- Lecture Notes for Lecture 6 of CS 5600 (Computer Systems) for the Fall, 2019 session at the Northeastern University Silicon Valley Campus.
- *Managing Memory*
- Philip Gust,
  Clinical Instructor
  Khoury College of Computer and Information Science
- *Part 1 of a two-part lecture on managing memory*

Lecture 5 Review

- In this lecture, we learned that a library is a collection of related code and other resources that are packaged together for use by applications.
- We looked at examples of code in a library representing classes and functions that can be combined with the code of an application that calls or instantiates them.
- We learned the differences between shared and static libraries in C/C++ and how to create them and link them with applications.
- Finally, we looked at Java Archive (JAR) files as the analog of libraries in Java, and how to create regular JAR files as well as executable JAR files that can be run directly.

Managing Memory

Overview

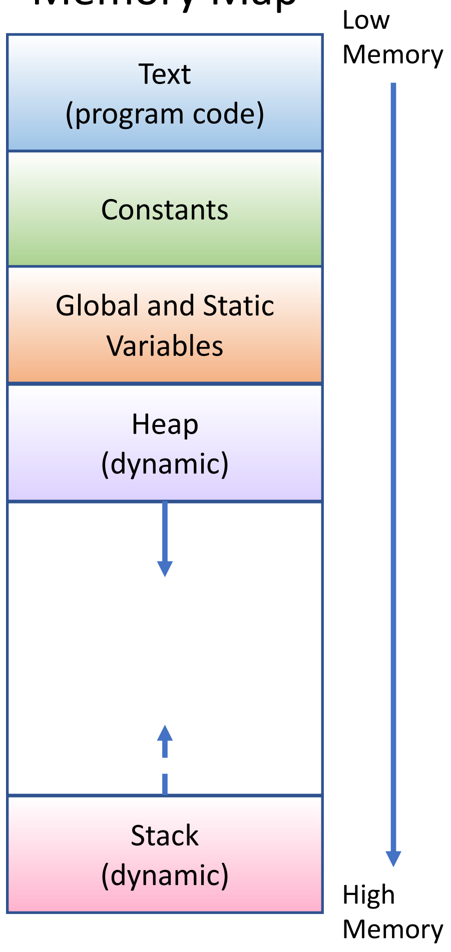
- In this lecture we will turn our attention to several topics related to how memory is managed. We will look at how memory is managed in a C program, including the use of the stack and heap segments.
- The stack segment is where *automatic memory* is managed. This is the storage used when a procedure is called. We will look at the call process, and see how stack storage is managed.
- The heap segment is where dynamic memory is allocated by C and C++ programs. We will begin looking at how this storage is organized and managed and the tradeoffs that must be made.

Managing Memory

Overview

- As we noted in the previous lecture, C or C++ program memory is divided into regions or *segments* that are used for specific kinds of storage
  - Text
    - Compiled code for the program
  - Constants
    - Literal strings and other fixed constants
  - Global and static variables
    - Variables declared globally or statically
  - Heap (dynamic)
    - A pool of memory programmers can allocate
  - Stack (dynamic)
    - Storage for local variables in functions

C Program Memory Map



Managing Memory

Overview

- The text, constants, and global storage areas are fixed in size based on the size of the code, constants, and global variables that are declared by a C or C++ program.
- The stack and heap segments occupy the remainder of memory, and provide storage used by calls to C or C++ functions, and for memory allocated by the program.
- The memory in the stack segment is called *automatic memory* because this memory is automatically allocated, managed, and deallocated by C or C++.
- Memory in the heap segment is called *dynamic* or *programmer-managed* memory because this memory is explicitly allocated and deallocated by a C or C++ program.

Managing Memory

Managing the stack segment

- We will first look at how automatic memory is managed by C or C++ when a function is called, including storage declared within a function, storage for parameters and a return value, and storage overhead for the function call mechanism.
- We will also consider how calls to instance and class functions are handled in C++, including how the hidden *this* parameter is made available to an instance function.

