211: Computer Architecture
Spring 2017

Instructor: Prof. Santosh Nagarakatte

Topics:

- Introduction to Computer Architecture
- C Programming

Architecture Trends: Moore's law

Gordon Moore was an Intel Engineer

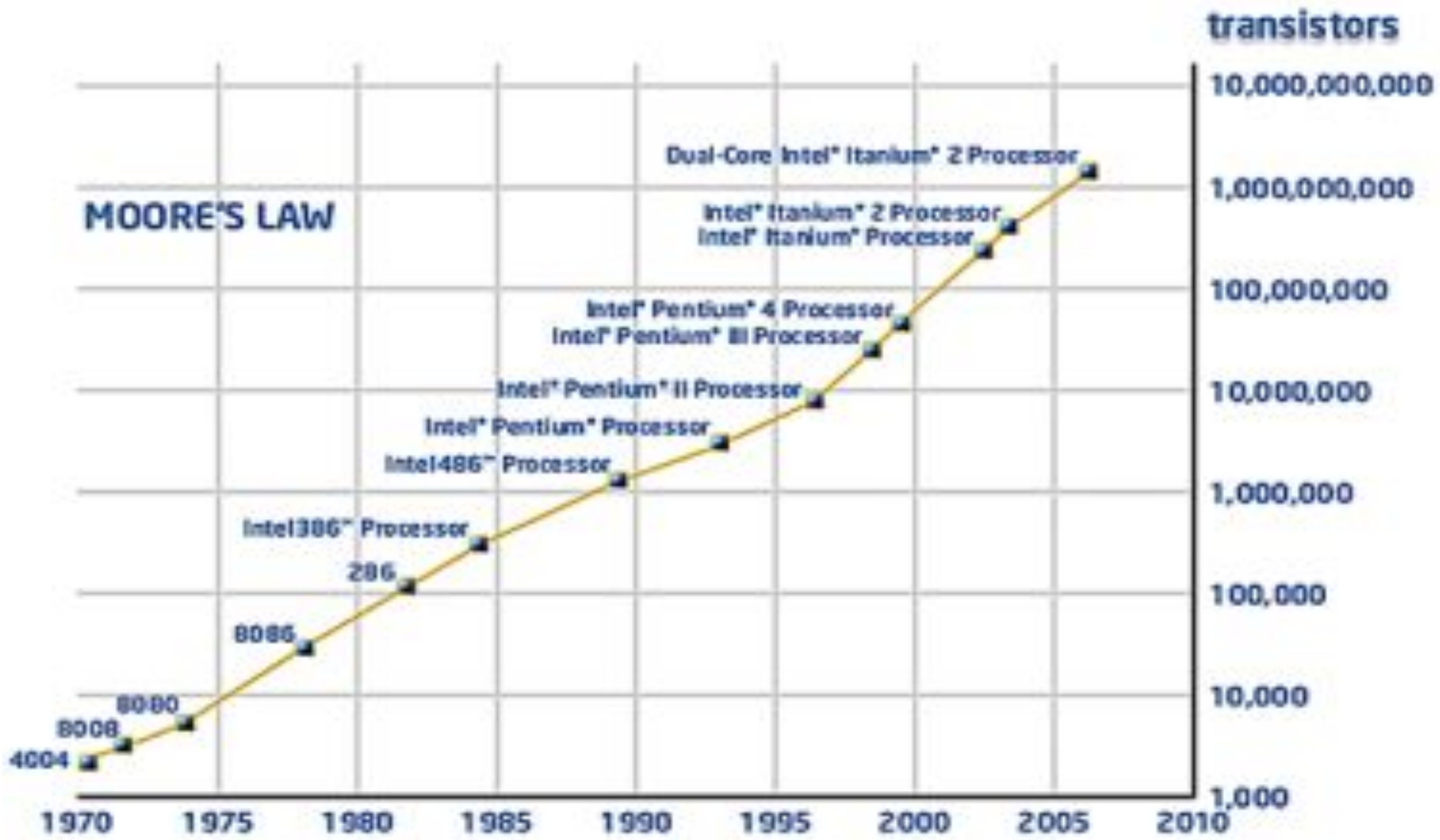
An observation about improvements in hardware

No of transistors on a chip double every 18 months

Exponential growth seen in other hardware dimensions

- Processor speed: 2x every 18 months
- Memory capacity: 2x every 2 years
- Disk capacity: 2x every year

Number of Transistors on a Chip

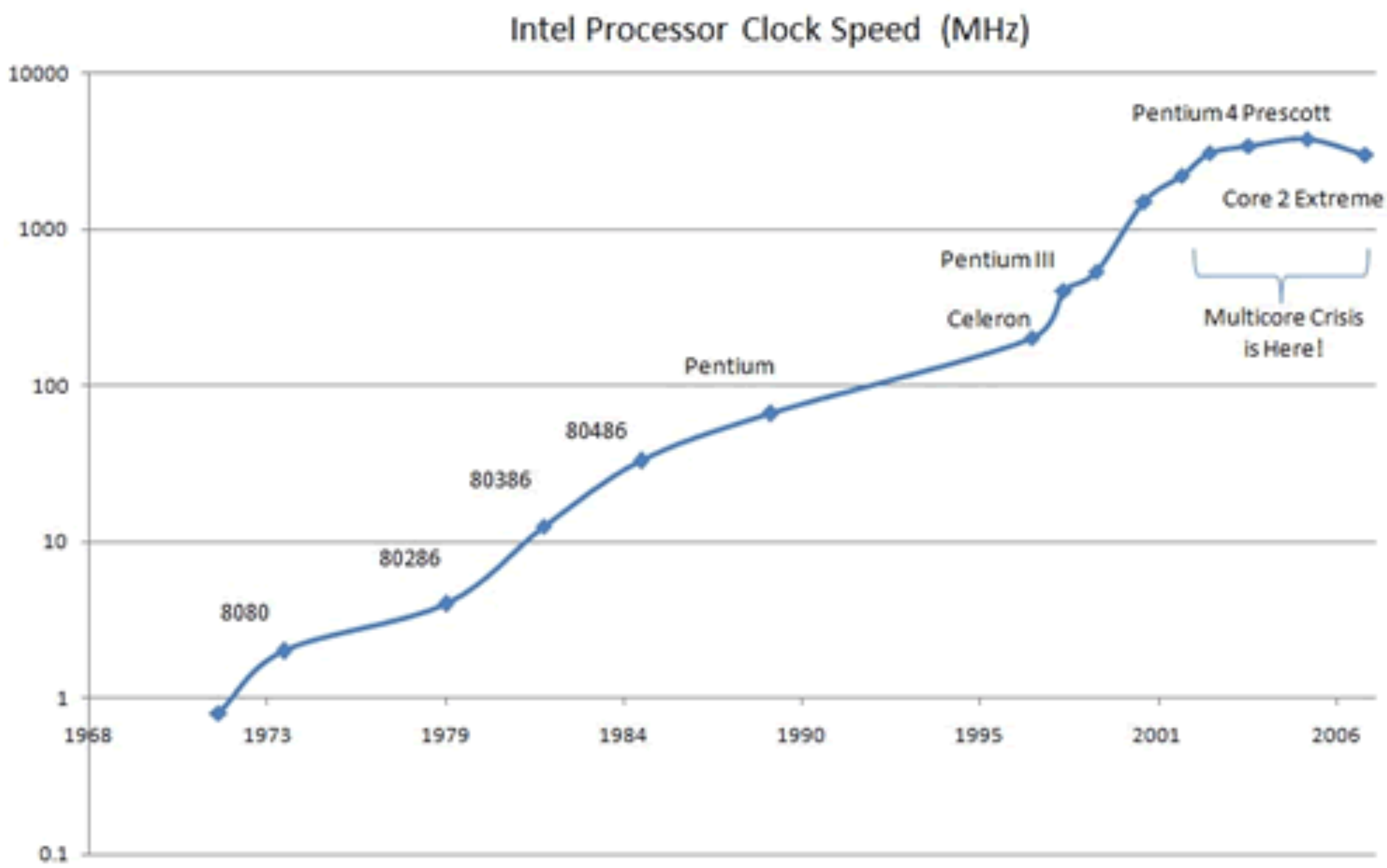


Any guess on the number of transistors in the latest Intel chip in your laptops?

Any guess on the number of transistors in the latest Intel chip in your laptops?

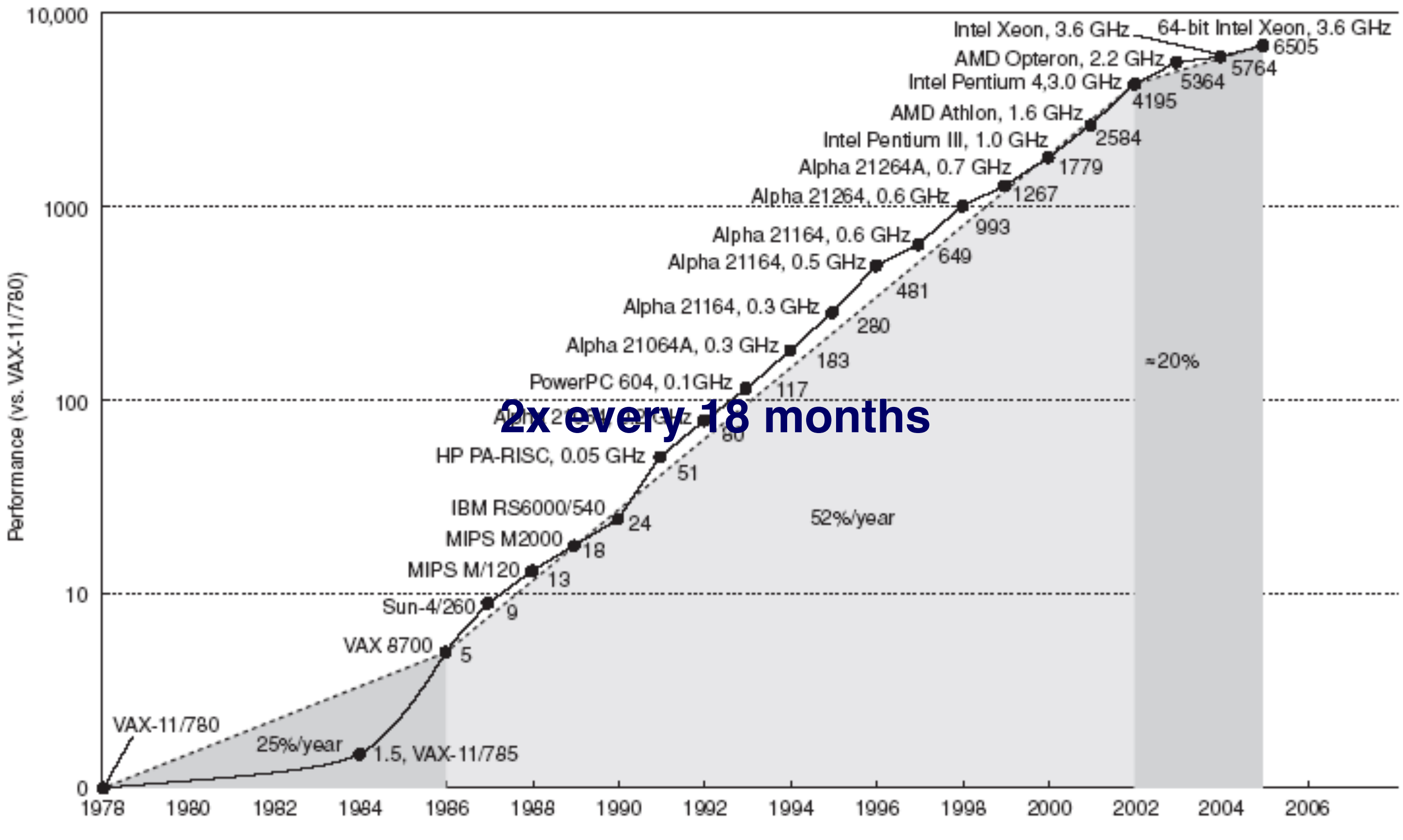
Intel Broadwell Xeon chip has 7.2billion transistors!

