Programming Languages
Lecture07 – Names, Scopes, Binding

남 범 석

bnam@skku.edu

Semantics

- Syntax (grammars, r.e.'s, DFAs) are well understood, and can be treated formally

- For assigning meaning to programs
  - as opposed to syntax, which is about the structure of programs
    - Eg) CFG doesn't know what + token is for.

- Semantics is less well formalized
  - but there are important abstractions that can help

Semantics (cont.)

- Entities discussed in programming languages describe chunks of memory in a computer
  - so they assign semantics to bits
- Examples: data structures, variables, functions, pointers
- When transforming a program by a compiler, semantics are assigned in many levels.

Name, Scope, and Binding

- A Name is exactly what you think it is.
  - Most names are identifiers
  - Symbols (like '+') can also be names
- A Binding is an association between two things, such as a name and the entity it names.
- The Scope of a binding is the part of the program in which the binding is active.
  - How long is the memory for a subprogram (e.g., function, subroutine, procedure) retained?

Questions:

```
int x = 1, y=1;
int *p = &x;
int &q = y;
q++;
(*p)++;
q = x;
p=&y;
```

Binding

- Binding Time is the point at which a binding is created
  - Ex. (cont.): look at local variables
    - 1. Fortran - all data memory for all subprograms is allocated once, by the compiler, and retained throughout the program lifetime
    - 2. C/C++/Java - data memory for local variables is allocated at runtime, each time a subprogram is invoked, and is deleted when the subprogram exits/returns
- These bindings imply that C/C++/Java can have recursive procedures, but Fortran can't
  - Recursion forces delay in binding time of local variables to locations

Binding (cont.)

- Binding time - when does the binding of an attribute to a program entity occur?
  - 1. Static - at translation (compile/link) time
  - 2. Dynamic - at runtime, during program execution

- In general,
  - Early binding times have greater efficiency
    - Compiled languages tend to have early binding times
  - Later binding times have greater flexibility
    - Interpreted languages tend to have late binding times

Static vs. Dynamic Binding

- Static binding
  - Can be more secure
    - e.g., type checking at compile time
  - more efficient
    - done once for multiple executions on multiple data sets at compile/link time

Static vs. Dynamic (cont.)

- Dynamic binding
  - Can be more flexible, based on actual types/values at run time for program entities
    - e.g., C++ virtual functions, Java dynamic dispatch
  - Can add new code/data at run time
    - as in Ruby, Smalltalk, Lisp, ML

Scope

- Scope is one of the binding issues
- Scope - range of program statements in which a variable is visible
  - 1. Static - based on lexical structure (syntax) of program, so can be determined by compiler
    - Ruby, C, C++, Java, Fortran, ML, Pascal
  - 2. Dynamic - based on execution path of program, so only determined at run time
    - original versions of Lisp, but newer versions, including CommonLisp, are statically bound

Static Scoping

- Find the declaration of the variable, from the inside out (for nested scopes)
  - Scope is a block in C/C++/Java/Ruby, a procedure in Pascal
- Next slide shows an example, with pseudo-code in Pascal-like syntax

Static Scoping - Example

```
procedure p
  var x: integer;

  procedure p1;
  begin
      write(x)
  end;

  procedure p2;
    var x: integer;
  begin
      x := 2
      p1;
  end;

begin
  x := 0
  p2;
  p1;
end;
```

0 and 0 will be printed out

Example (cont.)

- Use of x in p1 refers to x declared in procedure p, when p1 is called from either p2 or p
- Can tell, just from looking at program text (syntax), which declaration is used for variable reference

Dynamic Scoping

- Based on calling sequence, at runtime, of the program
- For same example - must look at call chain to determine binding of variable to the correct declaration
  - so procedure p, which declares x, calls p2, which also declares x, and p2 calls p1
    - then the x used in p1 is bound to the x declared in p2
  - On the other hand, for the direct call to p1 from p, the use of x in p1 is bound to the declaration of x in p

Dynamic Scoping - Example

```
procedure p
  var x: integer;

  procedure p1;
  begin
      write(x)
  end;

  procedure p2;
    var x: integer;
  begin
      x := 2
      p1;
  end;

begin
  x := 0
  p2;
  p1;
end;
```

p → p2 → p1

p → p1

2 and 0 will be printed out

Dynamic Scoping (cont.)

- Dynamic scoping is confusing, since can't tell from program text which variable declaration is being referred to
  - and it can be different for each invocation of a subprogram (based on the current call chain, or environment)

Extent

- Refers to *storage binding of variables*
- Allocation - when is memory for a bound variable taken from the available pool?
- Deallocation - putting the memory back when the variable is unbound

Static Extent

- Memory allocated before program begins execution (e.g., by compiler) and deallocated when program terminates
  - Ex.: global variables in C/C++, all variables in traditional Fortran
- Advantages
  - direct addressing, since memory locations known at compile time
  - no runtime overhead for allocation/deallocation

Dynamic Extent

- Memory allocated/deallocated at runtime
  - either on procedure/block entry/exit (where a variable is declared)
  - or by explicit runtime user call (new/delete in C++, new in Java)
- Advantages
  - allows recursive procedures and explicit user-managed allocation
  - allows dynamic data structures (e.g., linked lists, trees), but requires pointers/references

Dynamic Extent (cont.)

- Disadvantages
  - cost of managing recursive procedure calls (runtime stack)
  - cost of allocating/deallocating memory at runtime
  - allows memory leakage - losing the last pointer to dynamically allocated memory, without a call for explicit deallocation

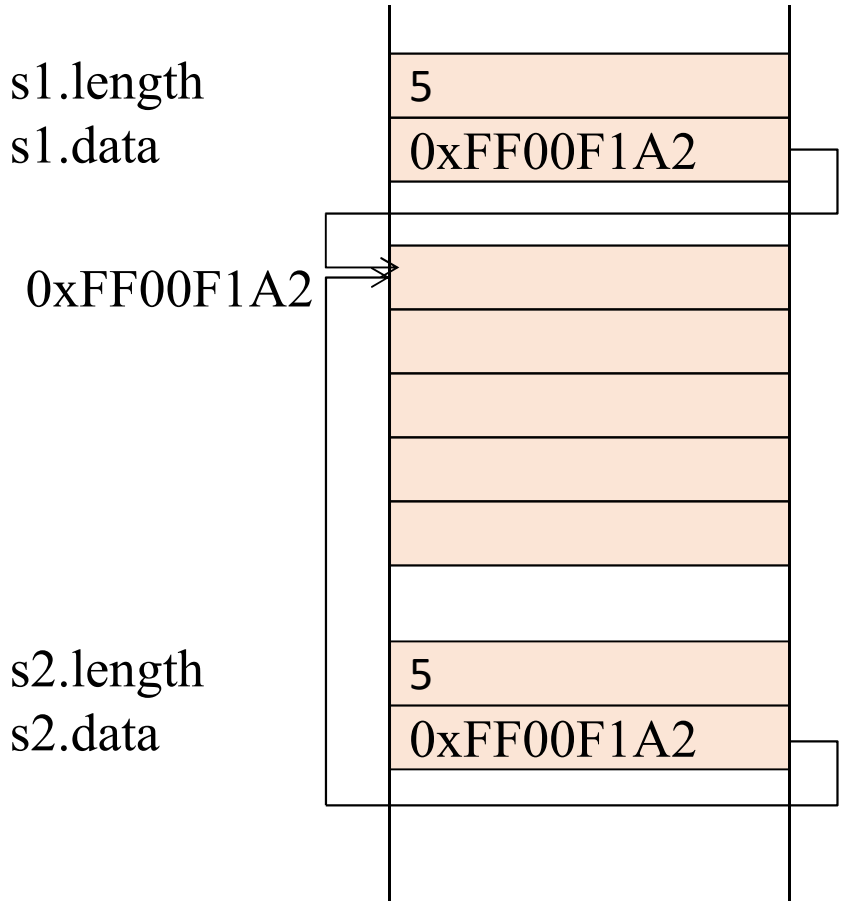
Question:

