

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;`

Will give us ~~segmentation fault~~

Array Storage

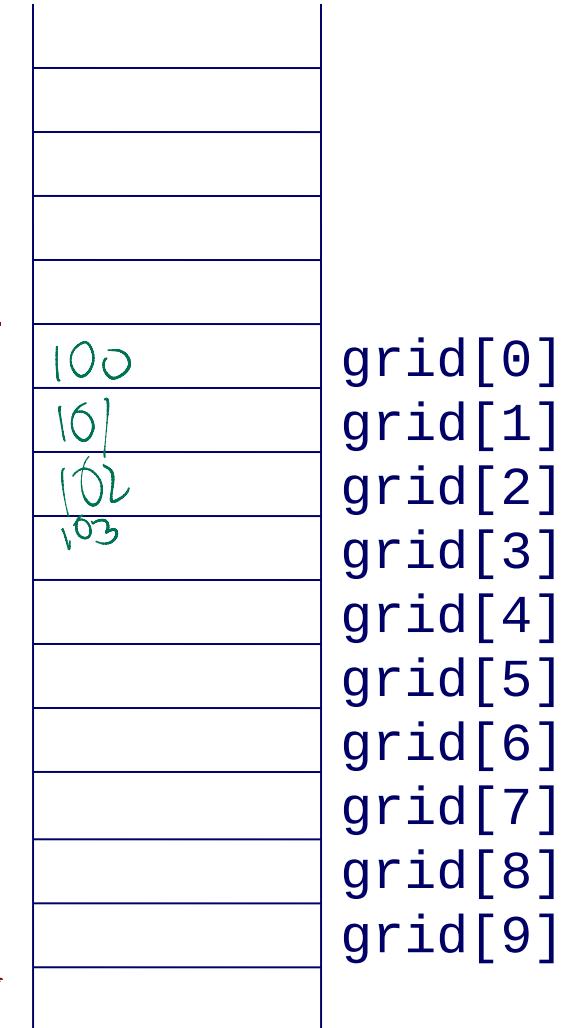
Elements of an array are stored sequentially in memory

char grid[10];



First element (grid[0]) is at lowest address of sequence

Knowing the location of the first element is enough to access any element



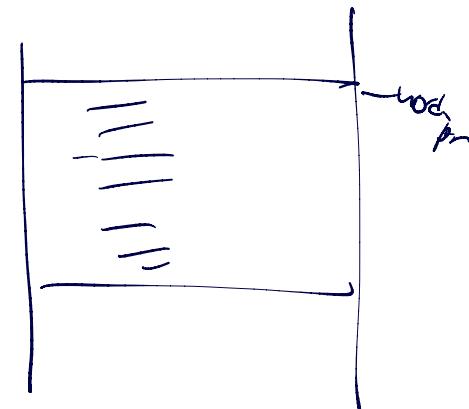
Arrays & Pointers

An array name is essentially a pointer to the first element in the array

1. `char word[10];` 10 bytes
2. `char *cptr;` 8 byte
3. `cptr = word; /* points to word[0] */`

Difference:

- Line 1 allocates space for 10 char items
- Line 2 allocates space for 1 pointer
- Can change value of cptr whereas cannot change value of word
 - Can only change value of `word[i]`



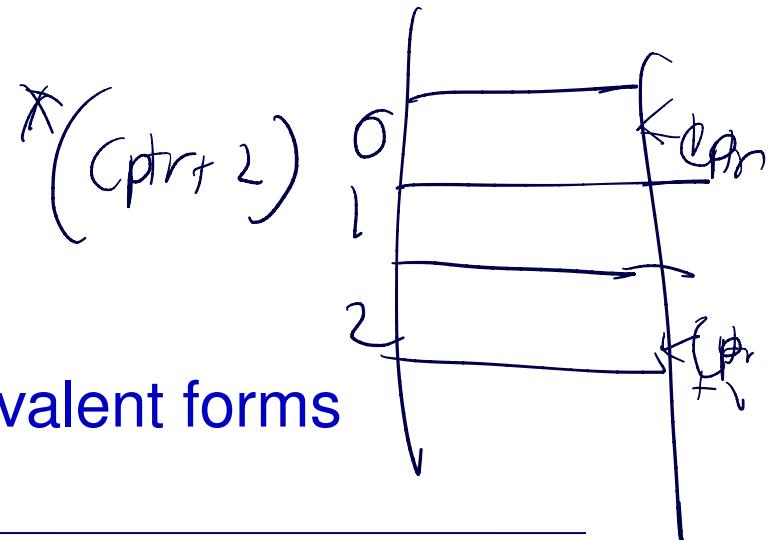
`word[0]=a chse`

Arrays & Pointers (Continued)

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

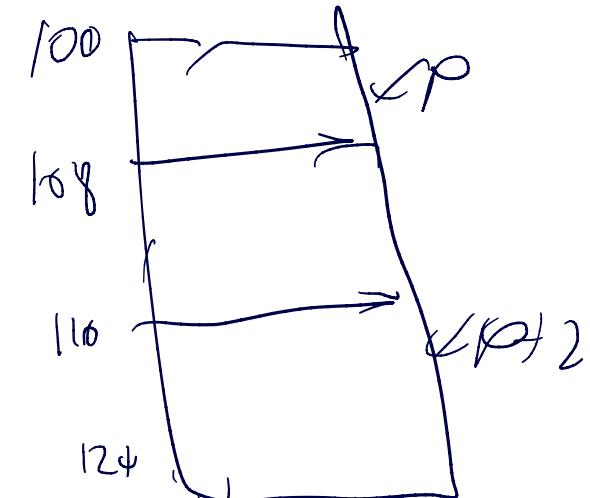
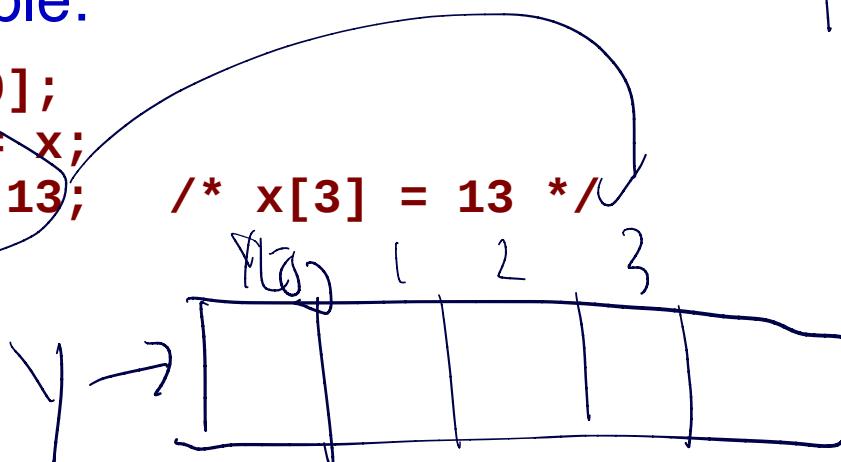
Address calculations with pointers are dependent on the size of the data the pointers are pointing to