Clock speed

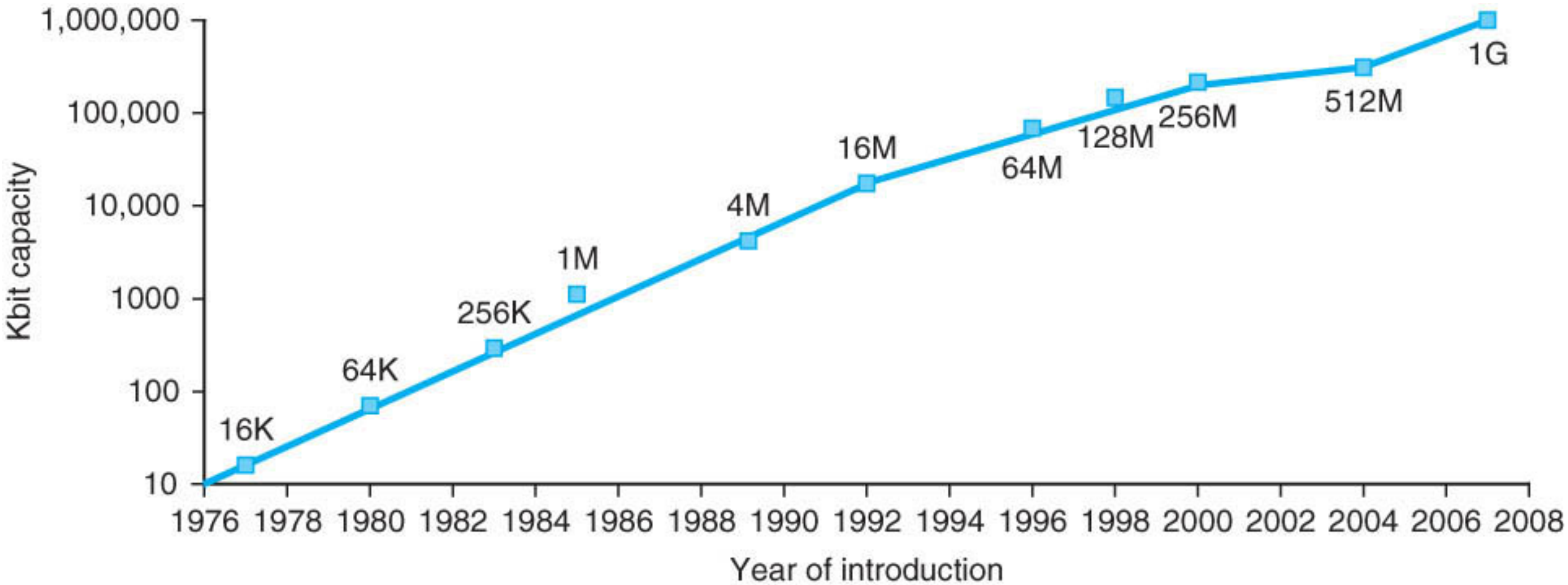


Intel Processor Clock Speed (MHz)
10000
1000
100
10
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80286
80386
80486
Pentium
Celeron
Pentium III
Pentium 4 Prescott
Core 2 Extreme
Multicore Crisis
is Here!