Managing Memory

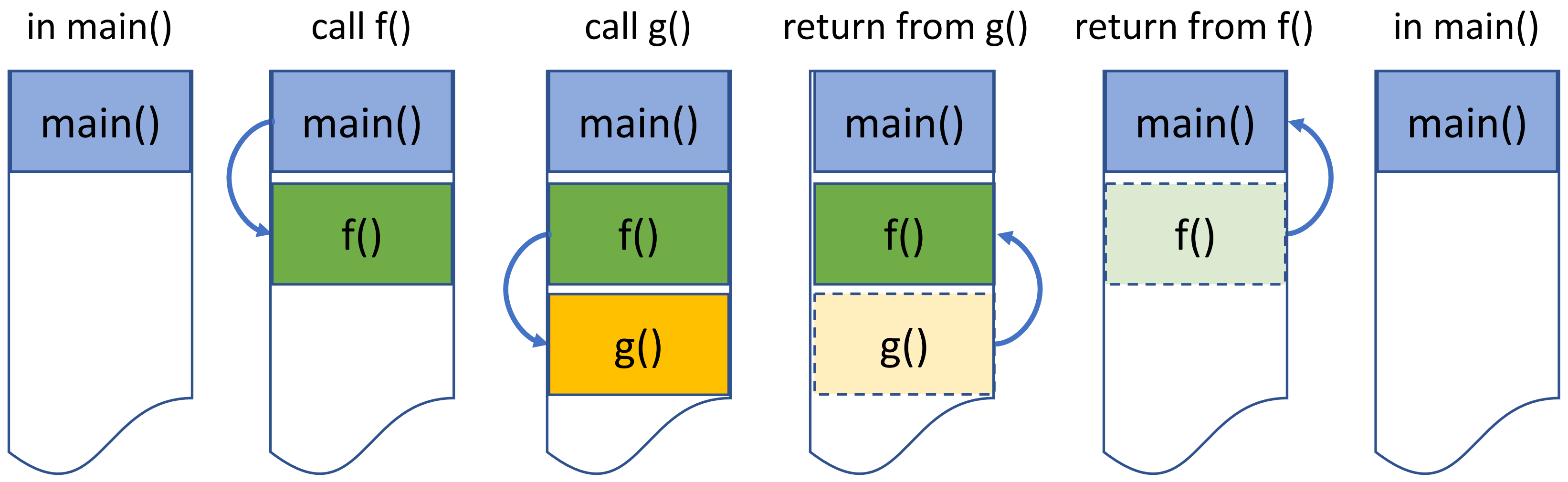
Managing the stack segment

- When a function is called, several things must take place to ensure that
  - parameters are passed
  - the return value is made available
  - state information is saved to enable the calling function to resume operations on return from the call.
- The storage for all of these steps is allocated on the stack as the function is called, runs and returns its result to the caller.

Managing Memory

Managing the stack segment

- The memory used by a call to a function and the storage used by the function is often referred to as the *activation record* because it holds the storage for an *activation* of the function.
- The a new activation record is created on the stack when the function is called, and removed when the function returns.



Managing Memory

Managing the stack segment

- An initial frame is created by the C or C++ runtime when the program is loaded and control transfers to a startup function.
- This function performs initial setup, opens default file descriptors in global storage, and calls the main() function, passing in the program arguments.

Managing Memory

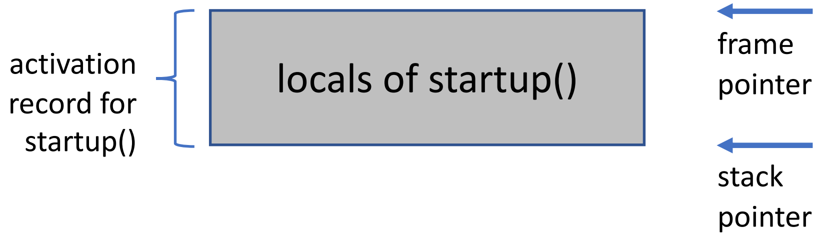
Managing the stack segment

- When the startup function calls main() the call *site processing* creates a partial activation record on the stack, consisting of:
  - **processor state**: This includes caller-saved registers. The state will be restored by the caller when the function returns.
  - **space for return value**: This is written to by the called function and retrieved by the calling function. Could instead be returned in a register.
  - **parameter values**: These values are accessed as local variables by the called function.