- Any problem with this code?

```
class SimpleArray {
   private:
      int length;
      int *data;
   public:
      SimpleArray(int s){
        data = new int[s];
        length = s;
      }
      ~SimpleArray(){
        delete[] data;
      }
};
```

Default Copy Constructor Causes Memory Sharing

```
SimpleArray s1(5);
if (true) {
  SimpleArray s2 = s1;
}
// s1 is corrupted
// when s2 goes
// out of scope and
// its destructor executes
```




```
//copy constructor will be called when you pass/return
// a class object to/from a function as a parameter!!
```

Copy Constructor

Rule #1. Implement Copy Constructor if a class has a pointer!!!

```
SimpleArray(const SimpleArray & that) {
   length = that.length;
   data = new int[length];
   for(int i = 0; i < length; i++)
         data[i] = that.data[i];
}
```

```
SimpleArray s1(10);          // one arg constructor is used to build s1.
SimpleArray s2(s1);          // copy constructor is used to build s2.
SimpleArray s3 = s1;         // copy constructor is used to build s3.
s3 = s1;                     // Assignment operator= is used to change s3.
```

BAD Copy Constructor

```
// DO NOT DO THIS
SimpleArray(const SimpleArray & that) {
  *this = that;
}
```

You also need to define *operator=* as a member function if the class has pointers as its members.

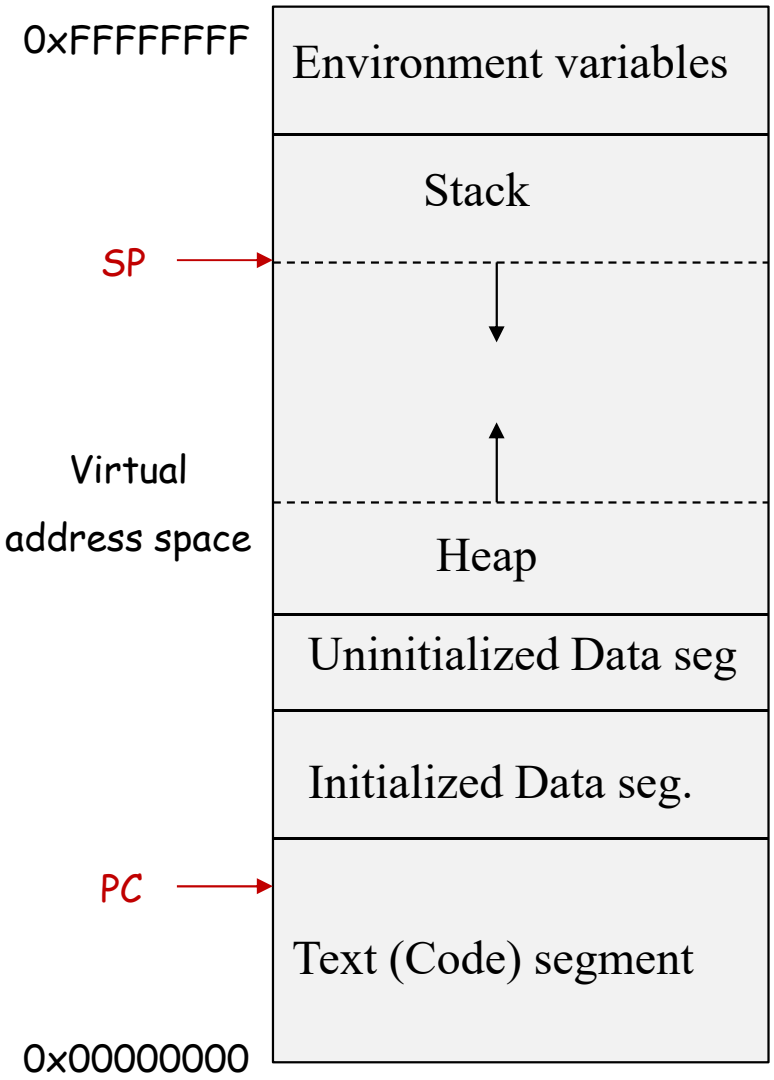
If your code uses class and pointers together, you better not use default member functions.

Lifetime and Storage Management

- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap

- Static allocation for
  - code
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

Memory Layout



- Your program is comprised of a set of memory segments.
  - Stack
    - Next slides
  - Heap
    - Dynamic memory allocation
  - Data segment
    - Initialized
      - Global variables and static variables that have initial values
    - Uninitialized
      - Often called BSS segment
      - Global variables and static variables without initial values
  - Text segment (or Code segment)
    - Your compiled binary code

Memory Layout

```
int f(char* m, float f){
    int n = (int) f;
    cout << m << ":" << n << endl;
}

int g1=0;
int g2;
int main(){
   float a = 1.2;
   char *msg = new char[10];
   strcpy(msg, "Hello");
   f(msg, a);
   delete[] msg;
}
```

- Q: Which segment will have each variable?