Processor Performance

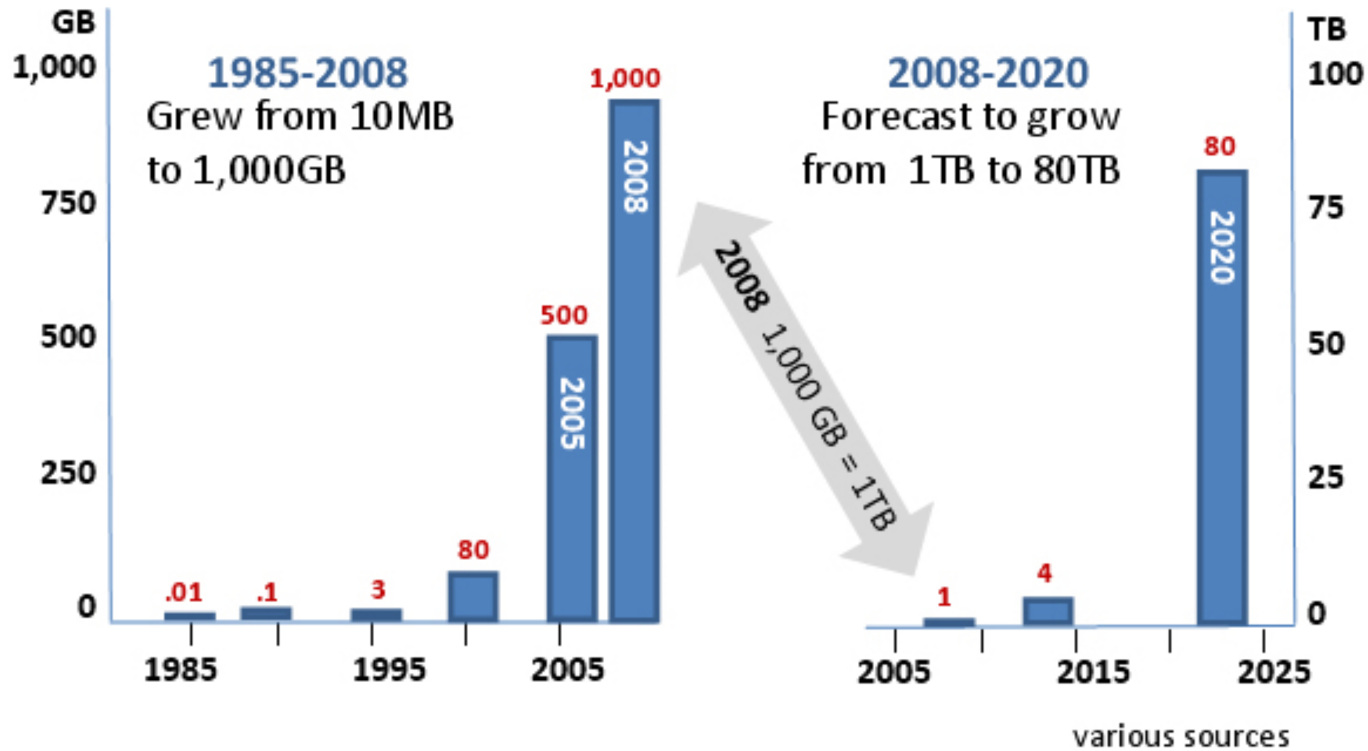


Memory Capacity

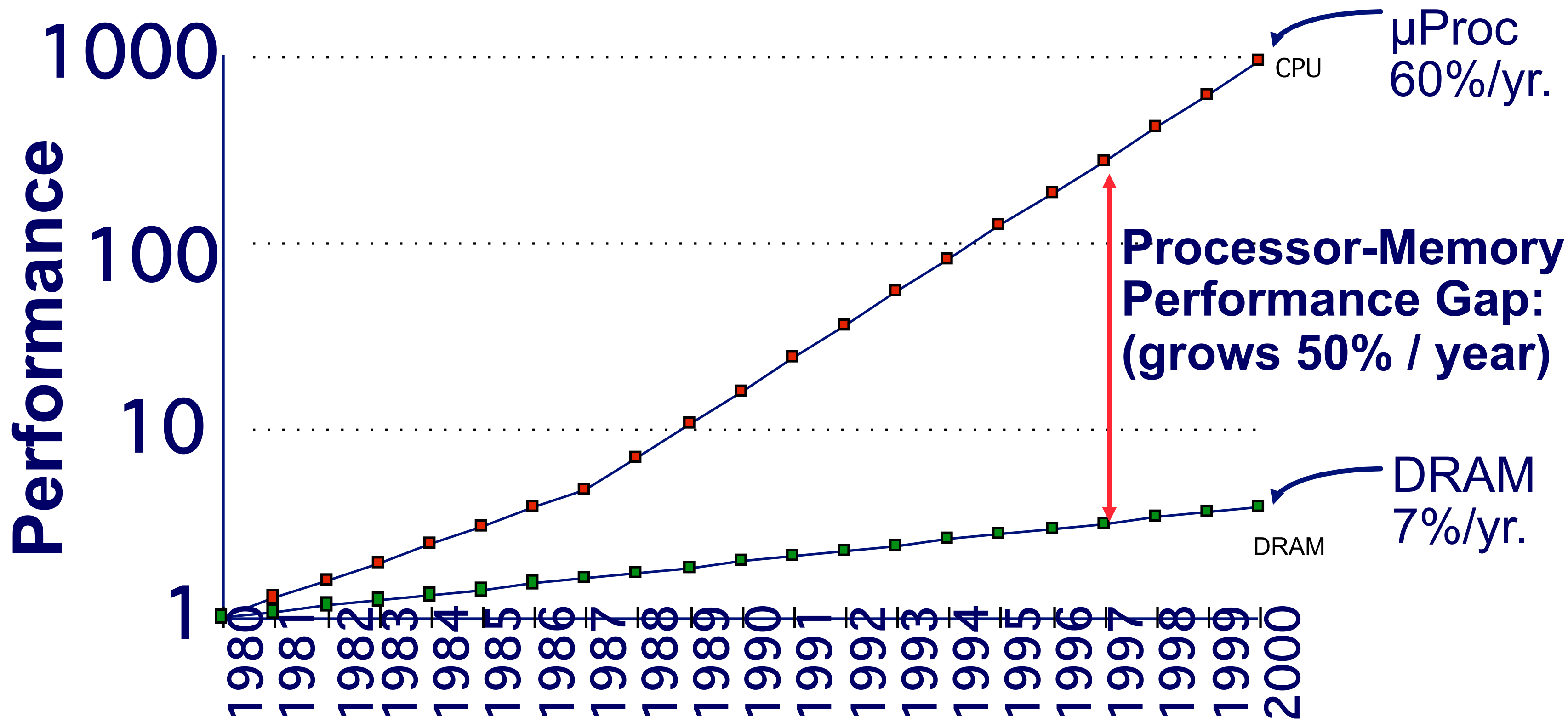


Now laptops have 16GB RAM

Disk Capacity



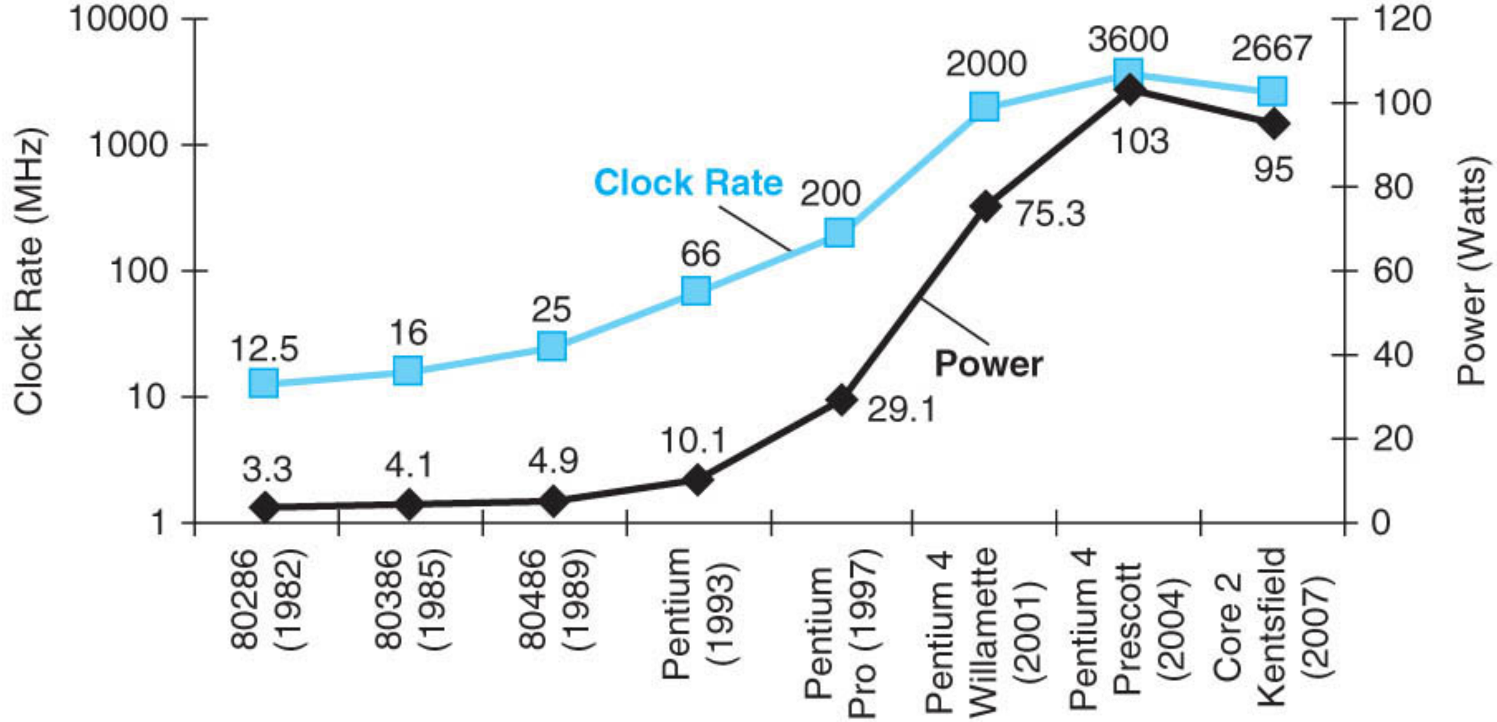
CPU/Memory Performance Gap



Implication: memory hierarchy

- registers, caches

“Power Wall”



10000
1000
100
10
1
Clock Rate (MHz)
120
100
80
60
40
20
0
Power (Watts)
Clock Rate
Power
12.5
16
25
66
200
2000
3600
2667
3.3
4.1
4.9
10.1
29.1
75.3
103
95
80286 (1982)
80386 (1985)
80486 (1989)
Pentium (1993)
Pentium Pro (1997)
Pentium 4 Willamette (2001)
Pentium 4 Prescott (2004)
Core 2 Kentsfield (2007)

Summary

Today's systems are complex with various components

Faster processors and systems

- Enables new domains of applications
- Imagine the running your favorite game on a Pentium machine.

Understanding the systems crucial

- To program these systems
- To make pragmatic performance/power/energy tradeoffs
- Address various walls --- memory wall, power wall, etc

Iclicker: Pop Quiz

Hardware directly understands

A: Any C program

B: Any Java program

C: Understands only 0's and 1's

D: None of the above

Iclicker: Pop Quiz

In a Von Neumann model

A: The program is a circuit

B: The program executes an arbitrary instruction when its operands become available

C: The program is stored in memory and the program counter determines which instruction executes next

D: None of the above

Iclicker: Pop Quiz

Moore's law states

A: The number of smartphones increase every year

B: The number of IOT devices increase every year.

C: The number of transistors double every 18 months.

D: Number of processors in a data center doubles every 18 months

Iclicker: Pop Quiz

If you have two processor P and Q. P is 1.2Ghz and Q is 2.4 Ghz, then

A: P executes every program twice as fast as Q.

B: Q executes every program twice as fast as P.

C: If the amount of work done in each cycle is the same, then Q executes programs twice as fast as P.

D: Both P and Q execute at the same speed.

INTRODUCTION TO C PROGRAMMING

Introduction to C

TAs will also cover C in more details in recitations

- Will also help you with machine/compilation logistics

Learning C

- Is no big deal; you already know Java
- Start by coding and testing small programs
- Learn how to use a debugger!
  - TAs will help

Why Learn C?

You are learning to be a computer scientist

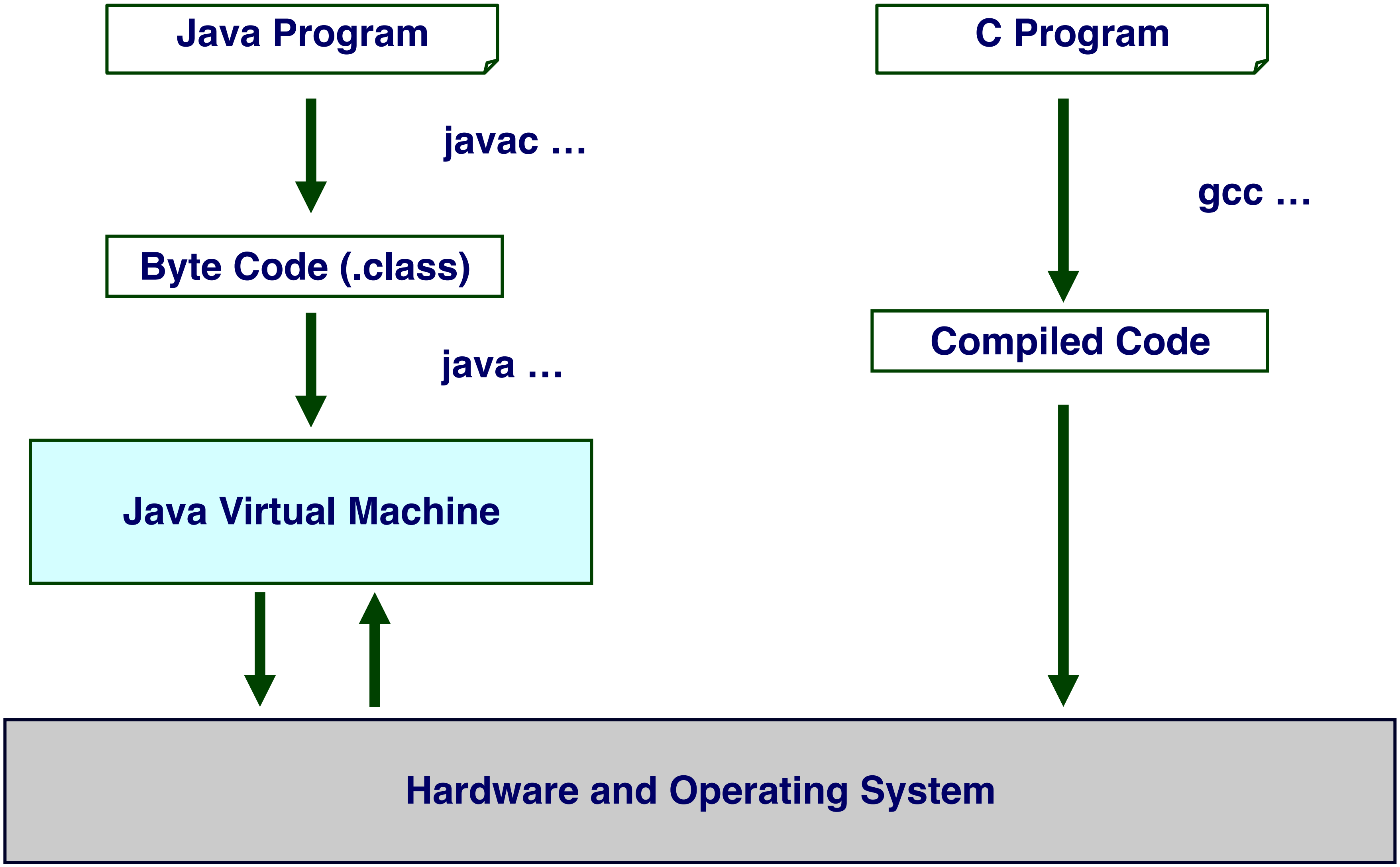
- Languages are just tools
- Choose tool appropriate to the task

Current task: learning computer architecture and how programs written in high-level language runs on computers

- C closer to machine so easier to see mapping

It's fun

Comparison with Java



Anatomy of a C Program

```c
#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

Comments

Begins with /* and ends with */

Can span multiple lines.

Cannot have a comment within a comment

- Example:

  “my/*don't print this*/string”

- would be printed as:

  my/*don't print this*/string

Comments are critical

- How much and where is an art

Variable Declarations

Variables are used as names for data items

Each variable has a type, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.)