Managing Memory

Managing the stack segment

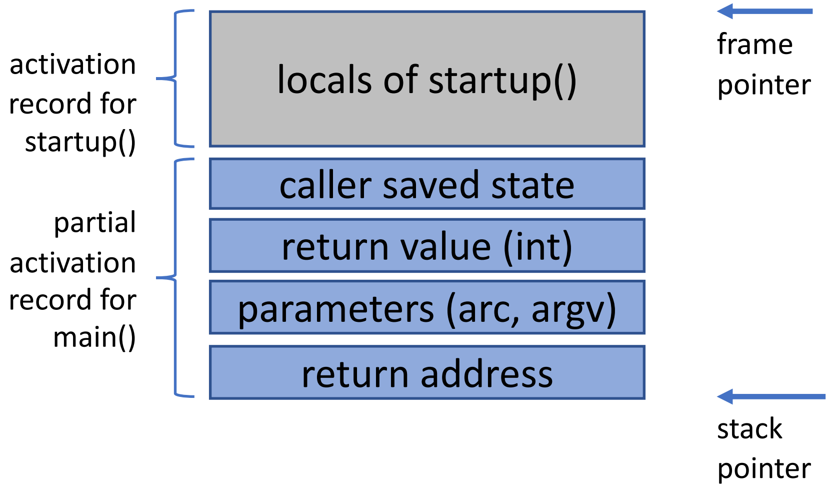
- Here is the stack before the startup function has called main().
- The initial frame has no caller so no preamble information is on the stack, and the frame pointer points to the start of the stack



Managing Memory

Managing the stack segment

- Here is the stack after just after the startup function has transferred control to main().
- The instruction that transfers control to the function pushes the return address onto the stack.

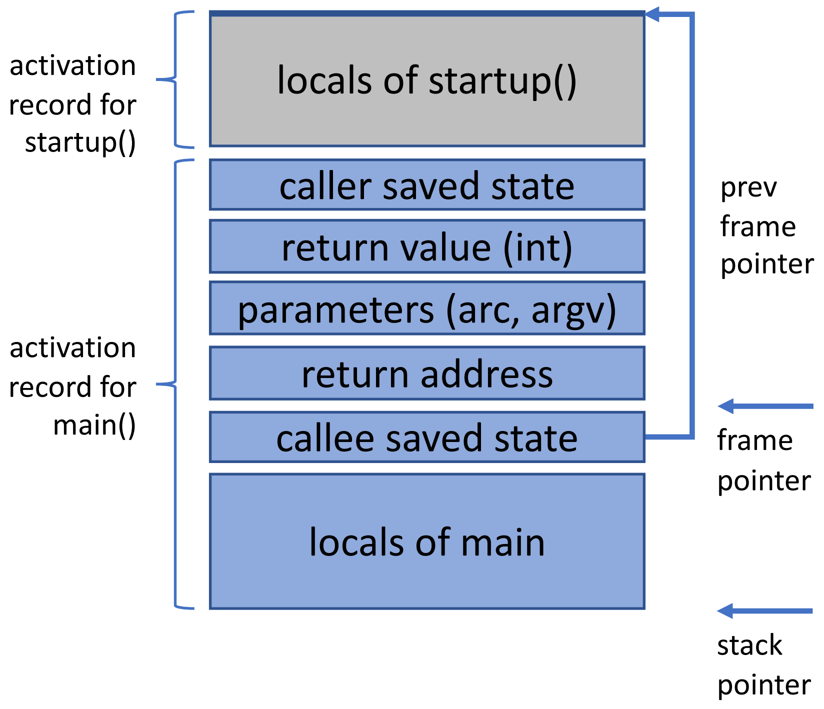


- At this point, the frame pointer still points to special frame for the startup function.
- Note that parameters are generally pushed in reverse order if the C stack is descending (usually so).