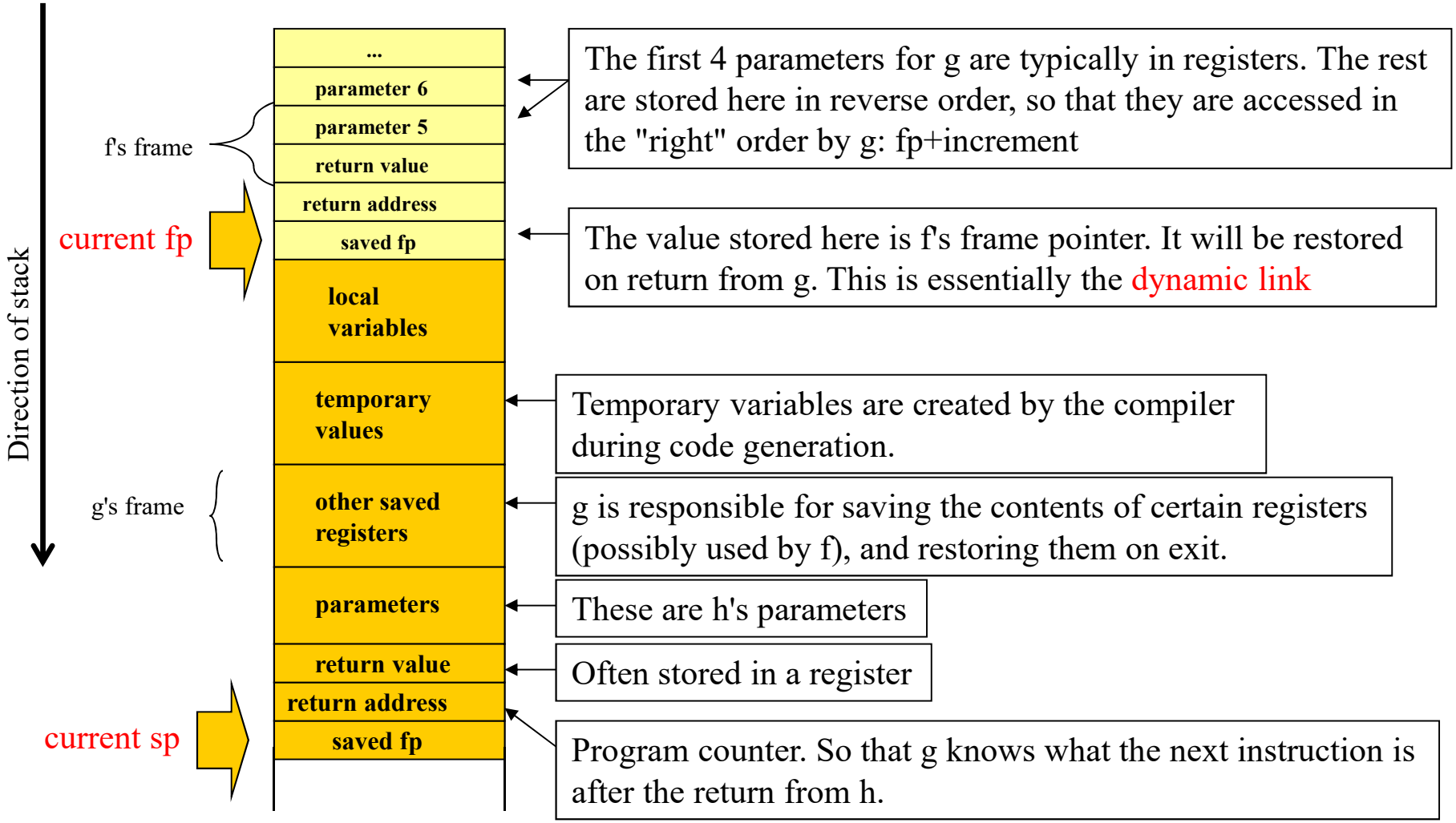
Lifetime and Storage Management

- Stack
  - The place where arguments of a function call are stored
  - The place where local data of called function is allocated
  - The place where called function leaves result for calling function
  - Supports recursive function calls

  - Why a stack?
    - The last function called will be the first one to exit.
    - allocate space for recursive routines
      (not necessary in FORTRAN – no recursion)
    - reuse space
      (in all programming languages)
    - Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Stack organization

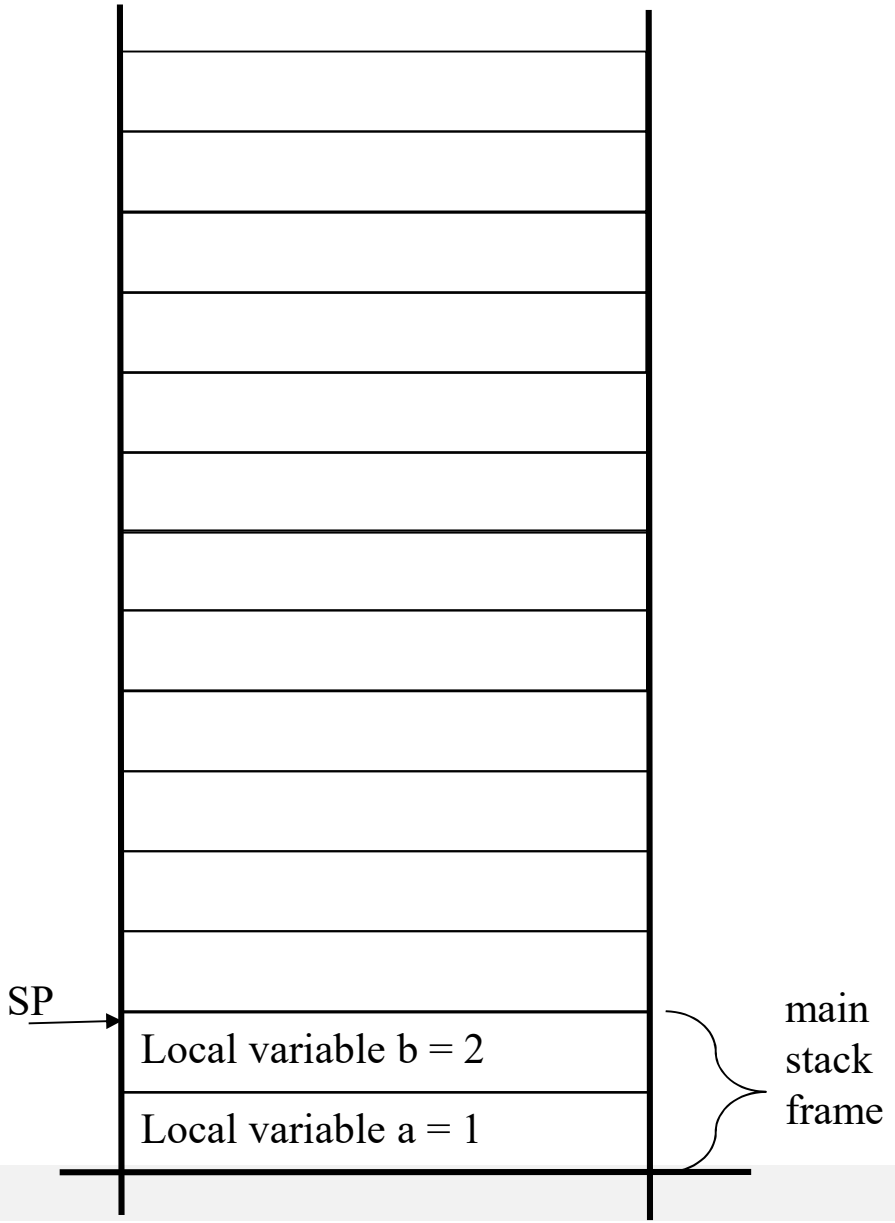
- Typical stack layout. Assume function f calls g, and g is about to call h