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 */
```



Passing Arrays as Arguments

Arrays are passed as pointers (like passing the array itself by reference).

Array items are passed by value (No need to declare size of array for function parameters)

```
#include <stdio.h>

int *bogus;

void foo(int seqItems[], int item)
{
    seqItems[1] = 5;
    item = 5;
    bogus = &item;
}
```

```
int main(int argc, char **argv)
{
    int bunchOfInts[10];

    bunchOfInts[0] = 0;
    bunchOfInts[1] = 0;

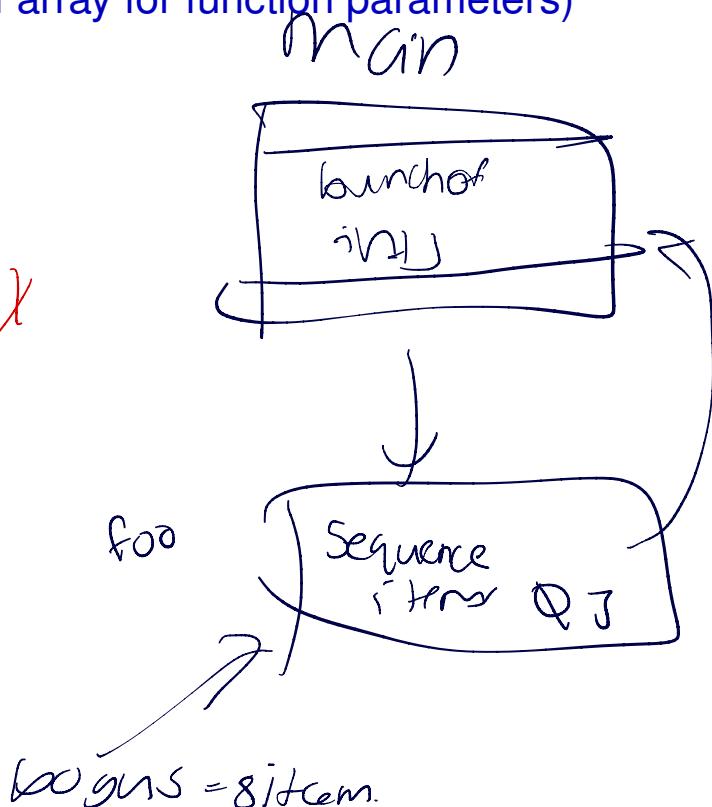
    foo(bunchOfInts, bunchOfInts[0]);

    printf("%d, %d\n", bunchOfInts[0], bunchOfInts[1]);
    printf("%d\n", *bogus);
}
```

Reference goes away

bogus = 8item.

returns random value



Common Pitfalls with Arrays in C

Overrun array limits

- There is no checking at run-time or compile-time to see whether reference is within array bounds.

```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

Declaration with variable size

- Size of array must be known at compile time.

```
void SomeFunction(int num_elements) {
    int temp[num_elements];
    ...
}
```

numbers
is an array

sizeOf(numbers)

size of(numbers)/size of(array)

↓
gives # of elements

dangers
-elements may not exist

Strings: Arrays of Characters

Allocate space for a string just like any other array:

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

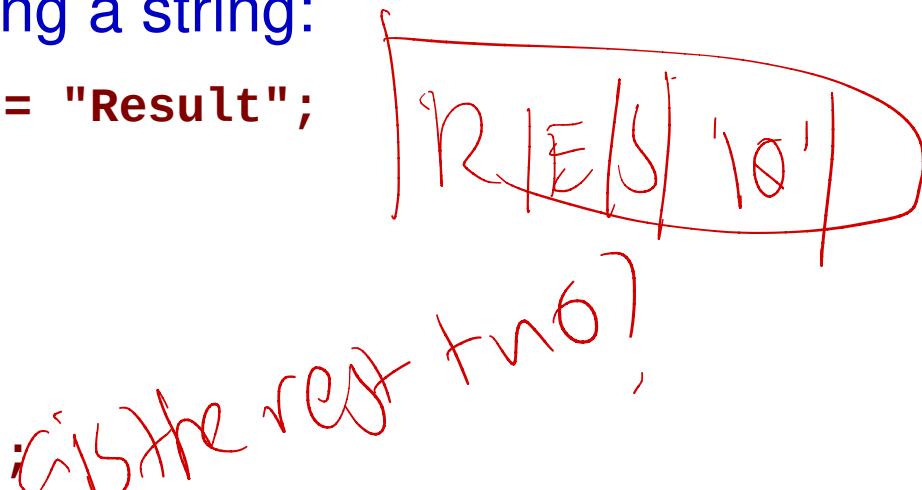
Each string should end with a '\0' character

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

Useful functions for Strings

Useful string related functions in standard C libraries

```
#include <string.h>
```

```
char *strcpy(char *d, char *s) Copy string s to d
```

```
int strcmp(s1, s2) Compare string s1 to s2
```

```
size_t strlen(s) Returns length of cs
```

Use “man” to learn more about these functions

- man strcpy

Special Character Literals

Certain characters cannot be easily represented by a single keystroke, because they

- correspond to whitespace (newline, tab, backspace, ...)
- are used as delimiters for other literals (quote, double quote, ...)

These are represented by the following sequences:

\n newline

\t tab

\b backspace

\\\ backslash

\' single quote

\\" double quote

\0nnn ASCII code *nnn* (in octal)

\xnnn ASCII code *nnn* (in hex)

*~~strcpy (char *s1, char *s2)~~

~~strcmp (s1, s2) compares s1 & s2~~

~~strlen (s) = returns length~~

Structures

A struct is a mechanism for grouping together related data items of different types.