Managing Memory

Managing the stack segment

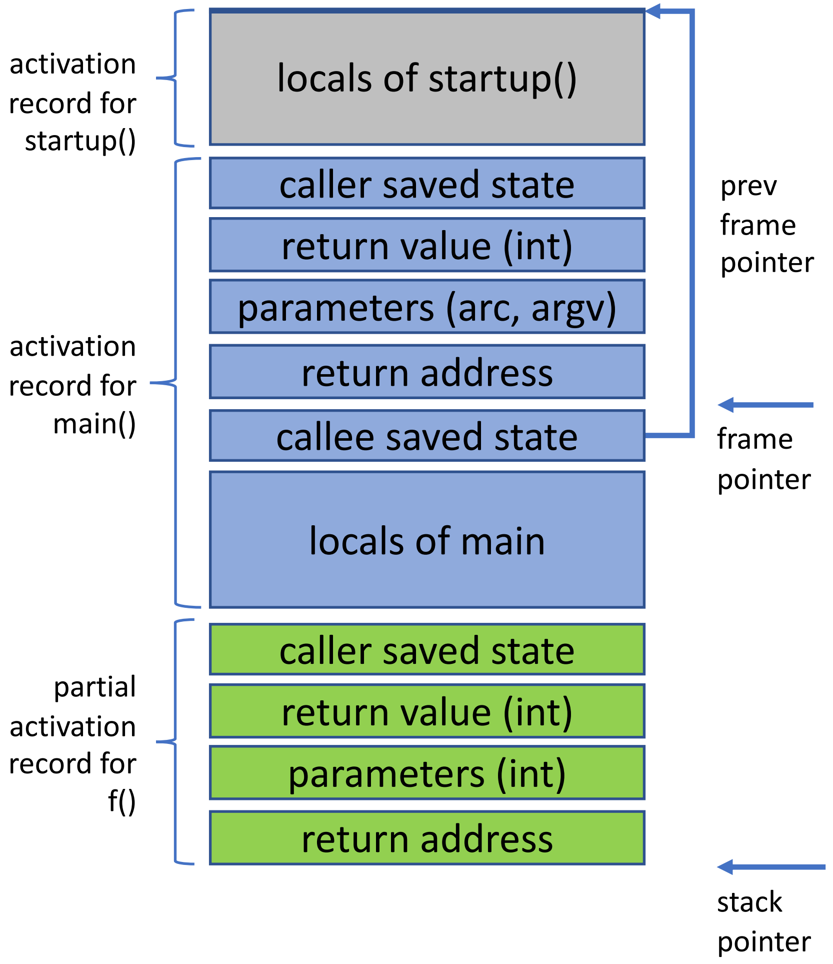
- During *function entry processing*, save caller's frame pointer and adjust to the current stack pointer.
- Next, space for storage that is declared in the main block of main() is allocated on the stack. Storage for variables declared in nested blocks is pushed upon block entry and popped upon block exit.