```
int counter;

int startPoint;
```

Variables can be global or local

global: declare outside scope of any function
accessible from anywhere

local: declare inside scope of a function
accessible only from inside of the function

Basic Data Types

Keyword	Data Type	Examples
char	individual characters	‘a’, ‘b’, ‘\t’, ‘\n’
int	integers	-15, 0, 35
float	real numbers	-23.6, 0, 4.56
double	real numbers with double precisions	-23.6, 0, 4.56

Modifiers

- short, long: control size/range of numbers
- signed, unsigned: include negative numbers or not

Arithmetic Operators

Symbol	Operation	Usage	Assoc
*	multiply	x * y	l-to-r
/	divide	x / y	l-to-r
%	modulo	x % y	l-to-r
+	addition	x + y	l-to-r
-	subtraction	x - y	l-to-r

* / % have higher precedence than + -

Rule of thumb: remember only a few precedence rules

Use () for everything else

Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression

Symbol	Operation	Usage	Assoc
++	postincrement	x++	r-to-l
--	postdecrement	x--	r-to-l
++	preincrement	++x	r-to-l
--	predecrement	--x	r-to-l

Pre: Increment/decrement variable before using its value

Post: Increment/decrement variable after using its value

Be careful when using these operators!

Relational Operators

Symbol	Operation	Usage	Assoc
>	greater than	x > y	l-to-r
>=	greater than or equal	x >= y	l-to-r
<	less than	x < y	l-to-r
<=	less than or equal	x <= y	l-to-r
==	equal	x == y	l-to-r
!=	not equal	x != y	l-to-r

Result is 1 (TRUE) or 0 (FALSE)

Don't confuse equality (==) with assignment (=)

Logic Operators

Symbol	Operation	Usage	Assoc
!	logical NOT	!x	r-to-l
&&	logical AND	x && y	l-to-r
||	logical OR	x || y	l-to-r

Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero)

Result is 1 (TRUE) or 0 (FALSE)

Bit Operators

Symbol	Operation	Usage	Assoc
~	complement	~x	r-to-l
&	bit AND	x & y	l-to-r
l	bit OR	x l y	l-to-r

Operate on bits of variables or constants

For example:

- ~0101 = 1010
- 0101 & 1010 = 0000
- 0101 l 1010 = 1111

Expressions and Assignments

Expression = "a computation" with a result

- (x + y) * z
- Be careful of type conversion!

  int x, z; float y;

  the result of the expression (x + y) * z will have what type?

Assignment

- x = (x + y) * z;
- The assignment statement itself is an expression and has a value. In this case, it's the value assigned to x.

Control Statements

Conditional

- if else
- switch

Iteration (loops)

- while
- for
- do while

Specialized “go-to”

- break
- continue

The if Statement

```
if (expression-1) {statements-1}
else if (expression-2) {statements-2}
else if (expression-n-1) {statements-n-1}|
else {statements-n}
```

Evaluates expressions until find one with non-zero result

- executes corresponding statements

If all expressions evaluate to zero, executes statements for "else" branch

The switch Statement

```
switch(expression) {
  case const-1: statements-1;
  case const-2: statements-2;
  default: statements-n;
}
```

Evaluates expression; results must be integer

Finds 1st “case” with matching contant

- Executes corresponding statements
- Continue executing until encounter a break or end of switch statement

“default” always matches

The switch Statement (Example)

```
int fork;
…
switch(fork) {
     case 1:
         printf("take left'');
     case 2:
         printf("take right");
         break;
     case 3:
         printf("make U turn");
         break;
     default:
         printf("go straight");
}
```

Loops

Statement	Repeats set of statements
while (expression) {…}	zero or more times, while expression $\neq$ 0, compute expression before each iteration
do {...} while (expression)	one or more times, while expression $\neq$ 0 compute expression after each iteration
for (start-expression; cond-expression; update-expression) {…}	zero or more times while cond-expression $\neq$ 0 compute start-expression before 1st iteration compute update-expression after each iteration

Specialized Go-to's

break

- Force immediate exit from switch or loop
- Go-to statement immediately following switch/loop

continue

- Skip the rest of the computation in the current iteration of a loop
- Go-to evaluation of conditional expression for execution of next iteration

Specialized Go-to's (Example)

What does the following piece of code do?

```
int index = 0;
int sum = 0;
while ((index >= 0) && (index <= 20))
{
        index += 1;
        if (index == 11) break;
        if ((index % 2) == 1) continue;
        sum = sum + index;
}
```

Functions

Similar to Java methods

Components:

- Name
- Return type
  - void if no return value
- Parameters
  - pass-by-value
- Body
  - Statements to be executed
  - return forces exits from function and resumes execution at statement immediately after function call

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

Function Calls

Function call as part of an expression

- x + Factorial(y)
- Arguments evaluated before function call
  - Multiple arguments: no defined order or evaluation
- Returned value is used to compute expression
- Cannot have a void return type

Function call as a statement

- Factorial(y);
- Can have a void return type
- Returned value is discarded (if there is one)

Function Prototypes

Can declare functions without specifying implementation

- int Factorial(int)
  - Can specify parameter names but don’t have to
  - This is called a function signature

Declarations allow functions to be “used” without having the implementation until link time (we’ll talk about linking later)

- Separate compilation
  - Functions implemented in different files
  - Functions in binary libraries
- Signatures are often given in header files
  - E.g., stdio.h gives the signatures for standard I/O functions

Input and Output

Variety of I/O functions in C Standard Library

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

```c
printf("%d\n", counter);
```

- String contains characters to print and formatting directives for variables
- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)

```c
scanf("%d", &startPoint);
```

- String contains formatting directives for parsing input
- This call says to read a decimal integer and assign it to the variable startPoint. (Don't worry about the & yet.)

iClicker Pop Quiz

What does this code do for n=10?

```
void foo (int n){
  int  i = 0;
   for(i = 1; i < n ; i++){
       if (i%2 == 0)
          break;
     printf(“%d “, i);
  }
}
```

A: 1 3 5 7 9

B: 0 1 2 3 4 5 6 7 8 9

C: 0 2 4 6 8

D: 1

iClicker Pop Quiz

What does this code do for n=10?

```
void foo (int n){
  int  i = 0;
   for(i = 1; i < n ; i++){
      if (i%2 == 0)
         continue;
    printf(“%d “, i);
}
```

A: 1 3 5 7 9

B: 0 1 2 3 4 5 6 7 8 9

C: 0 2 4 6 8

D: 1

iClicker Pop Quiz

Following is a correct function prototype for function factorial

A: void factorial (void);

B: void factorial (int a) { …}

C: int factorial (int a) { …}

D: int factorial (int);

iClicker Quiz

Which of the following is correct C statement to print an integer in variable a on screen?

A: scanf(%d, a);

B: printf(“%d”, a);

C: printf(“%d”, &a);

D: printf(%d, a);

iClicker Quiz

The format specifier for integer is

A: %f

B: %d

C: %c

D: %i

iClicker Pop Quiz

Following is the correct way to read an integer in C into a variable.

The variable declaration is: int temp;

A: scanf(%f,   temp);

B: scanf(“%f”, temp);

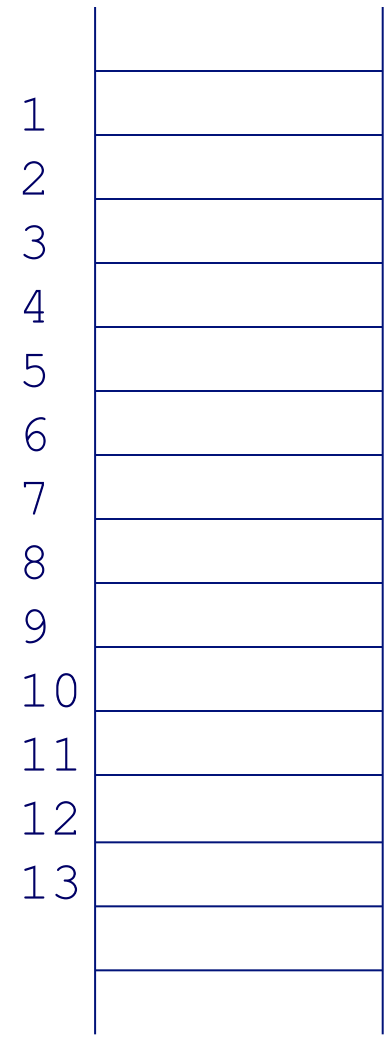
C: scanf(“%d”, temp);

D: scanf(“%d”, &temp);

Memory

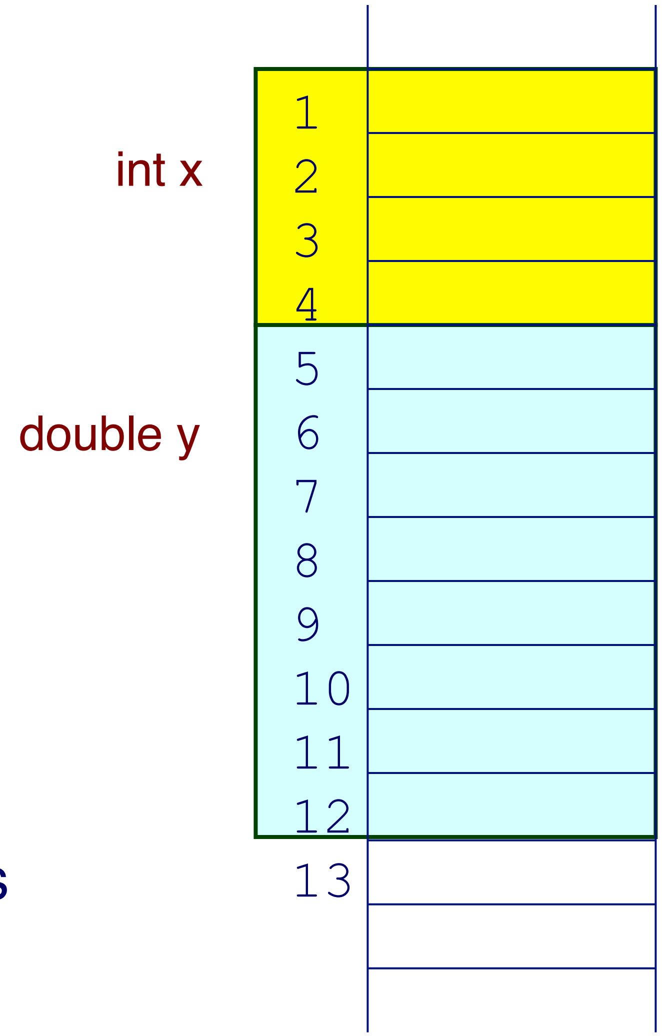
C’s memory model matches the underlying (virtual) memory system

- Array of addressable bytes



Memory

- C's memory model matches the underlying (virtual) memory system
  - Array of addressable bytes
- Variables are simply names for contiguous sequences of bytes
  - Number of bytes given by type of variable
- Compiler translates names to addresses
  - Typically maps to smallest address
  - Will discuss in more detail later



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 */

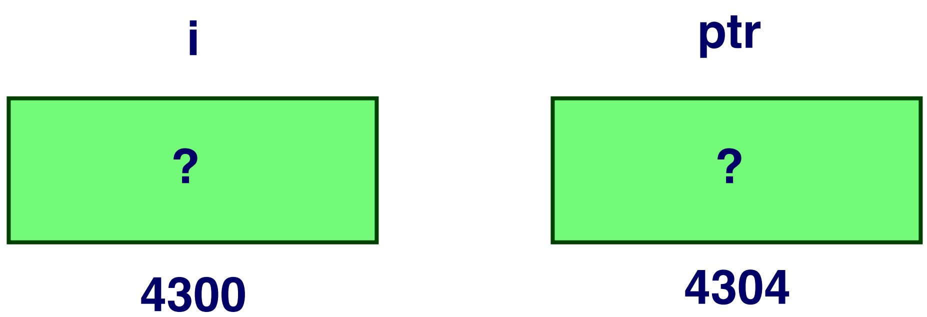
Pointer operators

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

Pointer Example