Example: we want to represent an airborne aircraft

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

We can use a struct to group these data items together

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

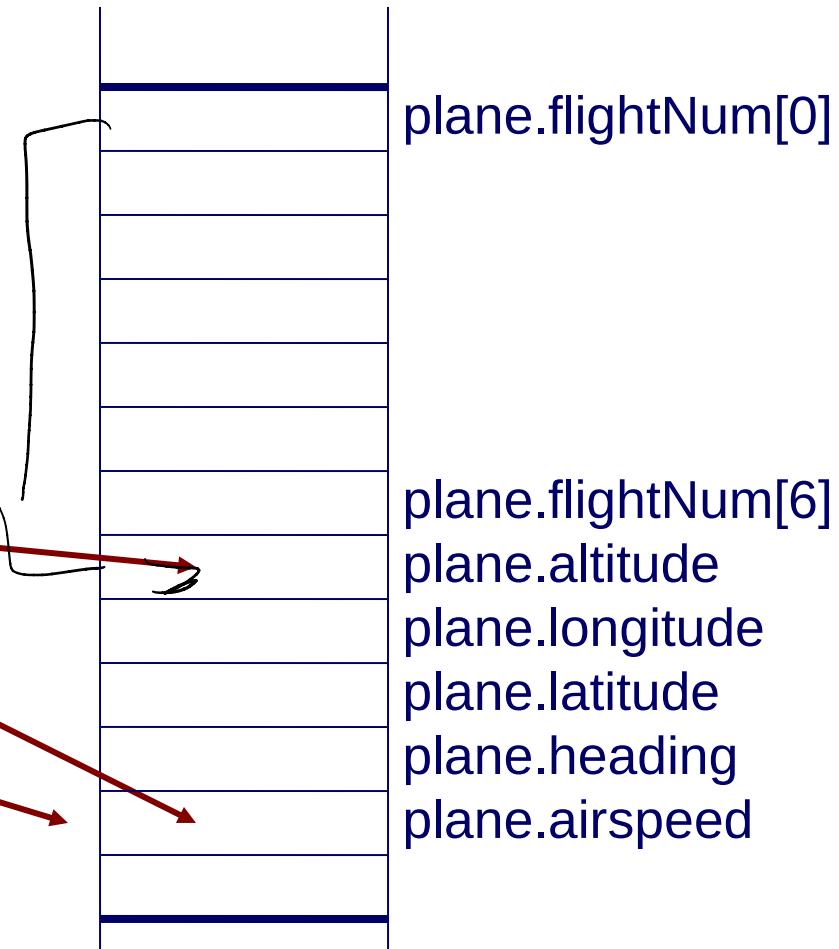
To allocate memory for a struct, we declare a variable using our new data type.

```
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~~ ^{don't have to do} 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;`~~

How high is it?

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

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

*Access a
struct member*

Passing Structs as Arguments

Unlike an array, a struct item is passed by value

Most of the time, you'll want to pass a pointer to a struct.

```
int Collide(struct flightType *planeA, struct flightType *planeB)
{
    if (planeA->altitude == planeB->altitude) {
        ...
    }
    else
        return 0;
}
```

Annotations:
- The code shows two pointers to `flightType` structures: `planeA` and `planeB`.
- The `->` operator is used to access the `altitude` member of each struct.
- Handwritten annotations explain the meaning of the operator:
 - A red circle is drawn around the `->` operator in `planeA->altitude`, with a handwritten note above it: "to de reference".
 - Another red circle is drawn around the `->` operator in `planeB->altitude`, with a handwritten note above it: "to get that altitude".

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**

Creates a
Stack Frame

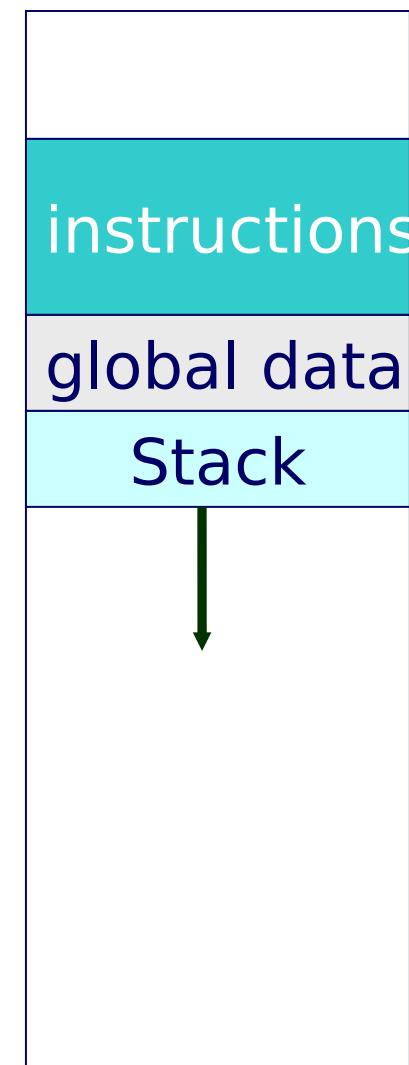
Allocating Space for Variables

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

0x0000



Allocating Space for Variables

Compute the sum of number from 1 to N

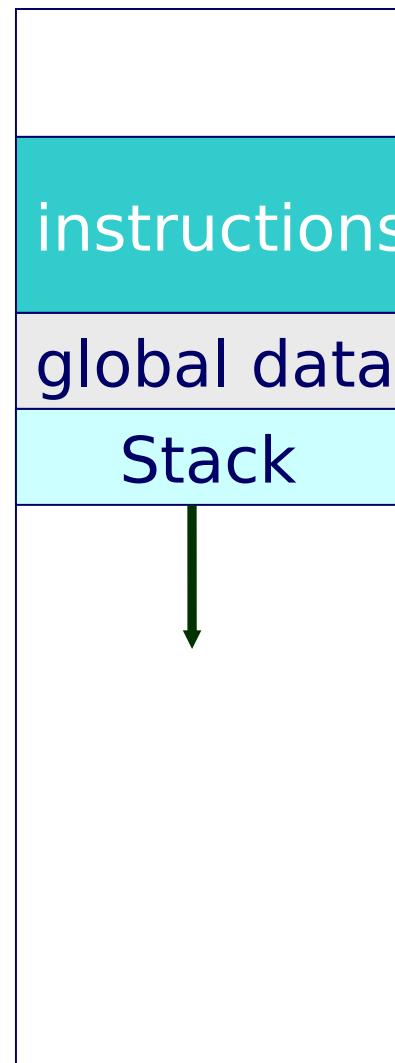
```
int summation(int n){  
    if(n == 0) return 0  
    else return n + summation(n-1);  
}  
...  
summation(5);
```

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