Managing Memory

Managing the stack segment

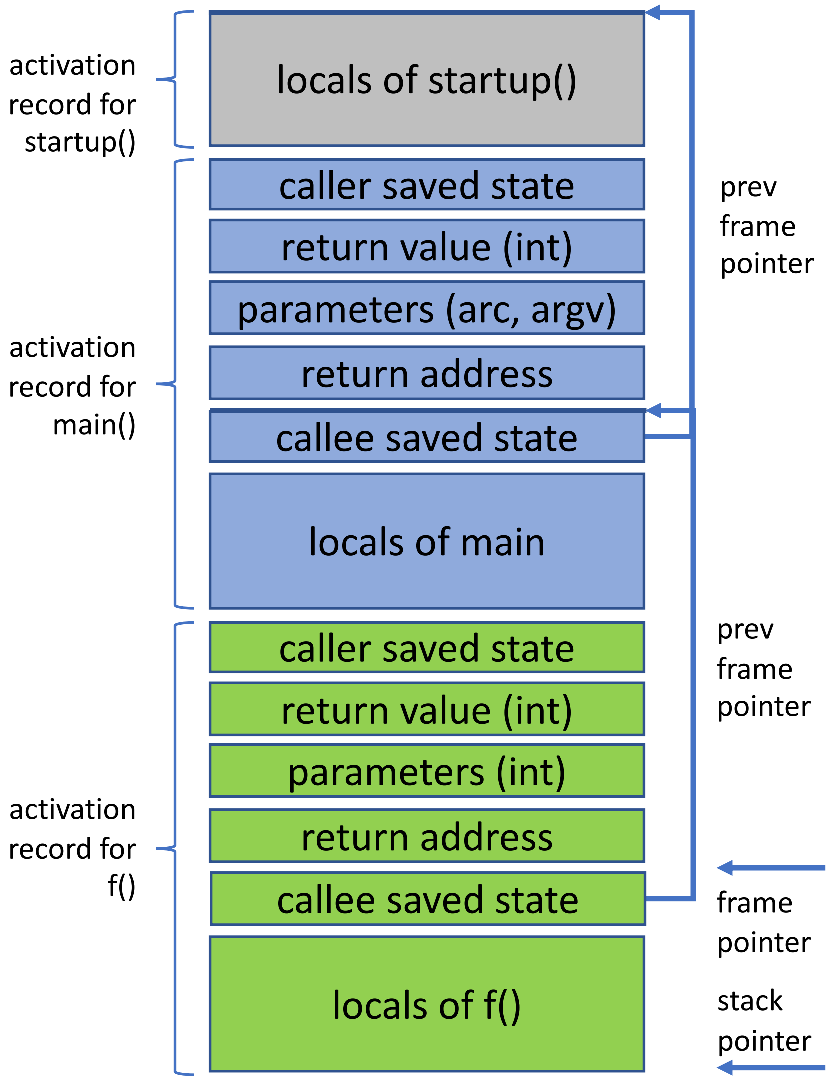
- The process is repeated if main calls function f().
- During site processing, caller saved state, storage for a return value, and parameters are pushed onto the stack, then control transfers to function f(), placing the return address onto the stack.



Managing Memory

Managing the stack segment

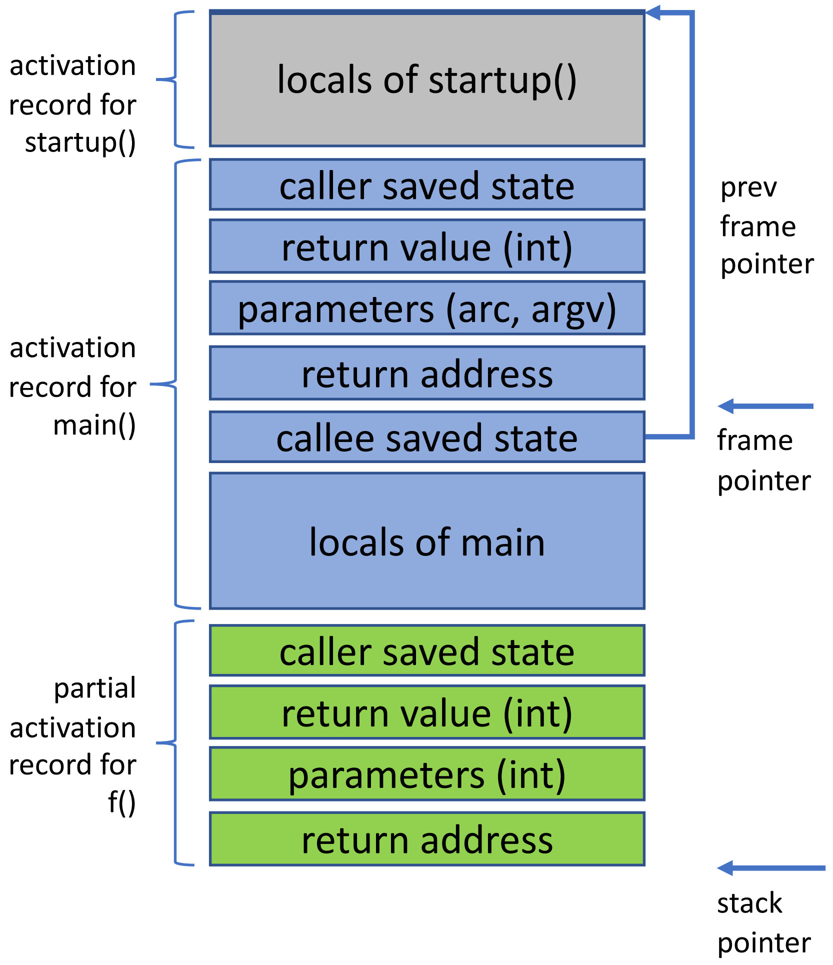
- Again, the frame pointer is saved and adjusted to the stack pointer on entry and space for storage declared in f() is allocated on the stack.
- Before returning, f() writes its return value into the return value location in the activation record for the caller to read.