```c
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

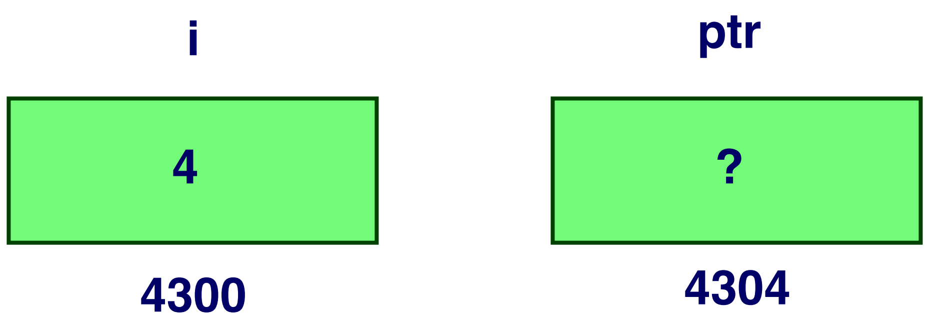


Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the value 4 into the memory location associated with i

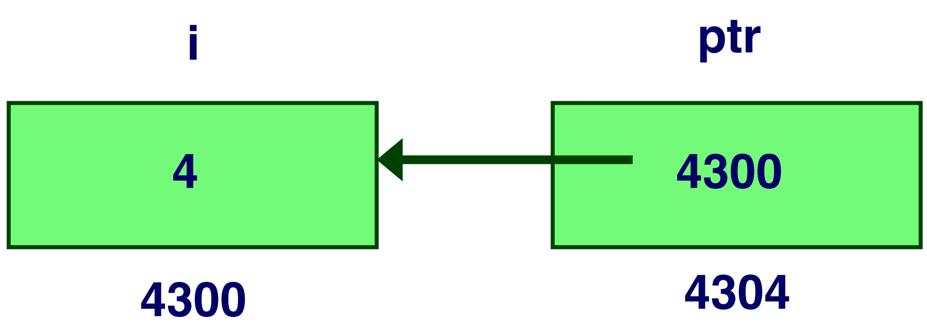


Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the address of i into the memory location associated with ptr



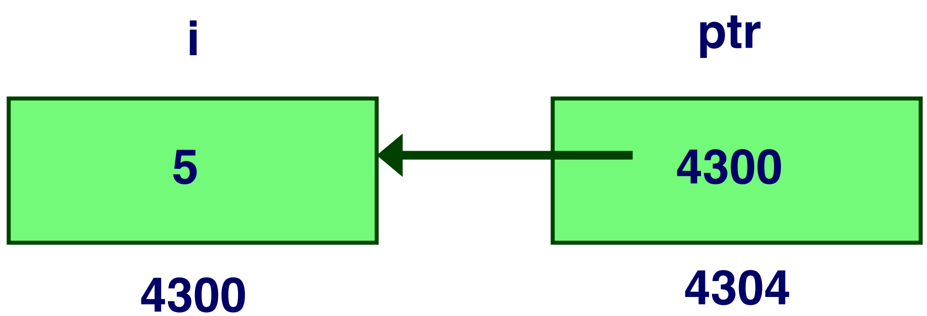
Pointer Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the result into memory at the address stored in ptr

read the contents of memory at the address stored in ptr



Example Use of Pointers

What does the following code produce? Why?

```
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Parameter Pass-by-Reference

Now what does the code produce? Why?

```
void Swap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Null Pointer

Sometimes we want a pointer that points to nothing

In other words, we declare a pointer, but we're not ready to actually point to something yet

```
int *p;
p = NULL;  /* p is a null pointer */
```

NULL is a predefined constant that contains a value that a non-null pointer should never hold

- Often, NULL = 0, because address 0 is not a legal address for most programs on most platforms

Type Casting

C is NOT strongly typed

Type casting allows programmers to dynamically change the type of a data item

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

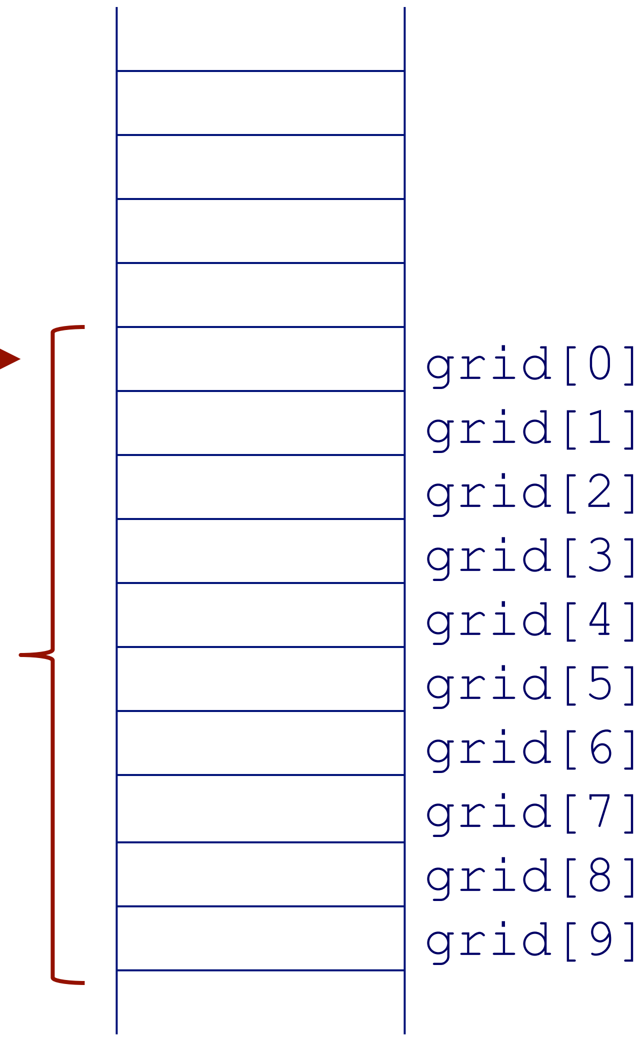
The C compiler and runtime system do not check array boundaries

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

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];
2. char *cptr;
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]

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]`

iClicker Pop Quiz

```
int main(int argc, char** argv){
    int a = 5, b = 6;
    int *p = &a;
    *p = 42;
     p = &b;
    *p = 71;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

A: a = 5, b = 6

B: a = 6, b = 5

C: a = 42, b = 71

D: a = 71, b = 42

iClicker Pop Quiz

```
int foo(int *p){
 for (int i = 0; i< 5; i++){
    printf("%d ", *p);
    p++;
 }
}
```

```
int main(int argc, char** argv)
    int arr[5] = {1,2,3,4,5};
    int b[3] = {211, 205, 206};
    foo(arr);
    return 0;
}
```

```
A: 1 2 3 4 5 211 205 206
B: 1 2 3 4 5
C: 211 205 206 1 2 3 4 5
D: 211 205 206
```

iClicker Pop Quiz

```
int foo(int *p){
 for (int i = 0; i< 3; i++){
    *p = 211;
    p++;
 }
}
```

```
int main(int argc, char** argv)
    int arr[5] = {1,2,3,4,5};
    int b[3] = {211, 205, 206};
    foo(arr);
    for (int i = 0; i < 5; i++){
      printf("%d ", arr[i]);
    }
    return 0;
}
```

A: 1 2 3 4 5 211 205 206

B: 1 2 3 4 5

C: 211 211 211 4 5

D: 211 205 206

iClicker Pop Quiz

```
int foo(int *p){
 for (int i = 0; i< 5; i++){
    printf("%d ", *p);
    p++;
 }
}
```

```
int main(int argc, char** argv)
    int arr[5] = {1,2,3,4,5};
    int b[3] = {211, 205, 206};
    foo(b);
    return 0;
}
```

A: 1 2 3 4 5 211 205 206

B: 1 2 3 4 5

C: 211 205 206

D: Wrong program

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 by reference (Makes sense because array name ~ pointer)

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);
}
```

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];
        …
}
```

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)
int strcmp(s1, s2)
size_t strlen(s)
```

`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
`\0`*nnn*	ASCII code *nnn* (in octal)
`\x`*nnn*	ASCII code *nnn* (in hex)

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

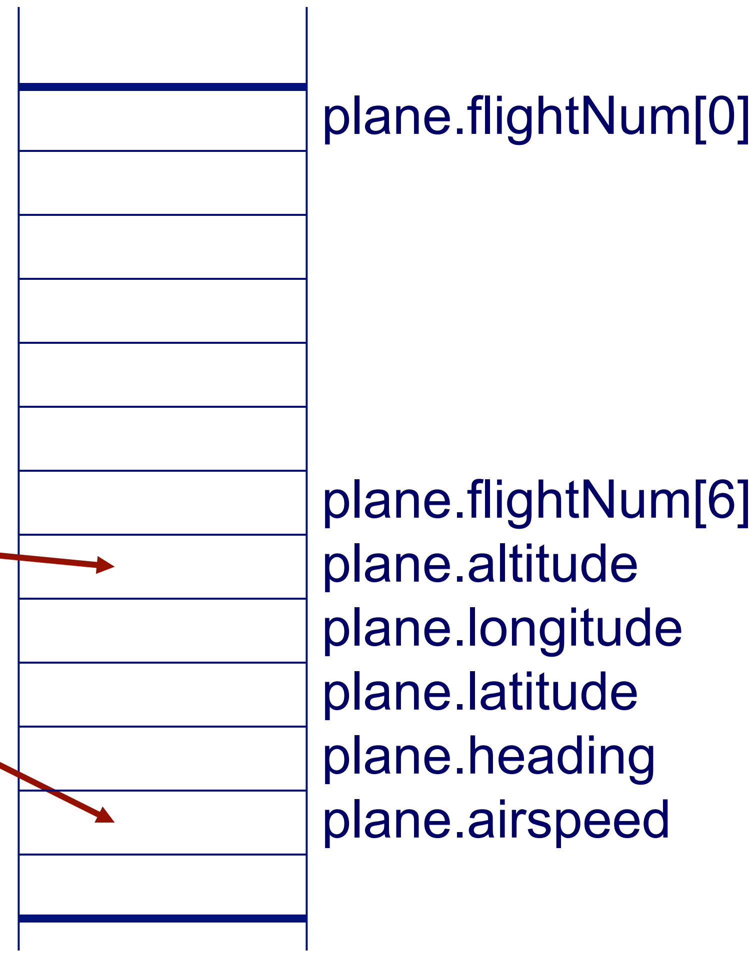
To allocate memory for a struct, we declare a variable using our new data type.

```
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 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;
```