Stack and Function Calls

```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);
11: }
12:
13: int g(int n) {
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```

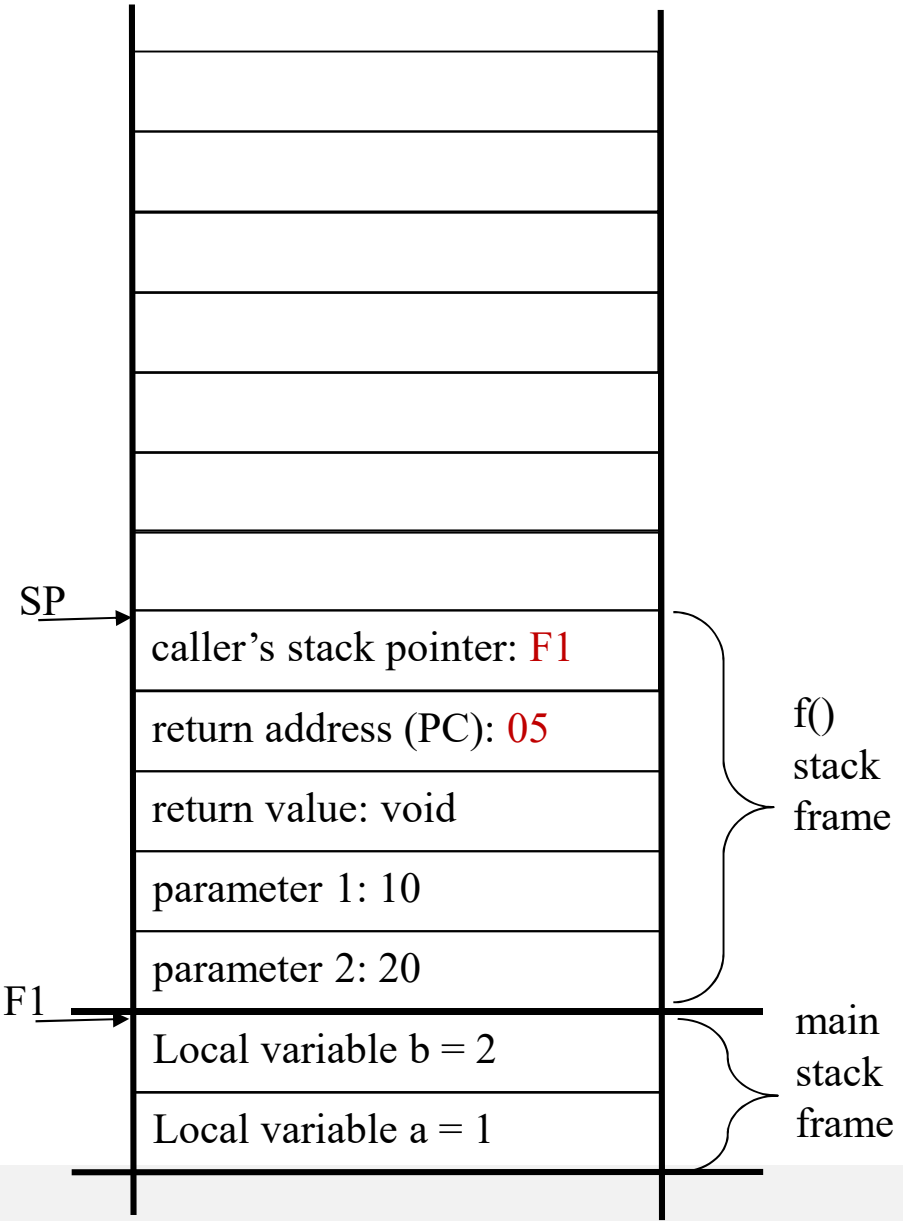
PC=04



Stack and Function Calls

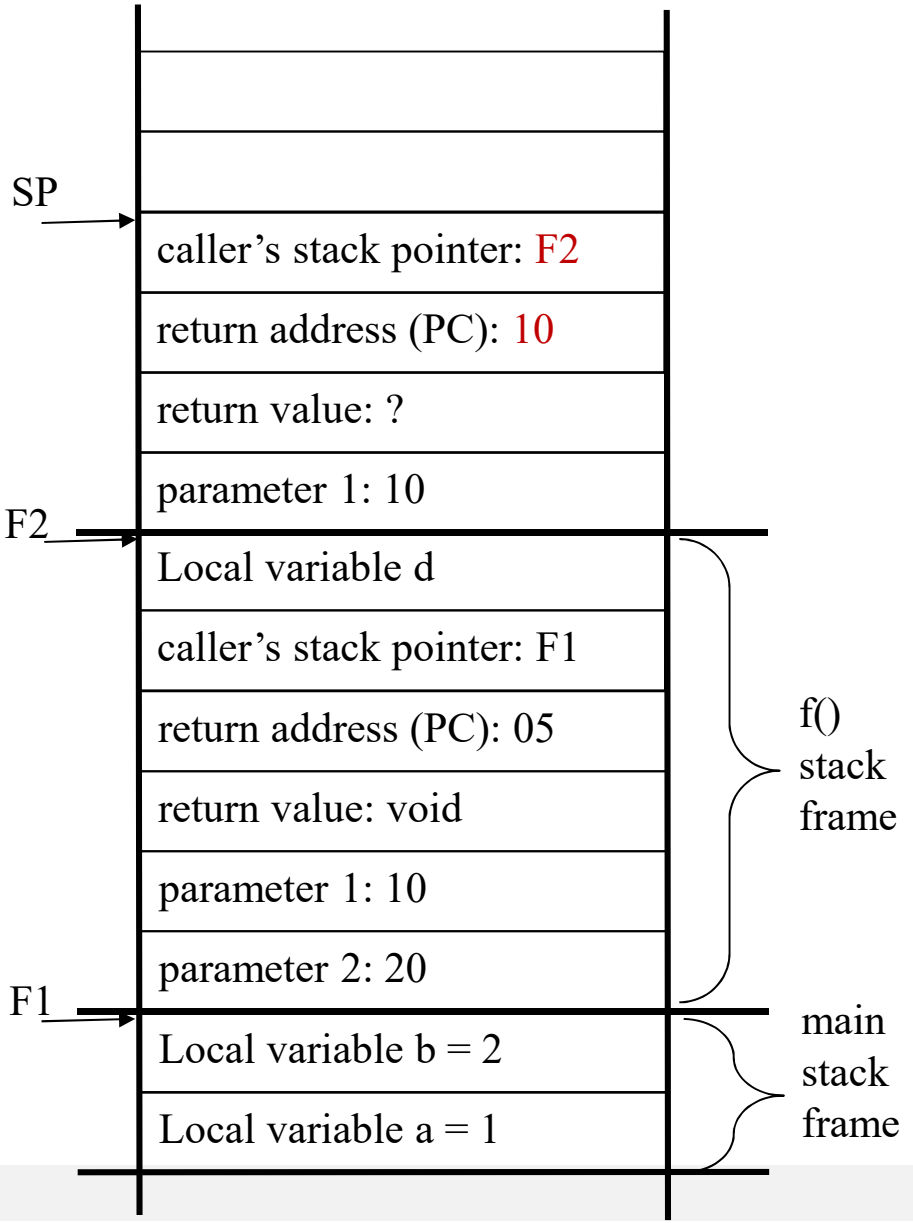
```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);
11: }
12:
13: int g(int n) {
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```