```
int n; int result;
```

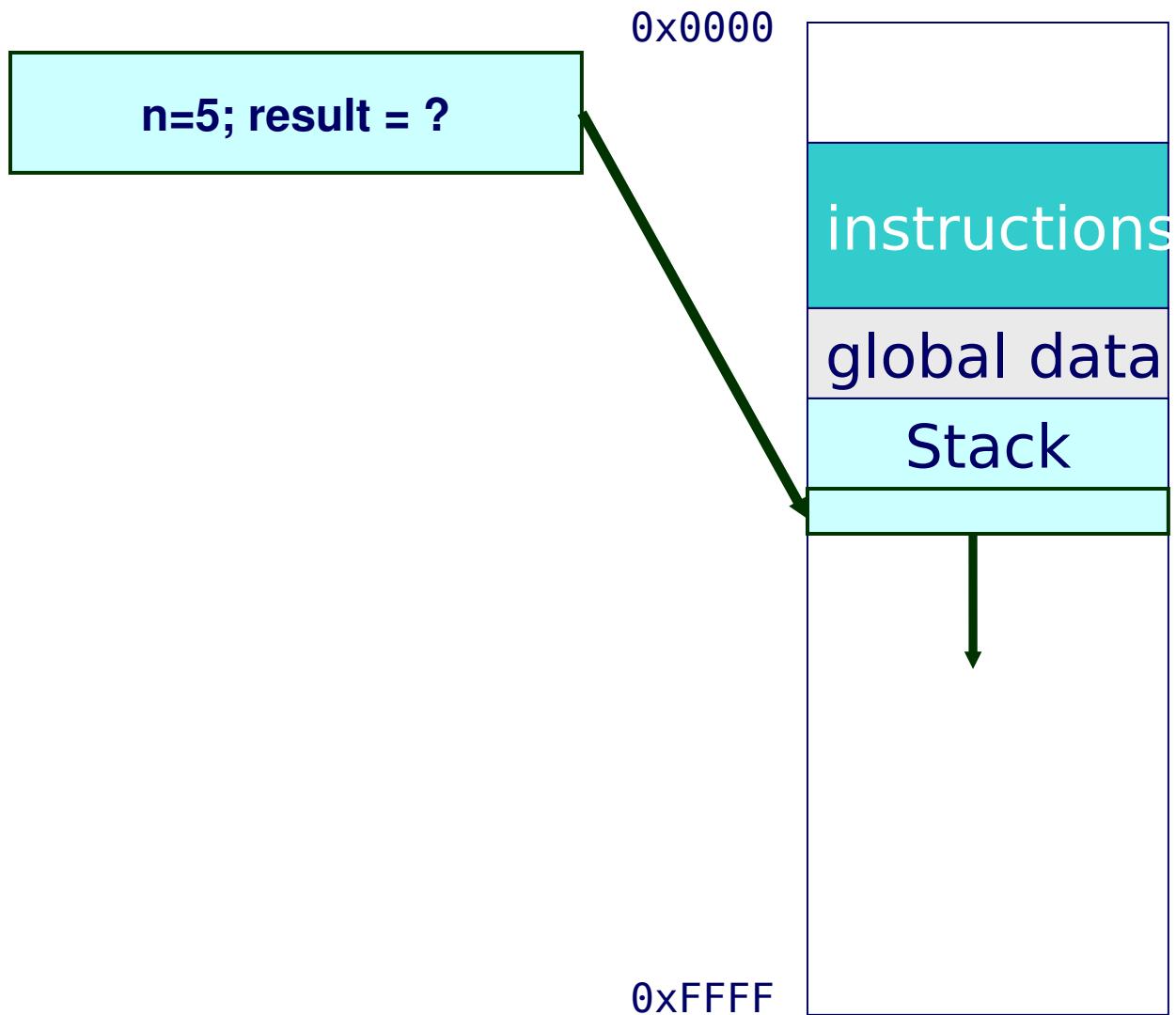
0x0000

0xFFFF

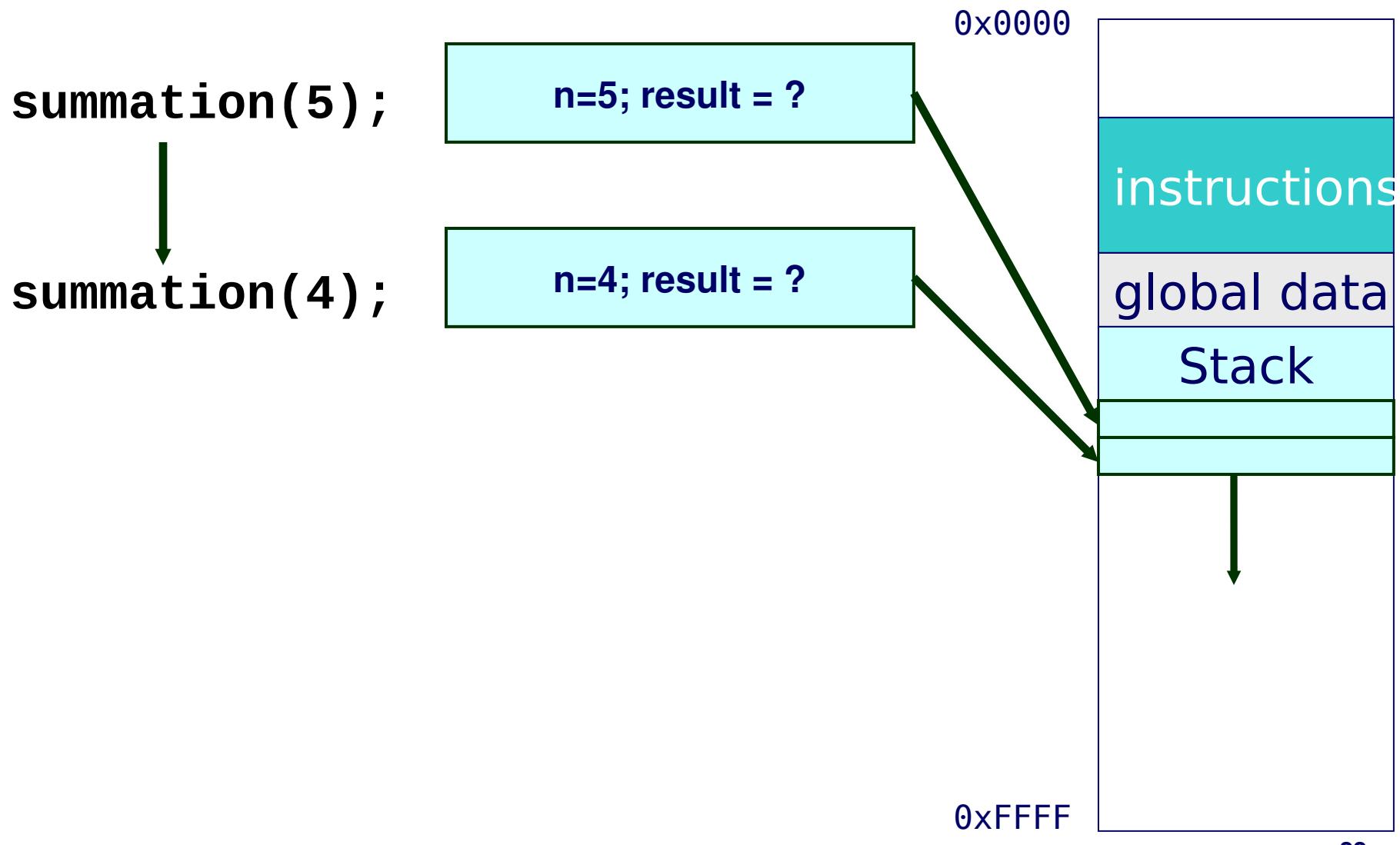


Allocating Space for Variables

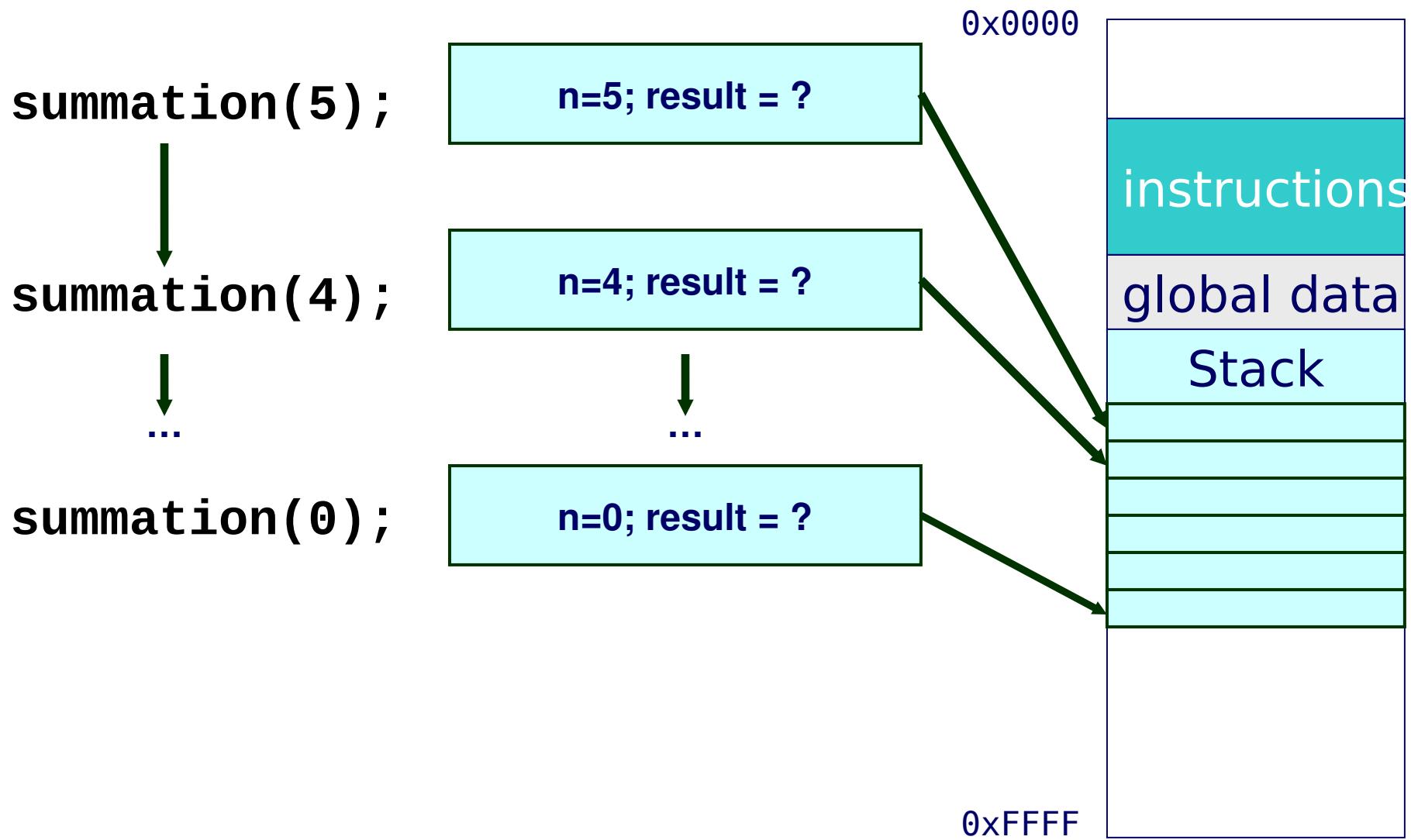
summation(5);



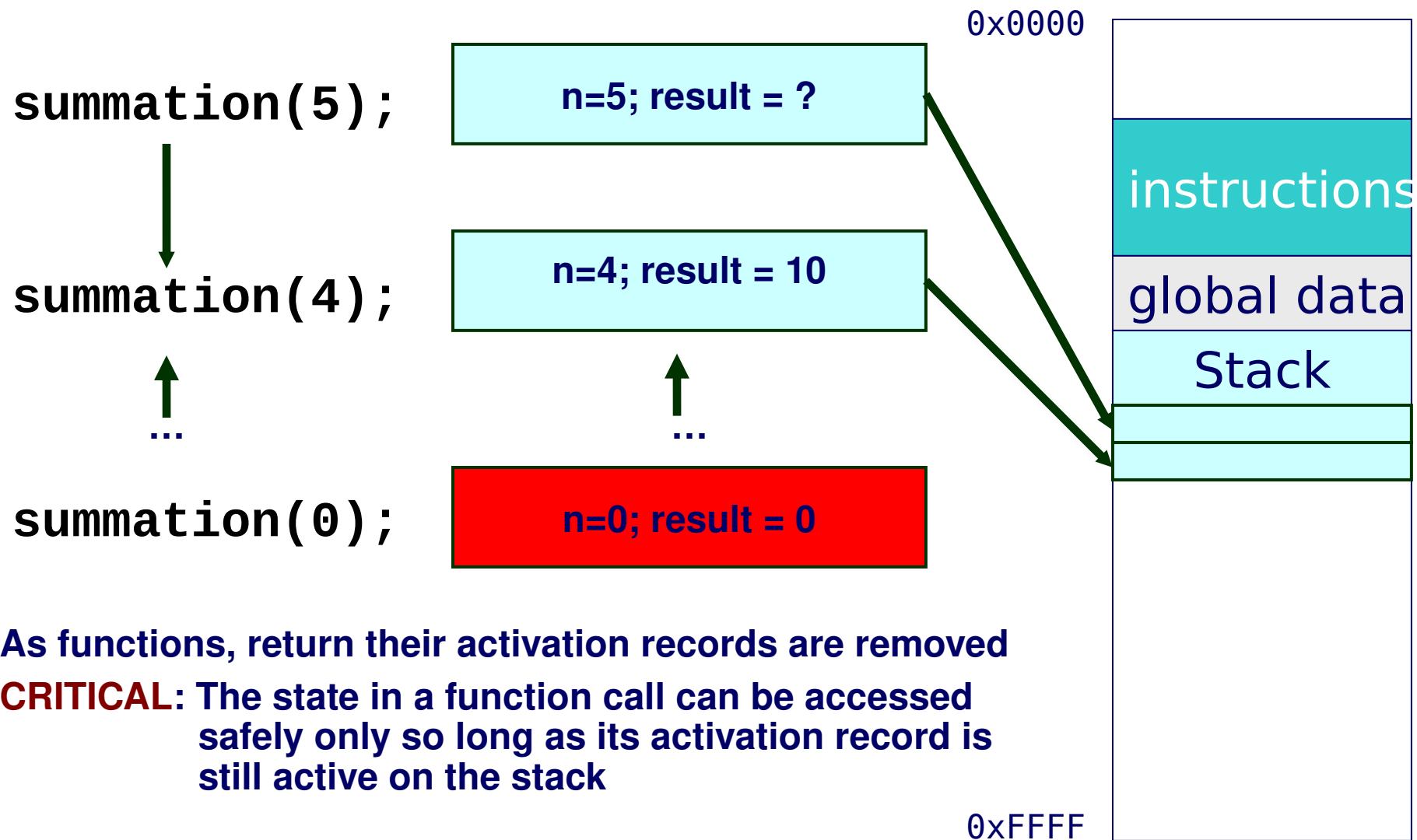
Allocating Space for Variables



Allocating Space for Variables



Allocating Space for Variables