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;
}
```

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

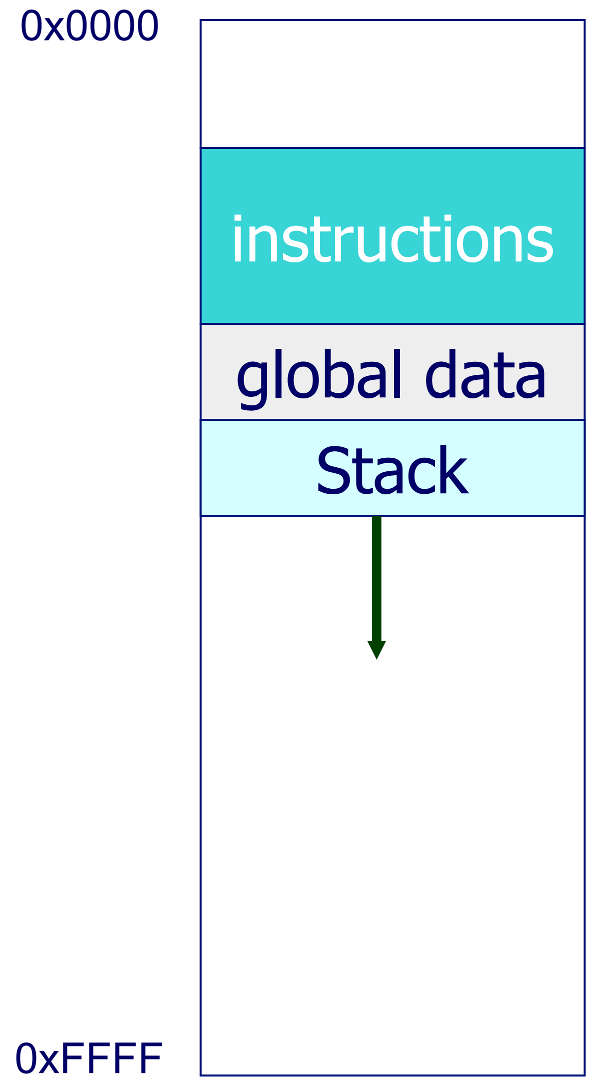
The state associated with a function is called an activation record

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



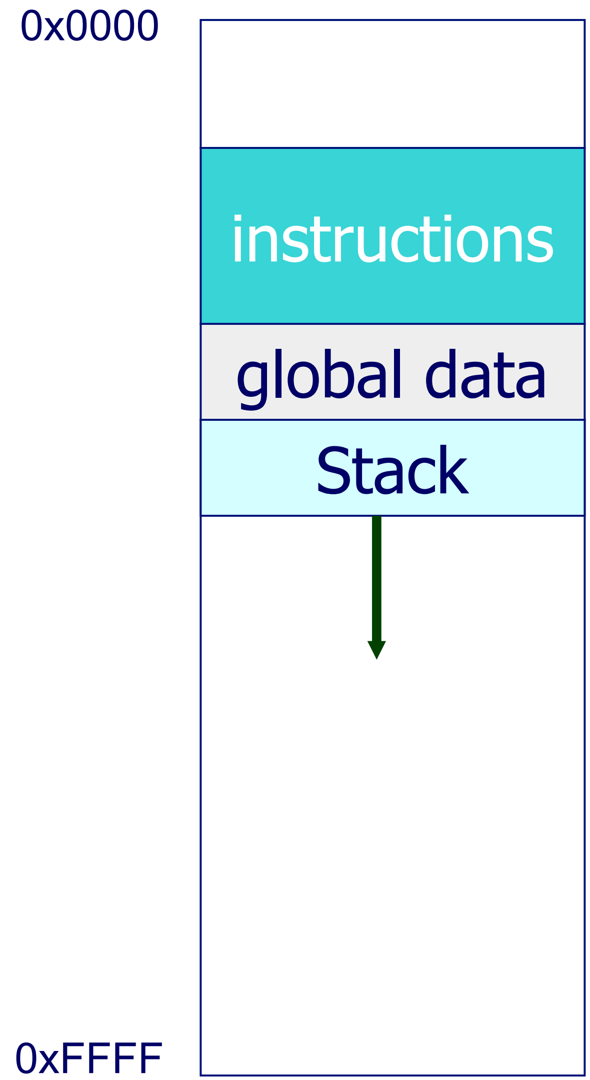
Allocating Space for Variables

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);
```

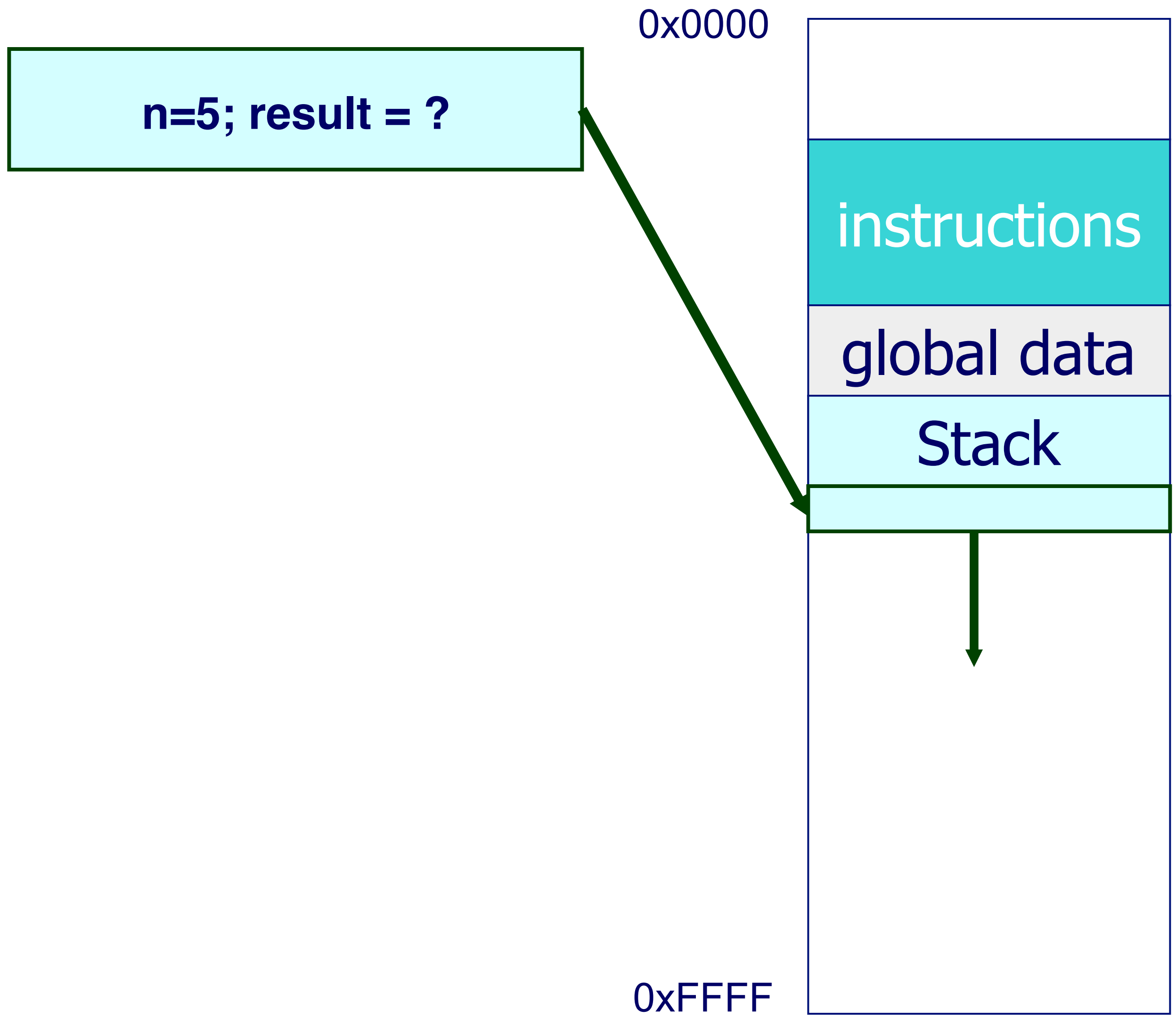
Recall that the activation record for a function contains state for all arguments, local variables, and result

int n; int result;



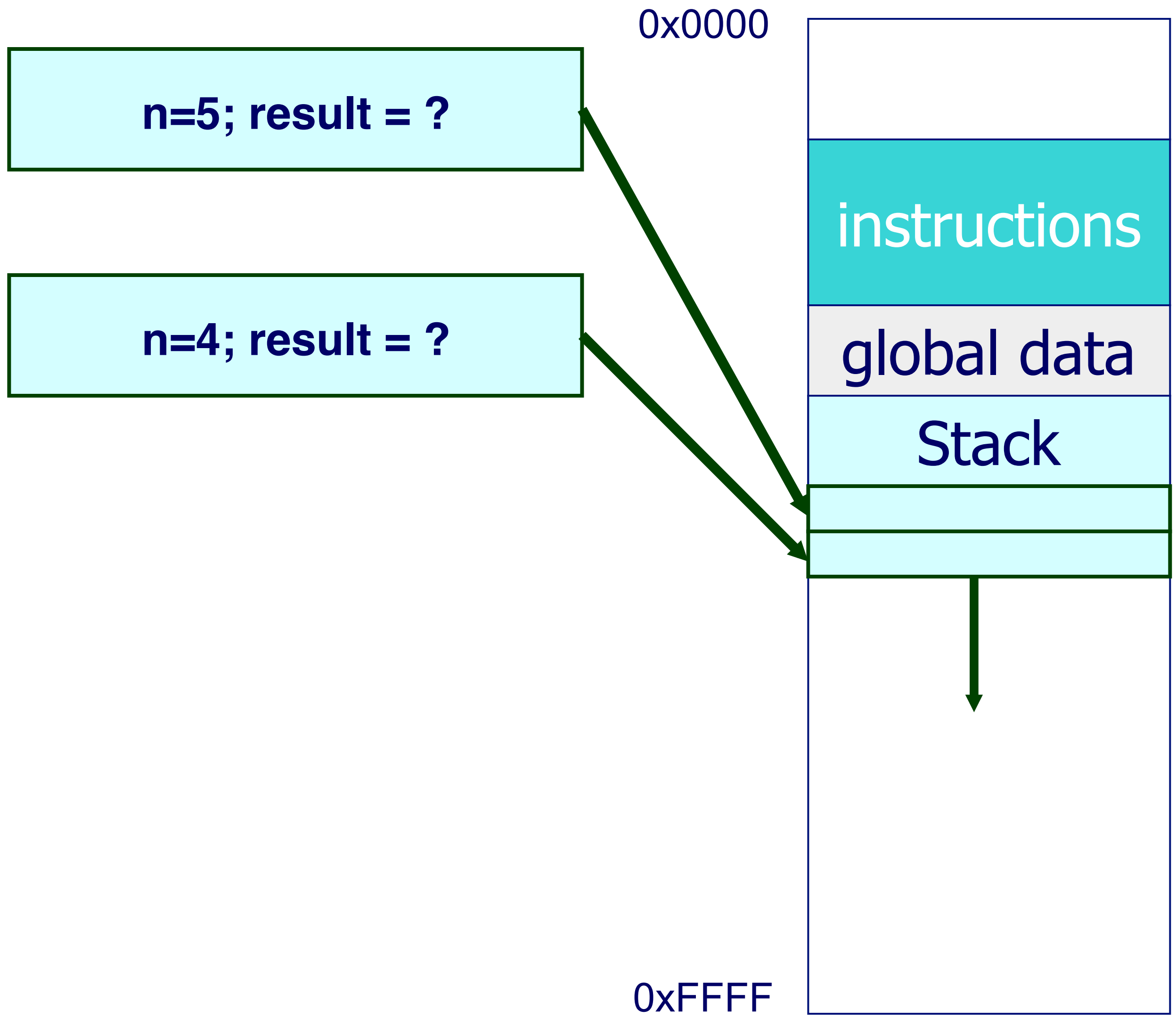
Allocating Space for Variables

```
summation(5);
```