PC=08



Stack and Function Calls

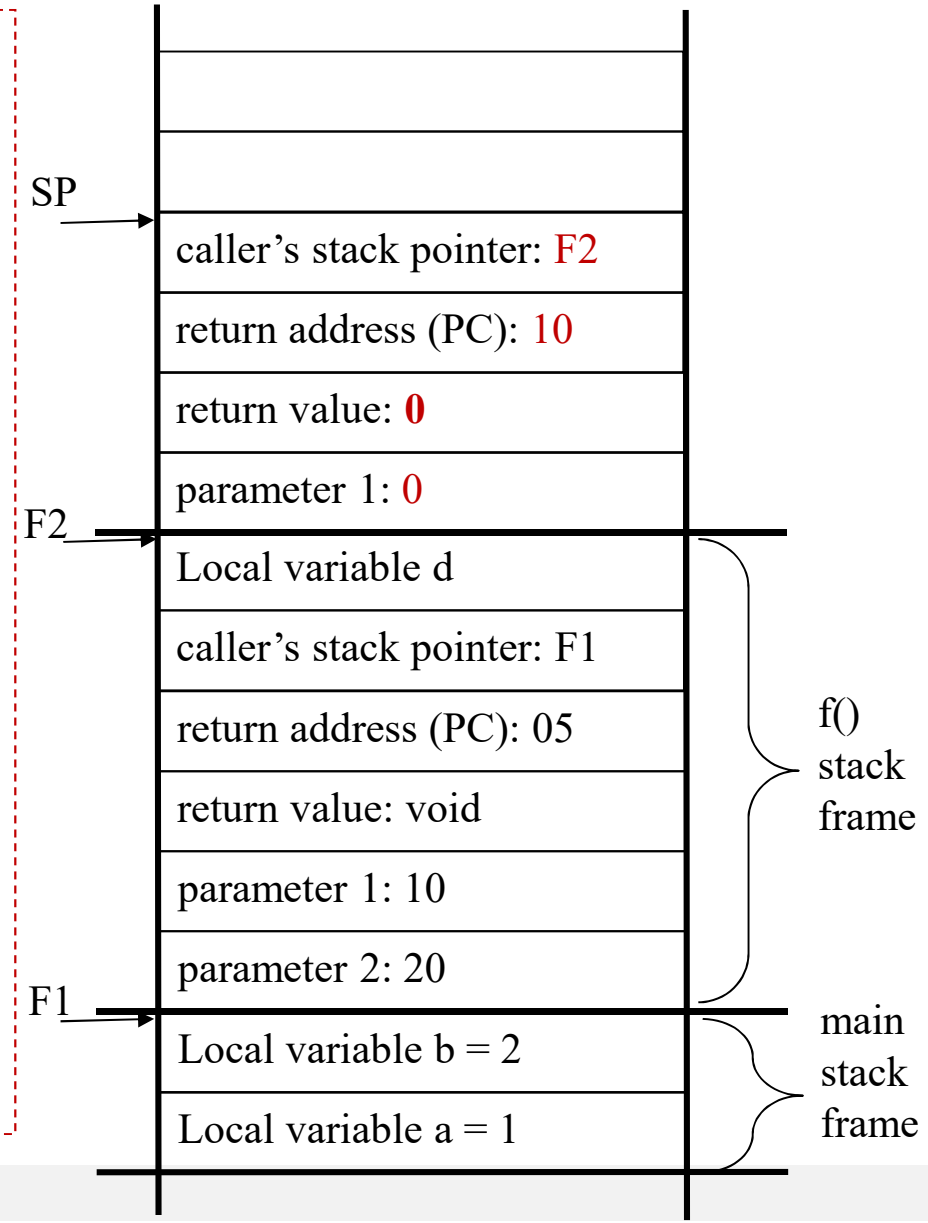
```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);
11: }
12:
13: int g(int n) {          PC=13
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```



Stack and Function Calls

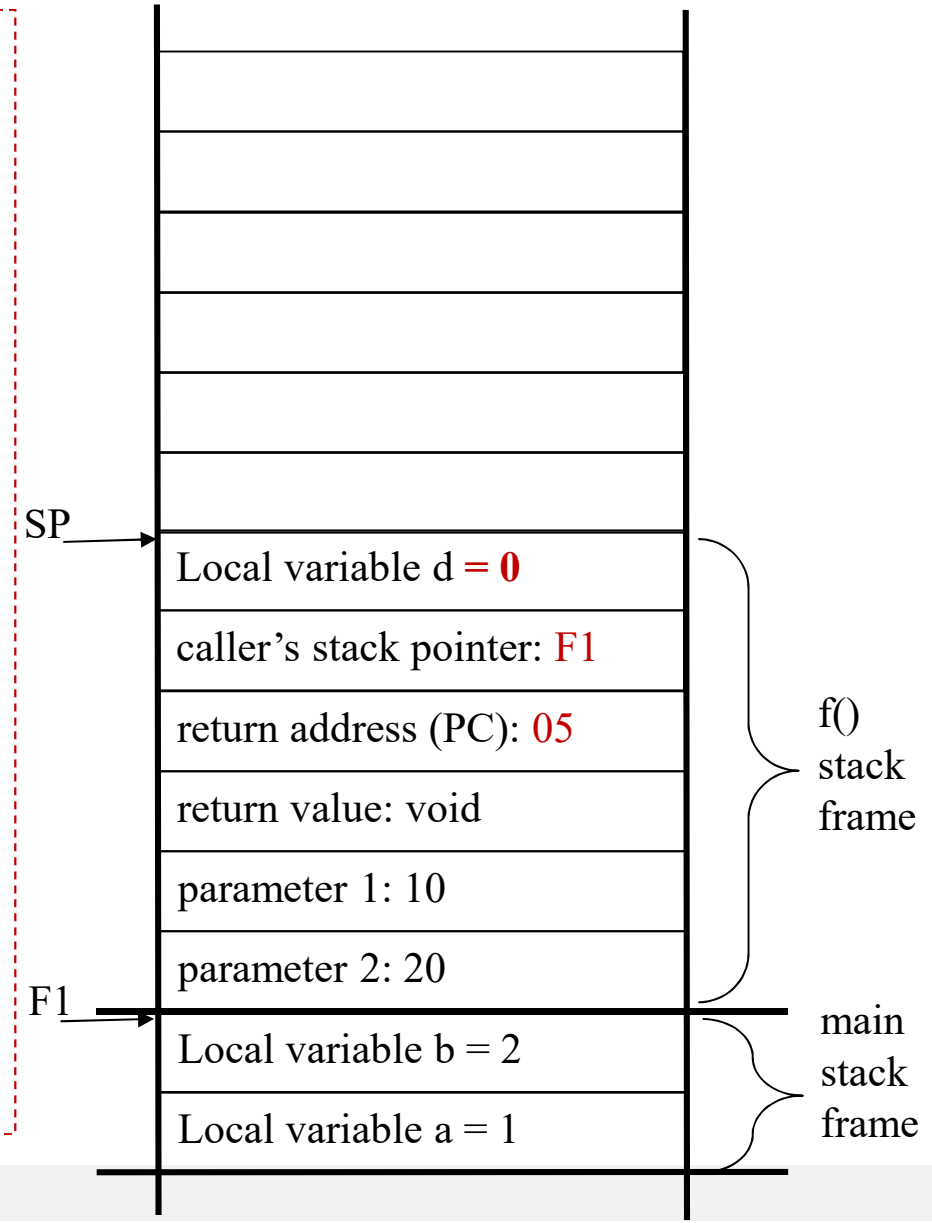
```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);
11: }
12:
13: int g(int n) {
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```