As functions, return their activation records are removed

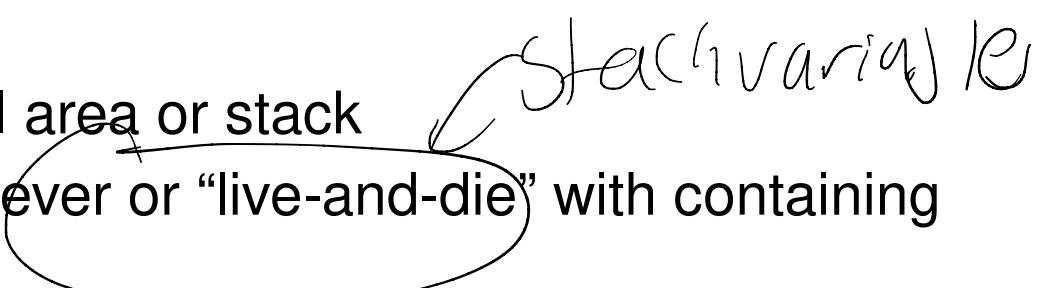
CRITICAL: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Dynamic Allocation

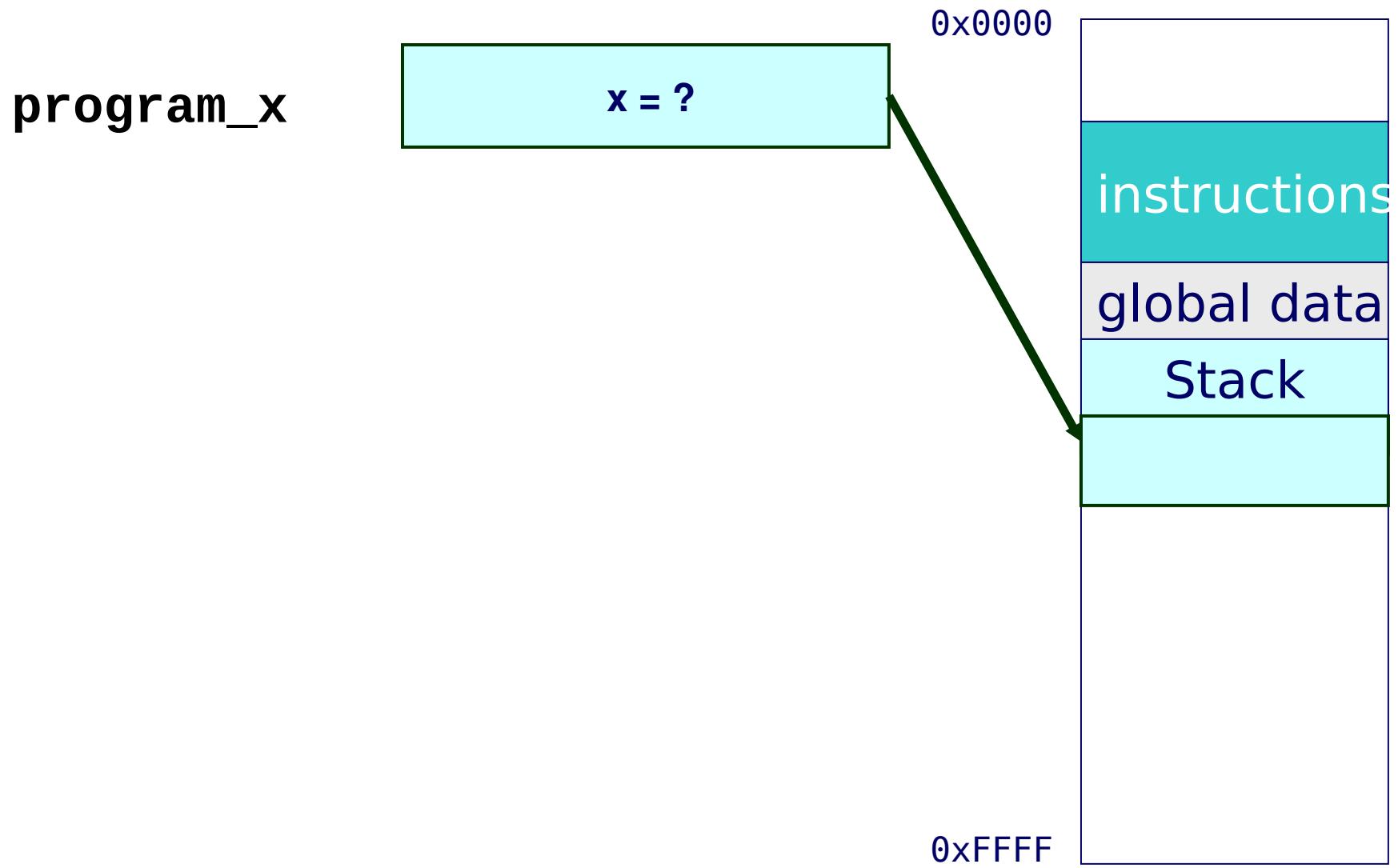
What if we want

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

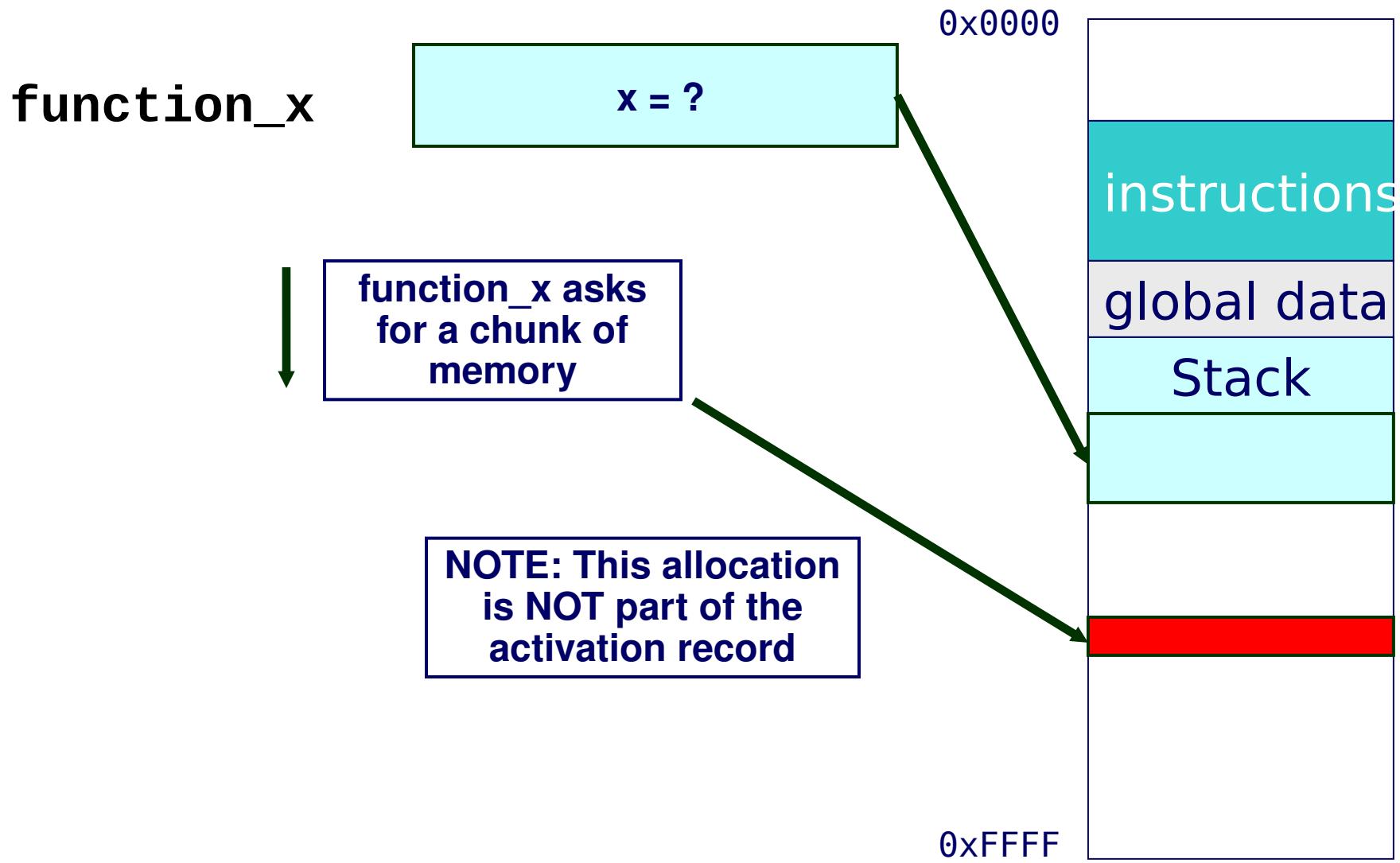
Two ways to “get memory”

