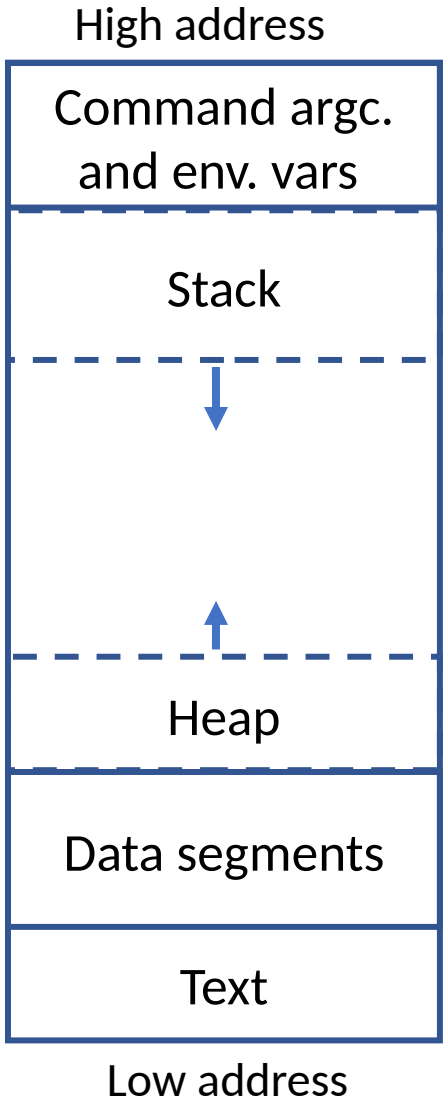
Memory Layout of a C Program

Eike Ritter and Aad van Moorsel
School of Computer Science
University of Birmingham

Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. Stack segment
4. Heap segment

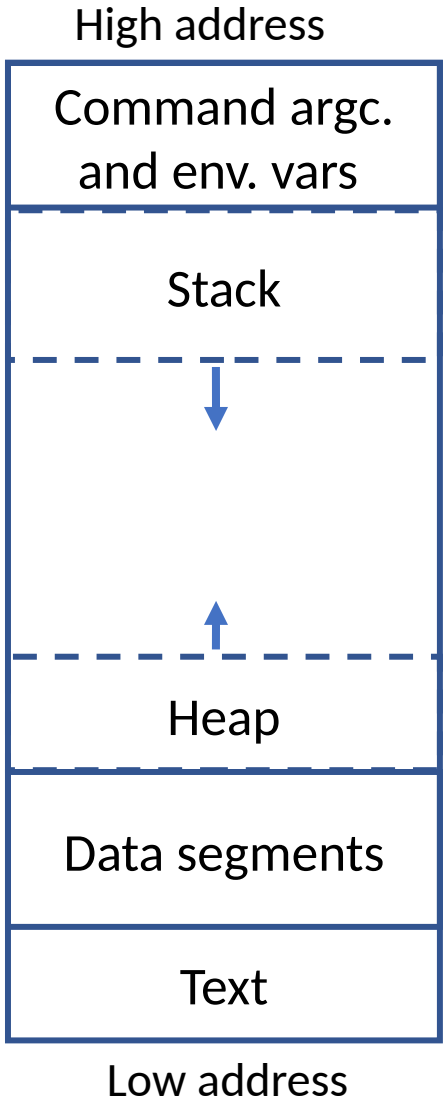


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. **Text or code segment**
2. Data segments
3. Stack segment
4. Heap segment

Text segment contains the the program i.e. the executable instructions.