Managing Memory

Managing the stack segment

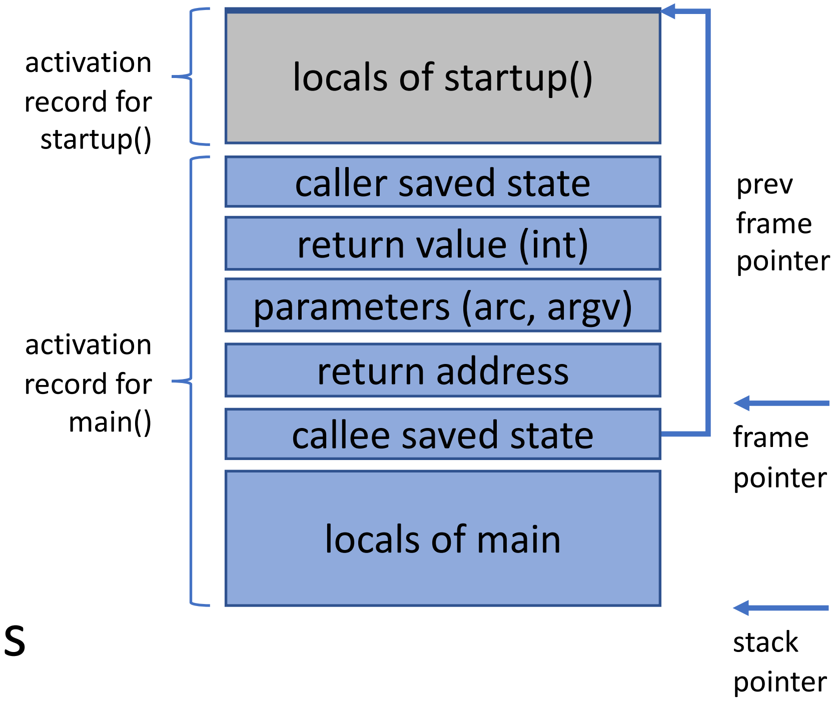
- When returning from a function, the stack must be unwound by reversing the steps we just followed.
- In the callee, we must
  - Reset the stack pointer to the current frame pointer to clear local storage
  - Restore the previous frame pointer from the saved callee state.
  - Return from the function, popping the return address



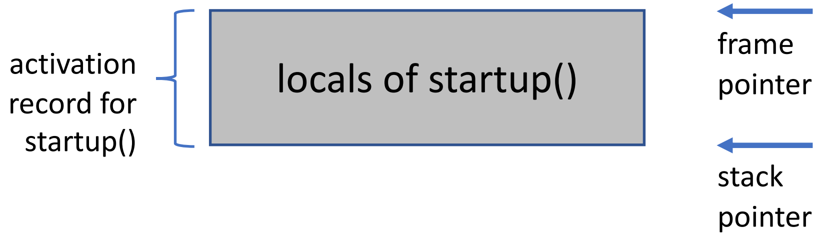
Managing Memory

Managing the stack segment

- In the caller, we must also:
  - Discard the parameters
  - Pop and assign the the return value
  - Pop and restore any caller-saved state
- At this point, the code resumes after the point where the call was made.
- In a C++ program, the callee function must also run destructors for local objects in the activation at the start of of the unwinding process.