PC=18



Stack and Function Calls

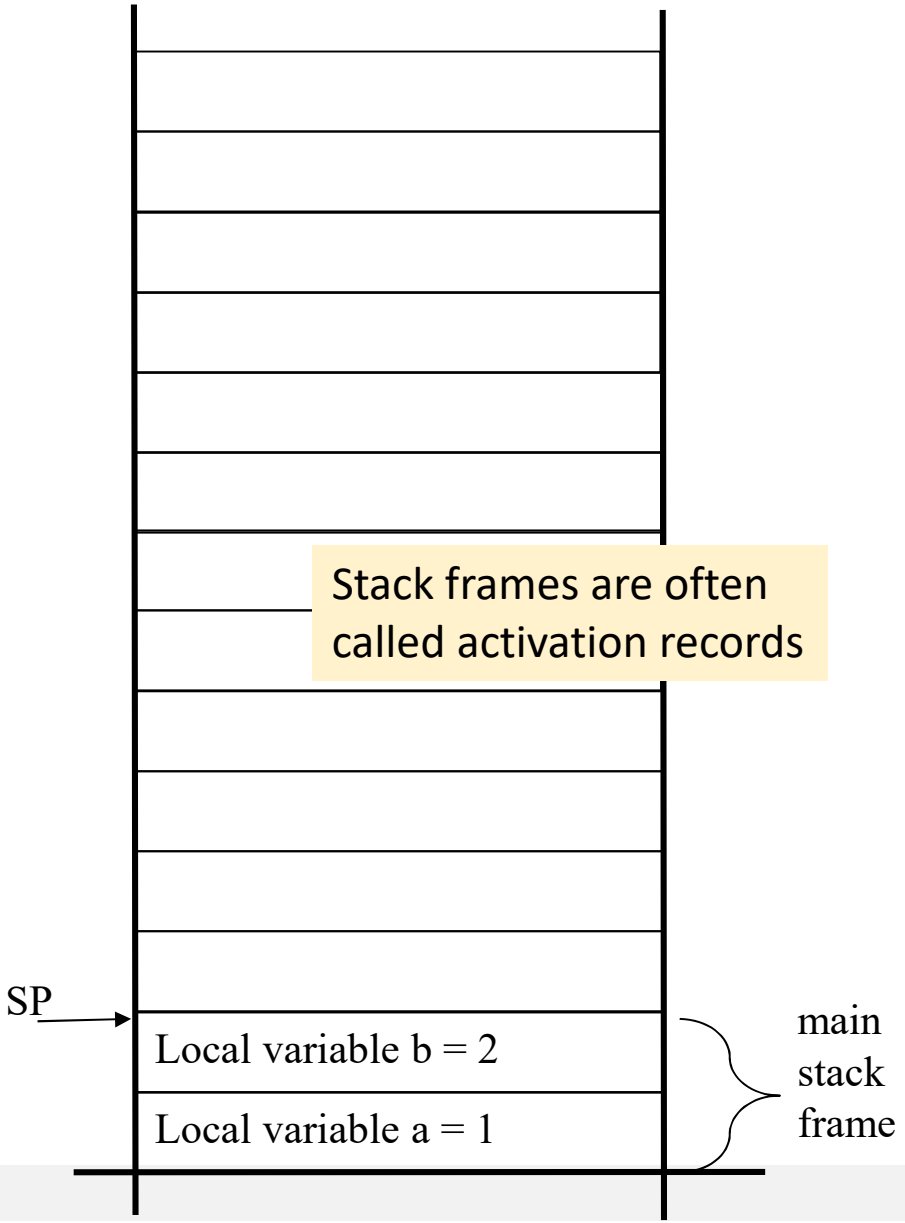
```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);          PC=10
11: }
12:
13: int g(int n) {
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```



Stack and Function Calls