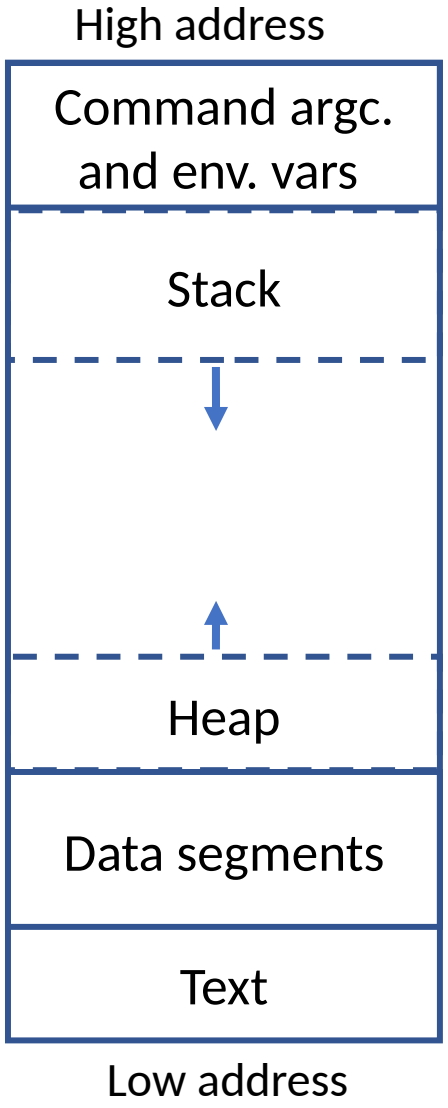


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. **Data segments**
3. Stack segment
4. Heap segment

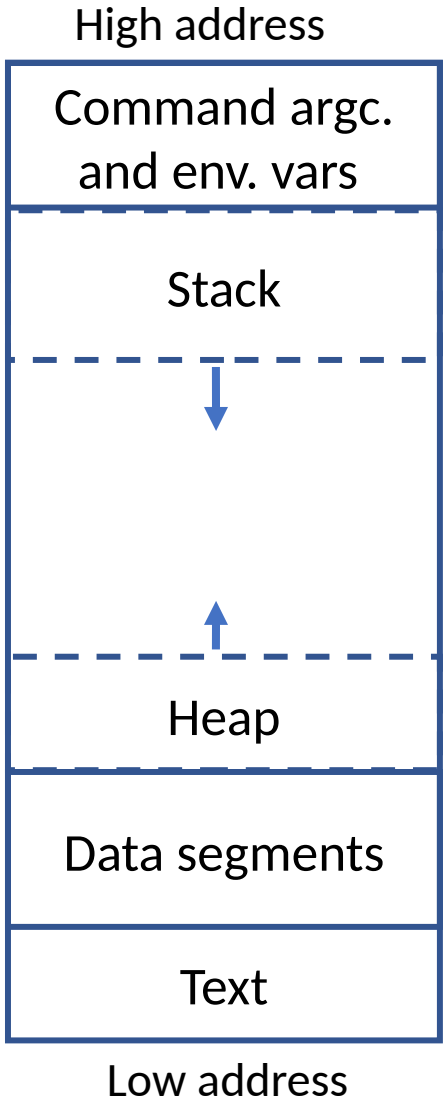
Two data segments contain initialized and uninitialized global and static variables respectively.



Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. **Stack segment**
4. Heap segment



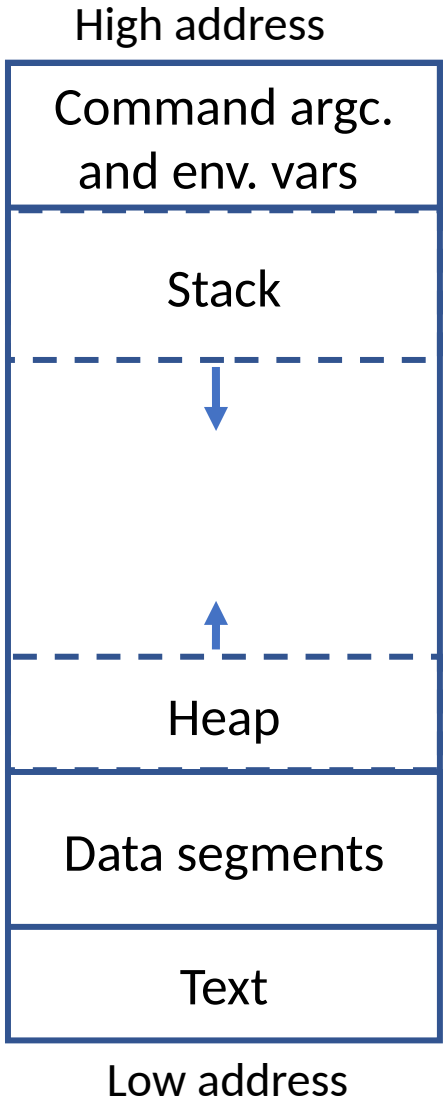
Stack segment is used to store all local or automatic variables. When we pass arguments to a function, they are kept in stack.

Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. Stack segment
4. **Heap segment**

Heap segment is used to store dynamically allocated variables are stored.



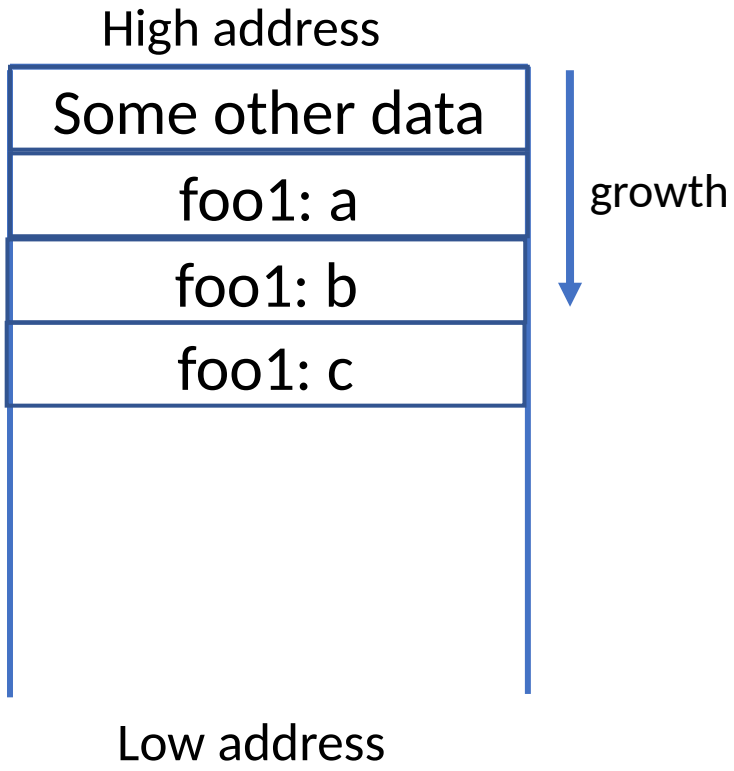
The use of Stack and Heap will be discussed in detail.