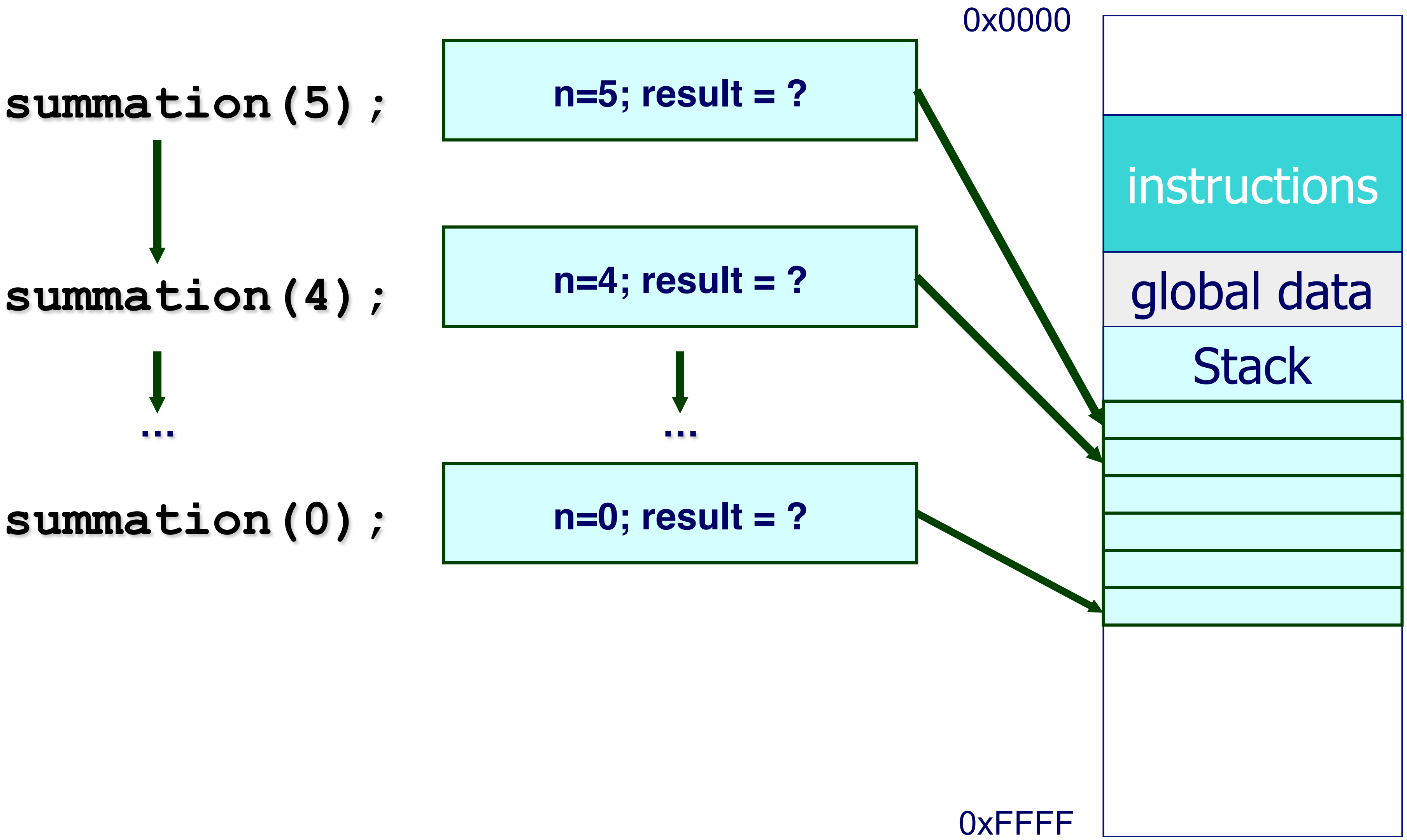


Allocating Space for Variables

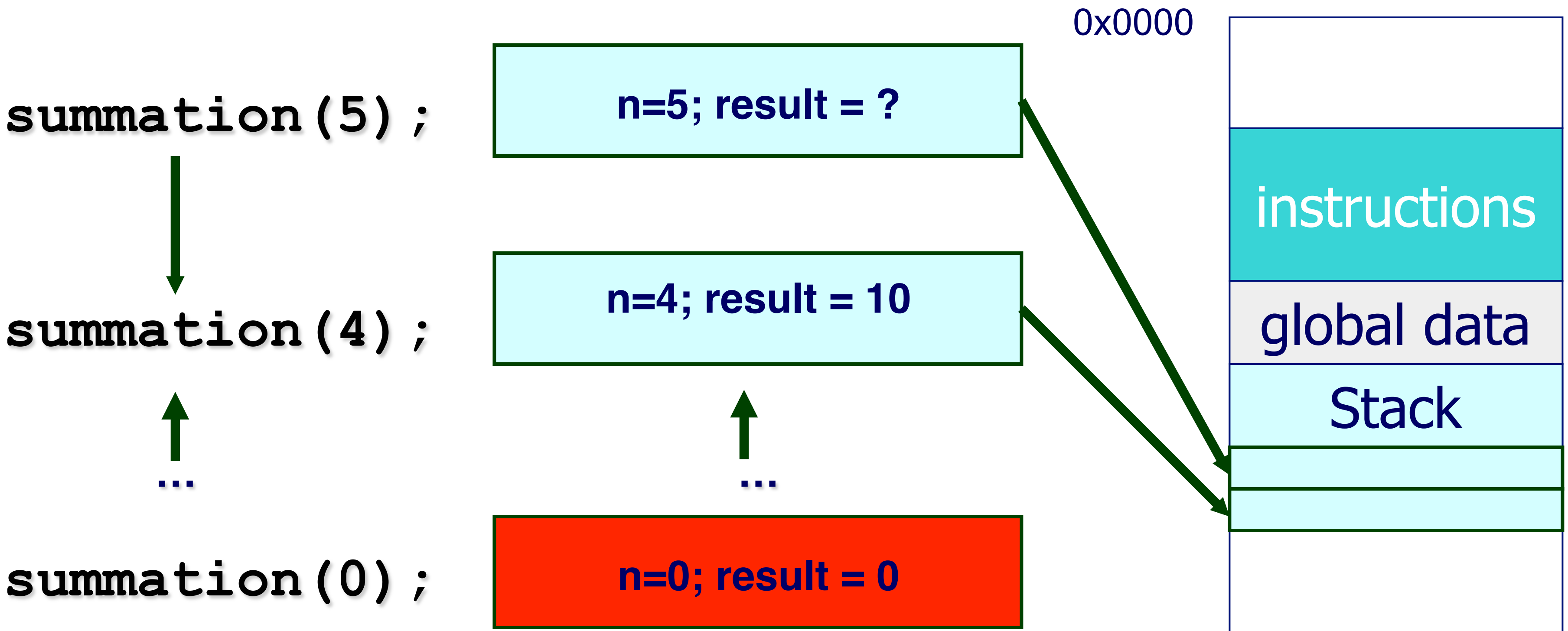
```
summation(5);
summation(4);
```



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

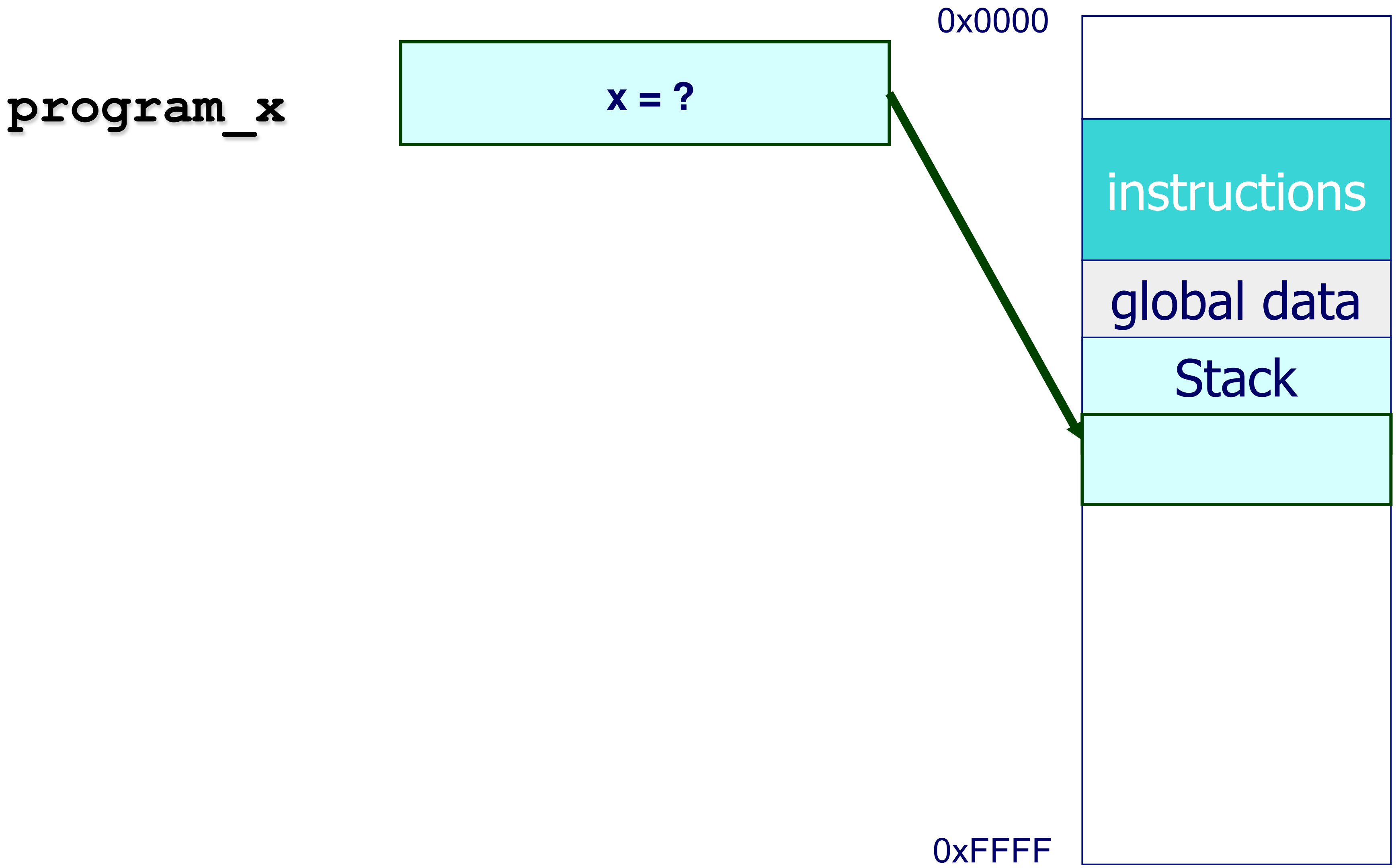
What if we want

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

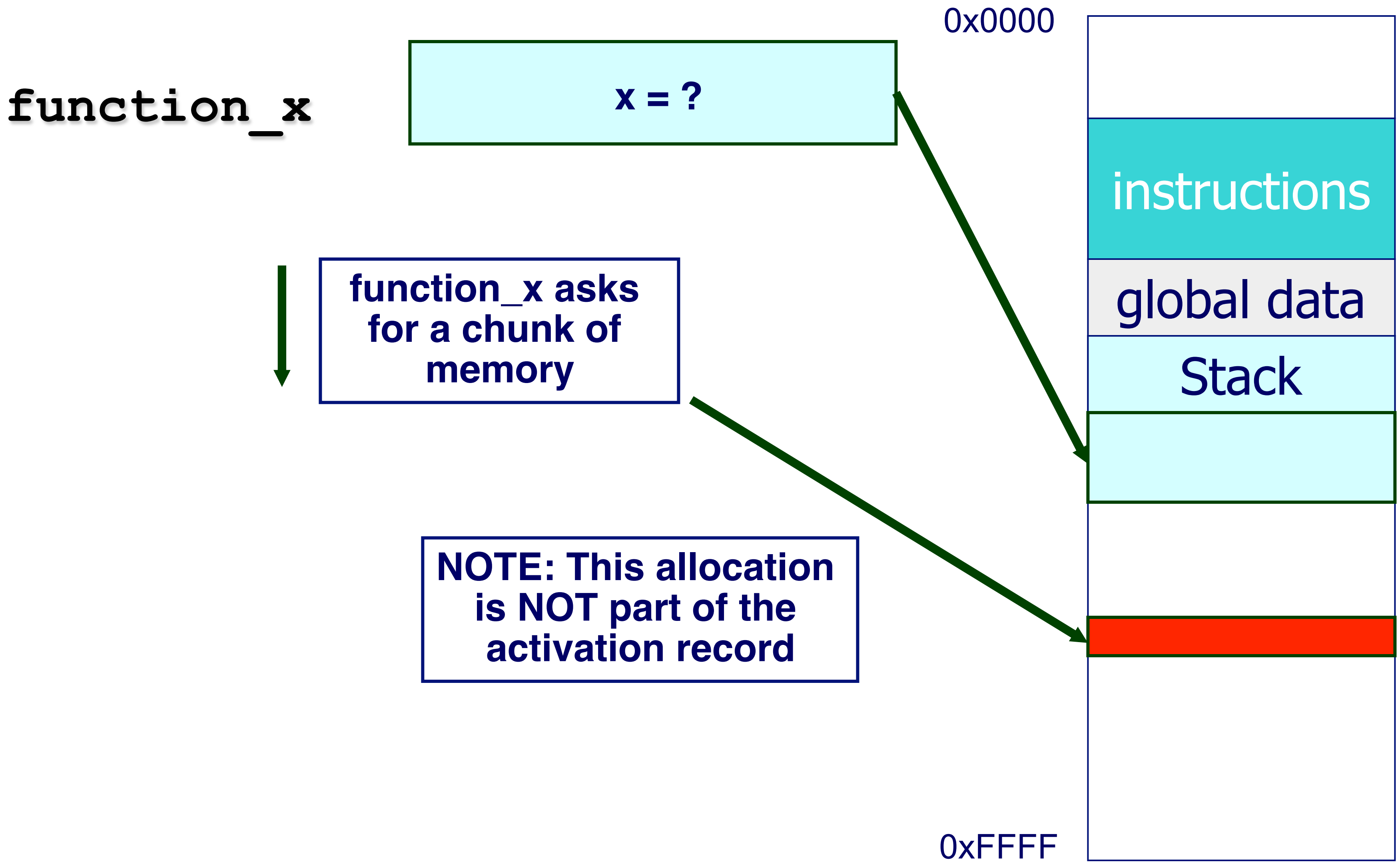
Two ways to “get memory”

- 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

Allocating Space for Variables



Allocating Space for Variables

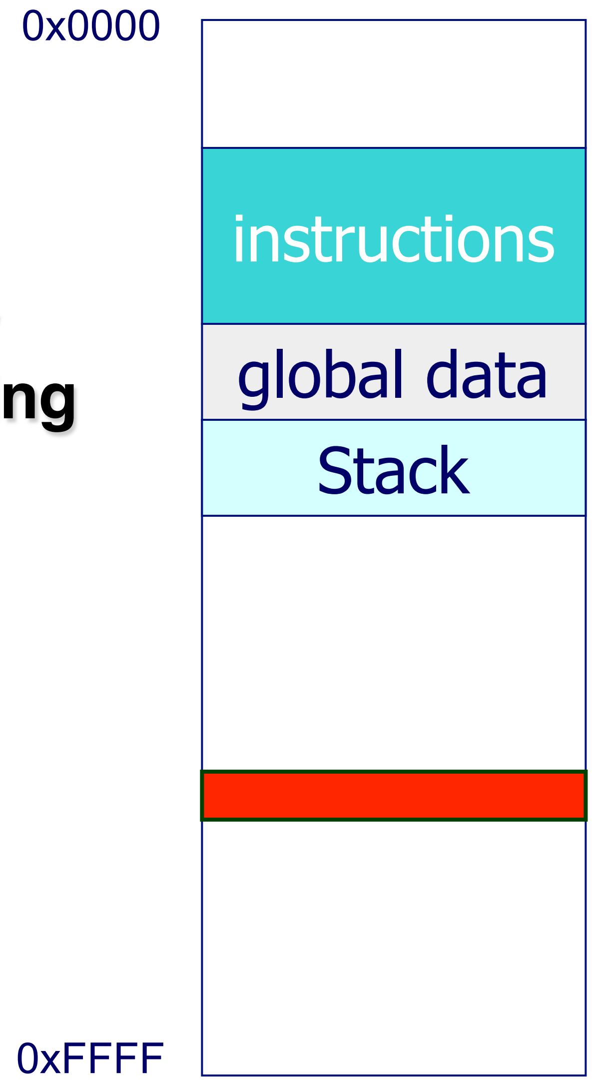


Allocating Space for Variables

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

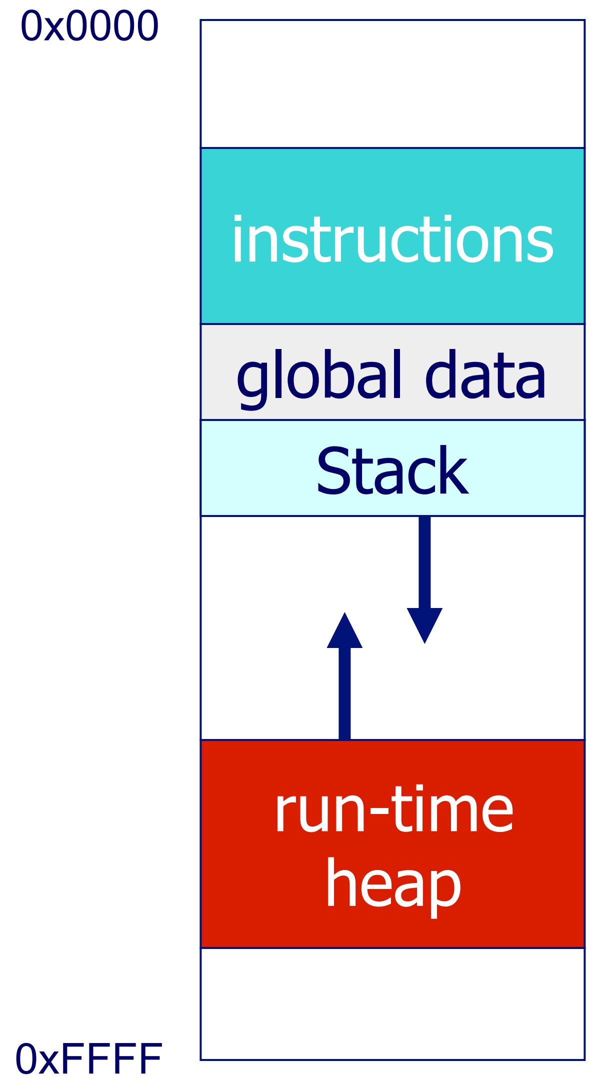


Dynamic Memory

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.

typedef

typedef is used to name types (for clarity and ease-of-use)

- typedef <type> <name>;

Examples:

- typedef int Color;
- typedef struct flightType WeatherData;
- typedef struct ab_type {
  int a;
  double b;
  } ABGroup;

Preprocessor

C compilation uses a preprocess called cpp

The preprocessor manipulates the source code in various ways before the code is passed through the compiler

- Preprocessor is controlled by directives
- cpp is pretty rich in functionality

Our use of the preprocessor will be pretty limited

- #include <stdio.h>
- #include "myHeader.h"
-
```
#ifndef _MY_HEADER_H
#define _MY_HEADER_H
            …
#endif /* _MY_HEADER_H */
```

Standard C Library

Many useful functionality provided by Standard C Library

- A collection of functions and macros that must be implemented by any ANSI standard implementation
  - E.g., I/O, string handling, etc.