```
01: int n= 2;
02:
03: int main() {
04:  int a=10,b=20;
05:  f(a, b);
06: }
07:
08: void f(int x, int y) {
09:  int d;
10:  d = g(x);
11: }
12:
13: int g(int n) {
14:  while (n > 0) {
15:    printf("%d", n);
16:    n--;
17:  }
18:  return n;
19: }
```

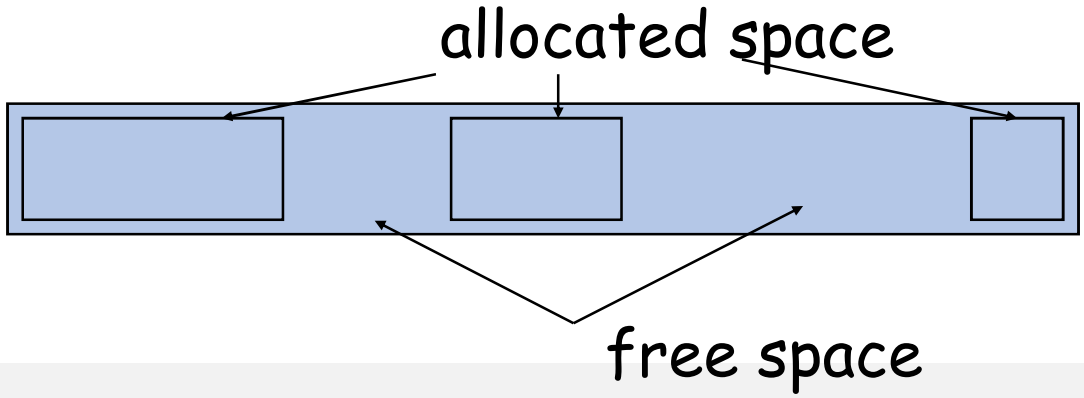
PC=5



Heap memory

- Heap memory
  - The heap is an area of memory for storing dynamic data
  - Heap memory can be allocated and de-allocated in any time while a program is running.

- Why use dynamically allocated memory?
  - Sometimes it's hard to know data sizes during compilation. (e.g, linked list)



Memory Allocation in C

- In C++: **`new`**
- In C:
  `void* malloc(size_t request);`
  - allocates **`request`** bytes of memory (if available) from heap
  - **`malloc`** doesn't perform any initialization of that memory space

`void* calloc(size_t request, size_t object_size);`

- allocates **`request`** objects of size **`object_size`** bytes each
- initialize all the memory to zero

- both return a memory address of the beginning of the space if memory can be allocated, otherwise return **`NULL`**.
- **`malloc`** and **`calloc`** both return a **`void *`** pointer.
  - Need typecasting

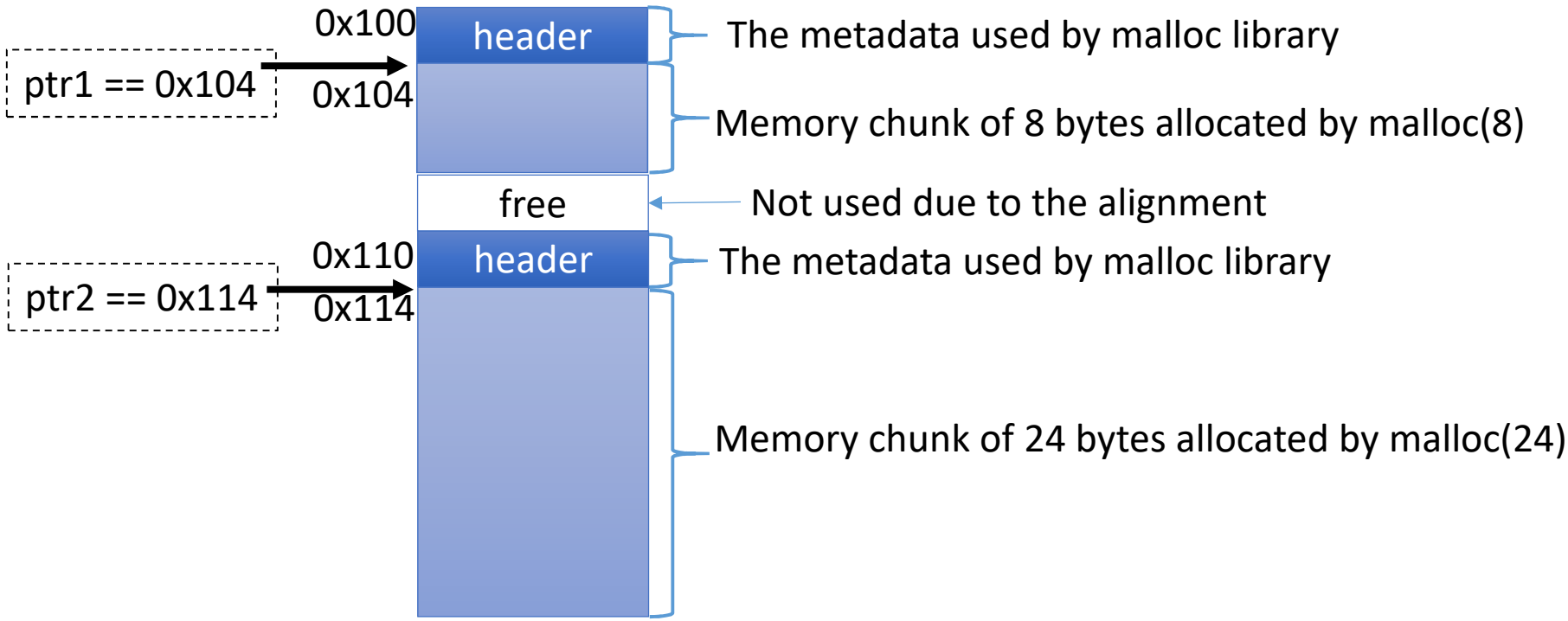
Implementing dynamic allocation

- How is the memory in the heap managed?
  - data structures need to be kept to:
    - keep track of what memory is in use, and
    - keep track of where the free memory regions are
  - calls to **`malloc()`**, **`free()`**, etc., update these structures

- Where does heap memory come from?
  - **`malloc()`**, etc make calls to the OS (system calls) to add memory to the heap of a running process
    - on Linux, this is **`void *sbrk(int incr)`**
      - adds at least **`incr`** bytes to the end of the program's data segment
    - the key idea is to get big chunks of memory from the OS and then hand out small regions to the program on demand
      - OS calls can be a hundred times slower than local subroutines

A simple data structure for a heap

- Divide the heap into blocks
  - Due to alignment requirements, allocated blocks must always begin at a memory address which is a multiple of 8
  - E.g.) ptr1 = malloc(sizeof(int)*2);
    ptr2 = malloc(sizeof(int)*6);



Tracking The Size Of Allocated Regions

- Q: Why free(void *ptr) does not need a size parameter?

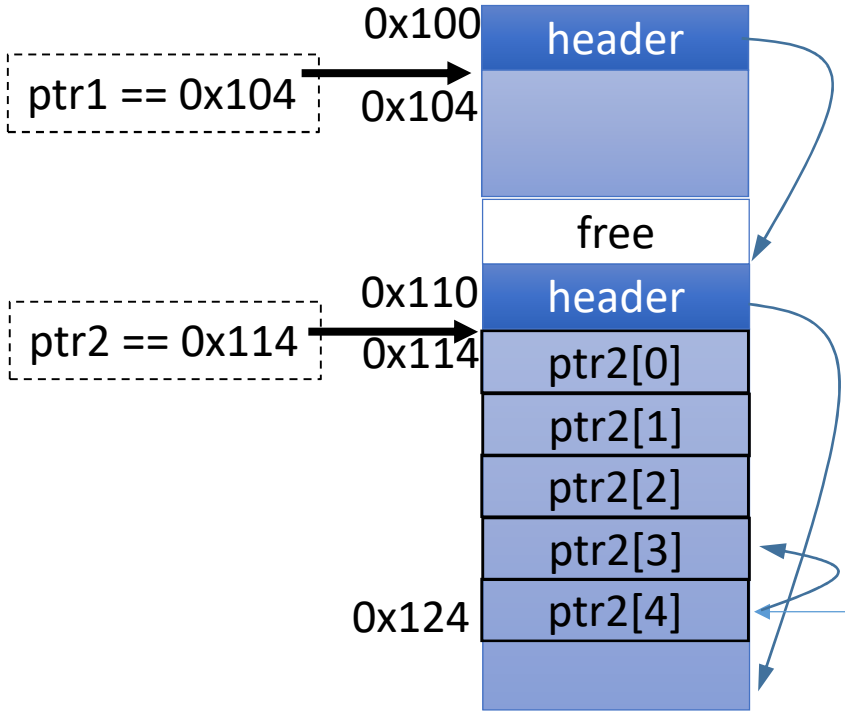
```
int main()
{
    int i, N=10;
    int *p = (int*) malloc(sizeof(int));
    int *arr = (int*) malloc(N*sizeof(int));
    printf("p=%x, arr=%x\n", p, arr);