Managing Memory



Managing the stack segment

- When main() returns, the startup function shuts down open file descriptors, and calls the system *exit()* function with program status returned from *main()* (default is 0).
- The startup function has a special activation record with only local variables but no return address, so there is nowhere for it to return to.
- In C++ programs, the startup function also catches and reports any exceptions that were not caught by main().

Managing Memory

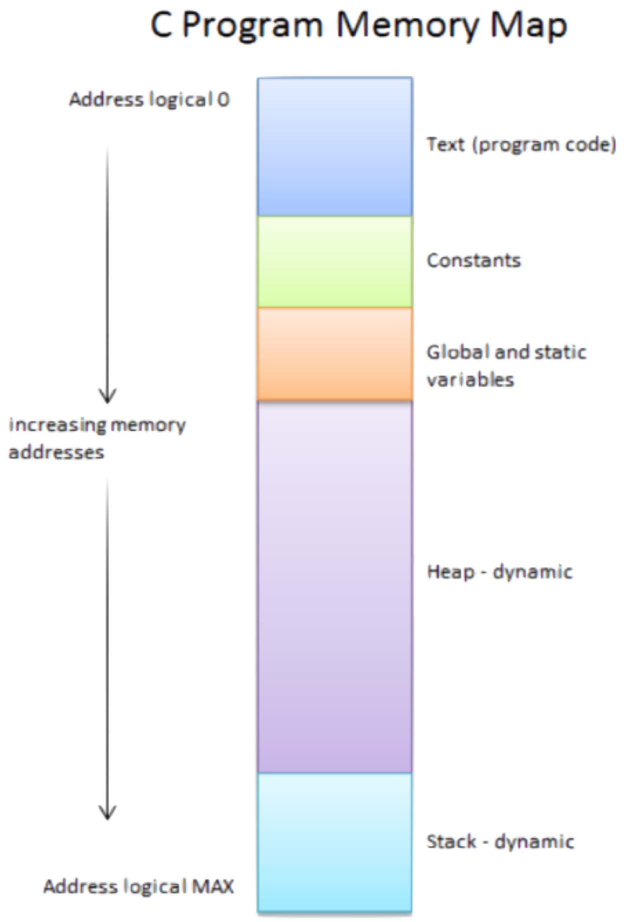
Managing the stack segment

- To summarize, the stack segment is used for local variables in functions. This storage is called *automatic memory* because it is automatically managed by the C or C++ runtime.
- When a function is called, an activation record is created on the stack for the local variables declared within the function.
- Additional overhead is required to save the caller state, pass parameters, provide space for a return value and save the return address and other context required by the callee.
- Both the caller and the callee share responsibility for the processing required to implement a function call.

Managing Memory

Managing the heap segment

- A dynamic memory allocator maintains an area of memory known as the heap. Details vary from system to system, but we will assume that begins immediately after the uninitialized bss area and grows toward higher addresses).
- A variable brk (pronounced "break") that points to the top of the heap.



Managing Memory

Managing the heap segment

- An allocator maintains the *heap* as a collection of various-sized blocks. Each block is a contiguous chunk of virtual memory that is either allocated or free.
- An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application.
- An allocated block remains allocated until it is freed, either explicitly by the application, or implicitly by the memory allocator itself.

Managing Memory

Managing the heap segment

- Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks.
- Explicit allocators require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the malloc package.
- C programs allocate a block by calling the malloc function, and free a block by calling the free function. The new and delete calls in C++ are comparable.

Managing Memory

Managing the heap segment

- Implicit allocators, on the other hand, require the allocator to detect when an allocated block is no longer being used by the program and then free the block.
- Implicit allocators are also known as garbage collectors, and the process of automatically freeing unused allocated blocks is known as garbage collection.
- For example, higher-level languages such as Lisp, ML, and Java rely on garbage collection to free allocated blocks.