- 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
- 
- A handwritten note "Stack variable" is written above the second bullet point of the first list item. A curved arrow points from the word "variable" in the note to the word "variable" in the list item.

Allocating Space for Variables



Allocating Space for Variables



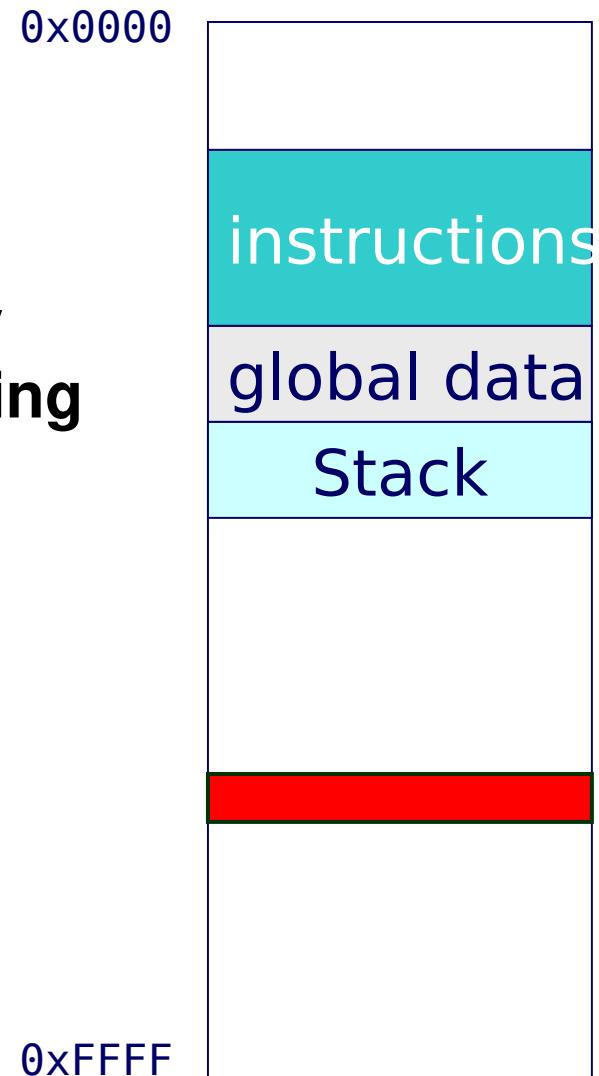