    *p = 100;
    for(i=0;i<N;i++)
        arr[i] = i;

    free(p);
    free(arr);
}
```

A simple data structure for a heap

- Memory allocator stores a little bit of extra information in a header block
  - Size of block
  - Free list
  - free(ptr) reads the header from ptr - 4 to figure out how to free



Q: free(&ptr2[4]); // ?

free(&ptr2[4]); will try to read a header from 0x120, which fails to find a valid header.
→ Error.

Nonstandard gcc facilities for heap inspection and debugging

- Heap inspection functions
  - Useful when debugging memory leak problems.
  - to use these, **`#include <malloc.h>`**

`void malloc_stats(void);`

  - prints a summary of information about the heap, including the current memory available and the amount currently allocated

`struct mallinfo mallinfo(void);`

  - returns a structure of type **`struct mallinfo`**, defined in **`malloc.h`**, with various useful fields
  - the **`uordblks`** field contains the current total allocated space
  - the **`fordblks`** field contains the current total free space

Nonstandard gcc facilities for heap inspection and debugging

- The best way of checking heap - manual checking
  - allows detecting exactly where problems are
  - Put debugging printf() in your code

- To use gcc facilities, #include <mcheck.h>
  - call the mcheck(NULL) function initially
    - Call mprobe() on any pointer for a consistency check
    - If it doesn't return MCHECK_OK the block is invalid
  - call mcheck_pedantic(NULL) initially for entire heap check.
  - mcheck_check_all(void) checks the whole heap immediately

Memory Debugging on Linux: Valgrind

- Valgrind is a memory mismanagement detector.
  - It shows you memory leaks, deallocation errors, etc.
- How to use:

```
$ gcc -g test.c
$ valgrind --tool=memcheck --leak-check=yes
            --show-reachable=yes ./a.out

==19769== Memcheck, a memory error detector
…
==19769== 16 bytes in 1 blocks are definitely lost in loss
   record 1 of 2
==19769==    at 0x4A0610C: malloc (vg_replace_malloc.c:195)
==19769==    by 0x400513: main (test.c:15)
…
==19769== LEAK SUMMARY:
==19769==    definitely lost: 35 bytes in 2 blocks
…
```

Environments

- a program is more than just code
- Two parts:
  - IP: instruction part (code and static data)
  - EP: environment part
- Definition: An *environment* is a mapping from (named) entities to memory addresses
  - Stack is the architectural implementation of environment

Environments

- Top-level environment in ML (or Perl):
  - table mapping between names and objects
    - Value binding
    - Function binding
    - Type binding

	...
^	fn. to concatenate strings
floor	fn. to compute floor of real number
+	integer addition

Identifier Value

Program Example

- Built-in functions include all operations pre-defined (for ML, or Perl), on ints, reals, strings, etc.
- So, what happens to the environment as an ML program is executed?

Example (cont.)

val pi = 3.14159;

pi	3.14159
All top-level bindings	

Identifier Value

Example (cont.)

fun area(radius) = pi * radius * radius;

area	fn: real → real
pi	3.14159
All top-level bindings	

Identifier Value

Example (cont.)

val x = area(1.0);

First, bind argument to formal parameter

radius	1.0
area	fn: real → real
pi	3.14159
All top-level bindings	

Identifier Value

Example (cont.)

Then, execute function body, bind result, unbind formal parameter

x	3.14159
area	fn: real → real
pi	3.14159
All top-level bindings	

Identifier Value

Environments

- *Environment* is *a set of binding occurrences* that are accessible at a point in the program.
- Can also think of it as a table that maps names to objects
- So, the process of executing a program changes the environment as the program runs

Environment Changes

- Question is: how does a procedure call change the environment?

- Let's use a static and dynamic scoping example.
  - Static scoping - a non-local variable x is bound to the declaration of x in the nearest enclosing procedure/function/block/…
  - Dynamic scoping - a non-local reference x is bound to the most recent declaration of x
    - the most recently called subprogram that declares x

Static Scoping Example

- The environment changes as the program runs:

```
procedure p;
  var i: integer;  (* I1 *)

  procedure q;
    begin
      i := i + 1;
    end;
  procedure r;
    var i: integer;  (*I2 *)
    begin
      i := 5;
      q;
    end;

begin
  i := 0;
  r;
  q;
  write(i);
end;
```

Procedure	Identifier	Value
q		
~~q~~		
~~r~~	~~i~~	~~I2 = 5~~
p	i	I1 = 0 1 2

Dynamic Scoping Example

```
procedure p;
  var i: integer;  (* I1 *)

  procedure q;
    begin
      i := i + 1;
    end;
  procedure r;
    var i: integer;  (*I2 *)
    begin
      i := 5;
      q;
    end;

begin
  i := 0;
  r;
  q;
  write(i);
end;
```

Procedure	Identifier	Value
q		
q		
r	i	I2 = ~~5~~ 6
p	i	I1 = ~~0~~ 1

Static Scoping, again

- More precise definition of static scoping
  - a non-local variable x is bound to the declaration of x in the most recent activation of the nearest enclosing subprogram
    - Assume Dynamic Binding
- Best shown with an example

Static Scoping Question

```
procedure P(x: integer);
  var i: integer;

  procedure Q;
    begin
      i := 0;
    end;

begin
  i := 3;
  if (x > 0) then
    P(x-1)
  else
    Q;
  write(i);
end;
```

What will be printed out for P(2); ?

Static Scoping Question

```
procedure P(x: integer);
  var i: integer;

  procedure Q;
    begin
      i := 0;
    end;

begin
  i := 3;
  if (x > 0) then
    P(x-1)
  else
    Q;
  write(i);
end;
```

The environment for a call P(2):

Q		
P(0)	i x	3 0 0
P(1)	i x	3 1
P(2)	i x	3 2

Procedure Identifier Value

And the procedure prints 0, 3, 3 for **i**