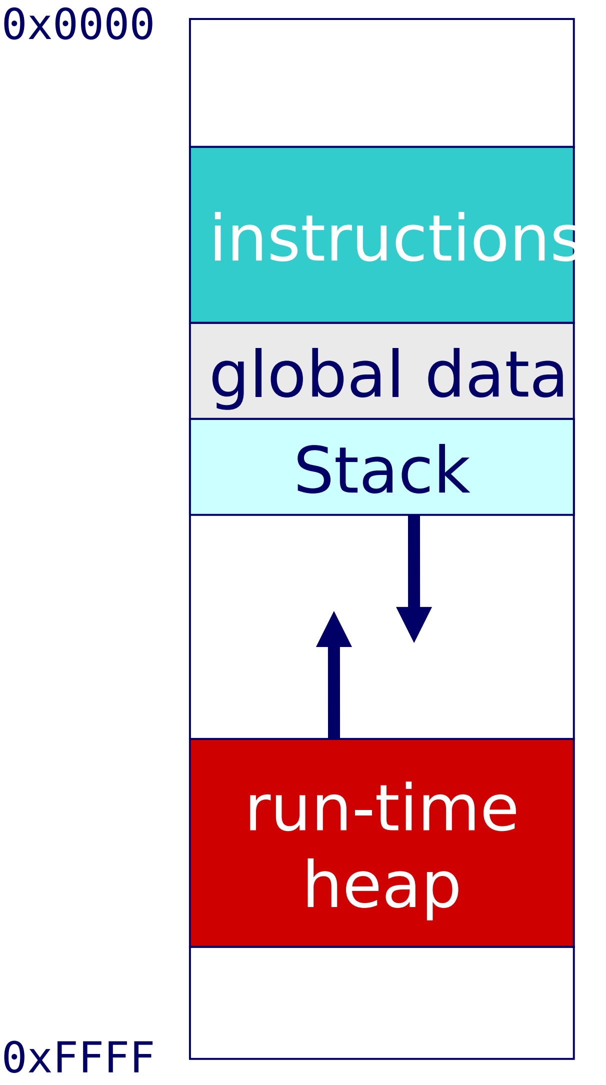


Dynamic Memory

Another area region of memory exists, it is called the heap

Dynamic request for memory are allocated from this region

Managed by the run-time system (actually, just a fancy name for a library that's linked with all C code)



malloc

The Standard C Library provides a function for dynamic memory allocation

```
void *malloc(int numBytes);
```

malloc() (and free()) manages a region of memory called the heap

- We’ll explain what a heap is later on and how it works

malloc() allocates a contiguous region of memory of size numBytes if there is enough free memory and returns a pointer to the beginning of this region

- Returns NULL if insufficient free memory

Why is the return type void*?

Using malloc

How do we know how many bytes to allocate?

Function

```
sizeof(type)
sizeof(variable)
```

Allocate right number of bytes, then cast to the right type

```
int *numbers = (int *)malloc(sizeof(int) * n);
```

free

Once a dynamically allocated piece of memory is no longer needed, need to release it

- Have finite amount of memory
- If don’t release, will eventually run out of heap space

Function:

```
void free(void*);
```

Example

```
int airbornePlanes;
struct flightType *planes;

printf(“How many planes are in the air?”);
scanf(“%d”, &airbornePlanes);

planes =
 (struct flightType*)malloc(sizeof(struct flightType) *

               airbornePlanes);
if (planes == NULL) {
  printf(“Error in allocating the data array.\n”);
  ...
}
planes[0].altitude = ...

…

free(planes);
```

If allocation fails, malloc returns NULL.

Note: Can use array notation or pointer notation.