Allocating Space for Variables

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

memory
allocate



Dynamic Memory

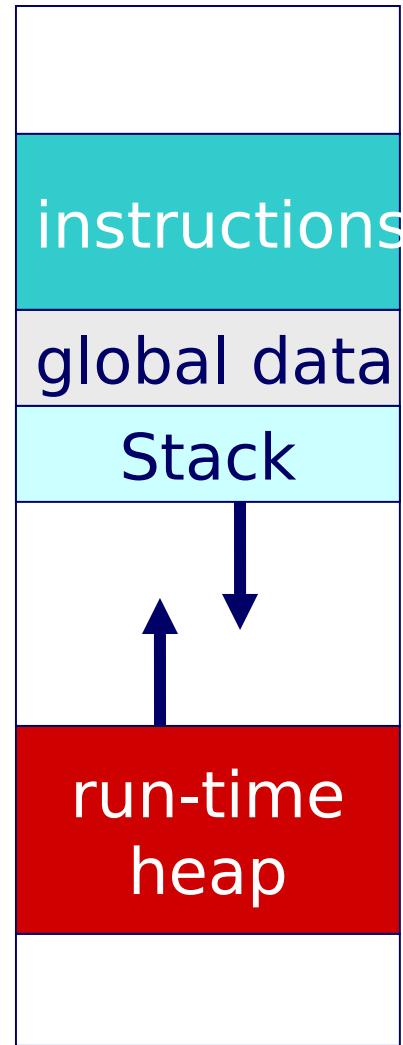
Another area region of memory exists,
it is called the **heap**

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)

0x0000

0xFFFF



malloc

The Standard C Library provides a function for dynamic memory allocation