Managing Memory

Managing the heap segment

- The original C memory allocation code, implemented by Brian Kernighan and Dennis Richie and described in the book on C (Section 8.7), is an explicit allocator.
- The allocator maintains a free list of blocks available for allocation.
  - –Free block with header (pointer and size) and user data
  - –Aligns the header with the largest data type
  - –Circular linked list of free blocks

Managing Memory

Managing the heap segment

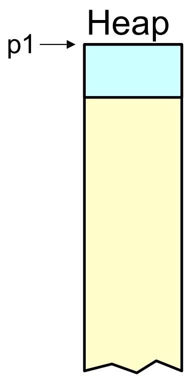
- The allocator provides the following function for interacting with the allocator.
  - void *malloc(size_t size)
    - Allocates memory in multiples of header size
    - Finding the first element in the free list that is large enough
    - Allocating more memory from the OS, if needed
  - realloc(void *ptr, size_t size)
    - Returns an allocated memory with the requested size
    - If the already allocated block is large enough, returns that block
    - If not large enough, allocates new block with copy of content and frees old block
  - free(void *ptr)
    - Puts an allocated block back in the free list
    - Coalesces the block with adjacent blocks if any

Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
➡ char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

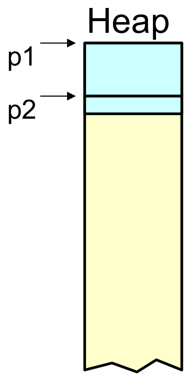


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
➡ char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

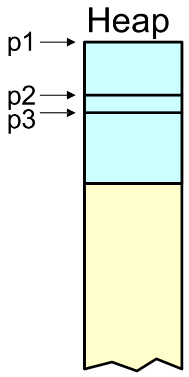


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
➡ char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

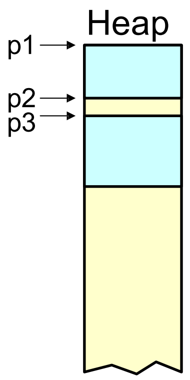


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
   char *p3 = malloc(4);
➡ free(p2);
   char *p4 = malloc(6);
   free(p3);
   char *p5 = malloc(2);
   free(p1);
   free(p4);
   free(p5);
```

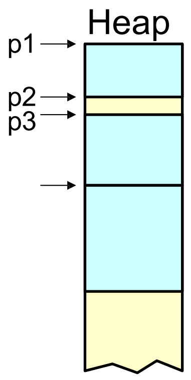


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
➡ char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

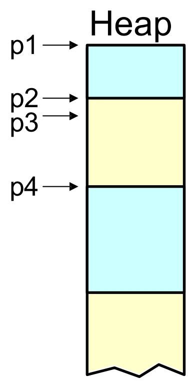


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
➡ free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

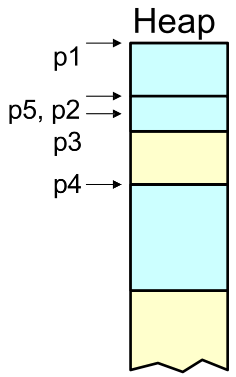


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
➡ char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

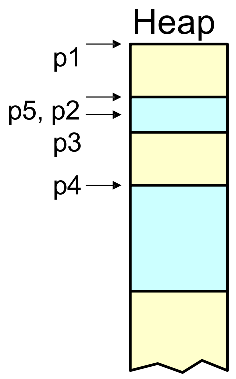


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
➡ free(p1);
  free(p4);
  free(p5);
```

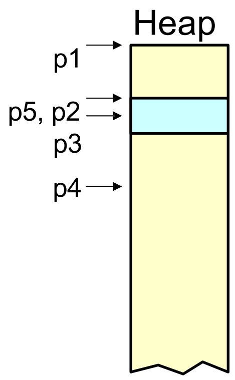


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
➡ free(p4);
  free(p5);
```

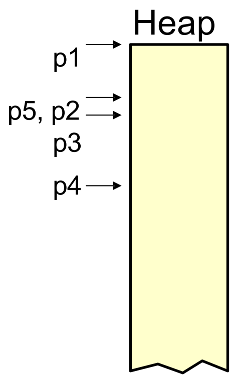


Managing Memory

Managing the heap segment

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
➡ free(p5);
```


Heap
p1
p5, p2
p3
p4