- Automatically linked with every executable
- Implementation depends on processor, operating system, etc., but interface is standard

Since they are not part of the language, compiler must be told about function interfaces

Standard header files are provided, which contain declarations of functions, variables, etc.

- E.g., stdio.h
- Typically in /usr/include

Command Line Arguments

When using a shell

$ hello 5

Entire command line will be given to your program as a sequence of strings

- White spaces are typically the separator characters
  - Shell dependent
- int main(int argc, char * argv []) {

  …

  }
  - argc: number of strings in command line
    - In our example, argc = 2
  - argv: the strings themselves
    - In our example, argv[1] = “hello\0” and arg[1] = “5\0”

System Calls

The operating system extends the functionality of the underlying hardware

- OS functionalities exported as a set of system calls
- In C, system calls are “wrapped” by C functions
  - System calls look like C function calls
- System calls are described in section 2 of online manual
  - E.g., man 2 open

In some instances, the C standard library adds functionality on top of system calls

- File I/O

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 %lf",
  &name, &bMonth, &bDay, &bYear, &gpa);

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

- stdin, stdout, stderr

Summary

C is a language close to the hardware

- Ideal for building system software

- We looked at the basic aspects of C
  - Control flow – loops and break/continue statements
  - Pointers, structures, arrays
  - Memory management – malloc and free
  - File I/O operations