```
void *malloc(int numBytes);  
    ^ void pointer generic can be cast to anything
```

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
(if there is)

- 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);`

type (w)

Recent
version
unnecessary

gets the size and multiply

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*);

Memory leak - memory that's free but told that you aren't

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.

Diagram illustrating memory allocation:

File I/O

A file is a contiguous set of bytes

- Has a name
- Can create, remove, read, write, and append

Unix/Linux supports persistent files stored on disk

- Access using system calls: open(), read(), write(), close(), creat(), lseek()
- Provide random access
- Section 2 of online manual (man)

C supports extended interface to UNIX files

- fopen(), fscanf(), fprintf(), fgetc(), fputc(), fclose()
- View files as streams of bytes
- Section 3 of online manual (man)

fopen

The fopen (pronounced "eff-open") function associates a physical file with a stream.

```
FILE *fopen(char* name, char* mode);
```

First argument: name

- The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.

Second argument: mode

- How the file will be used:
 - "r" -- read from the file
 - "w" -- write, starting at the beginning of the file
 - "a" -- write, starting at the end of the file (append)

fprintf and fscanf

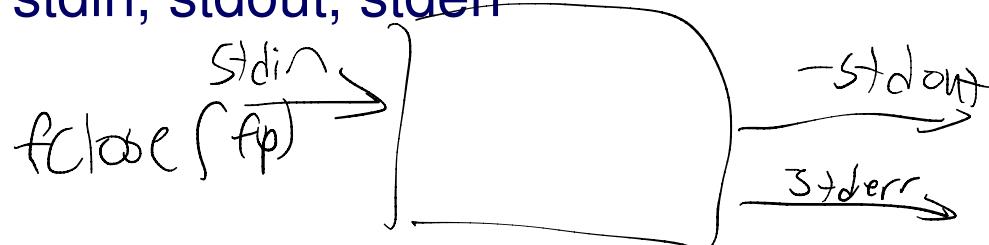
Once a file is opened, it can be read or written using fscanf() and fprintf()

These are just like scanf() and printf() except with an additional argument specifying a file pointer

- `fprintf(outfile, "The answer is %d\n", x);`
- `fscanf(infile, "%s %d/%d/%d %f", &name, &bMonth, &bDay, &bYear, &gpa);`

When started, each executing program has three standard streams open for input, output, and errors

- `stdin, stdout, stderr`



fprintf and fscanf

Once a file is opened, it can be read or written using fscanf() and fprintf()

These are just like scanf() and printf() except with an additional argument specifying a file pointer

- `fprintf(outfile, "The answer is %d\n", x);`
- `fscanf(infile, "%s %d/%d/%d %f",`
`&name, &bMonth, &bDay, &bYear, &gpa);`

When started, each executing program has three standard streams open for input, output, and errors

- `stdin, stdout, stderr`