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

1. Before foo2() is called: Stack has data of foo1()

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



Stack during function calls

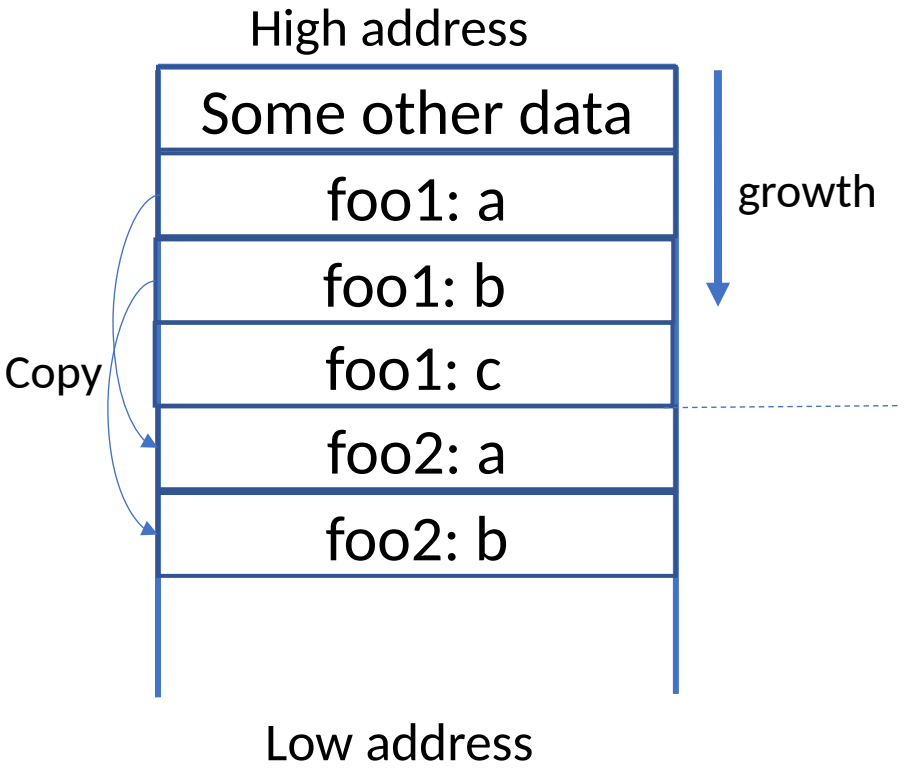
Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

2. When foo2() is called: New stack frame for foo2() created

3. Function arguments are copied

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



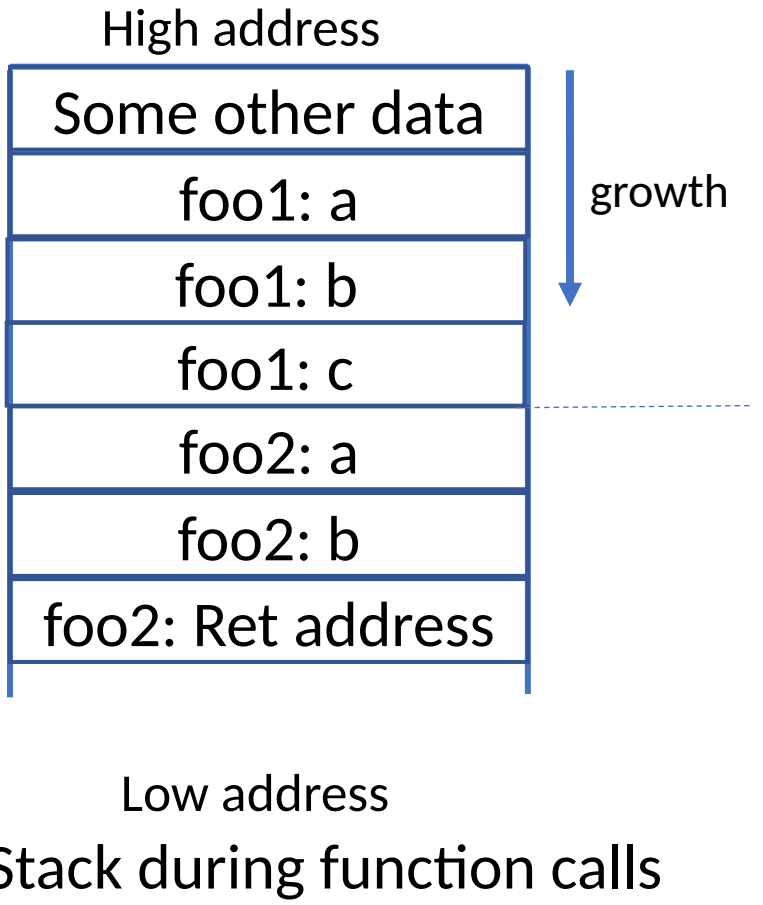
Stack during function calls

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

4. The address of the instruction that will be executed from foo1() just after foo2() finishes, is copied. This is called 'Return address'.

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



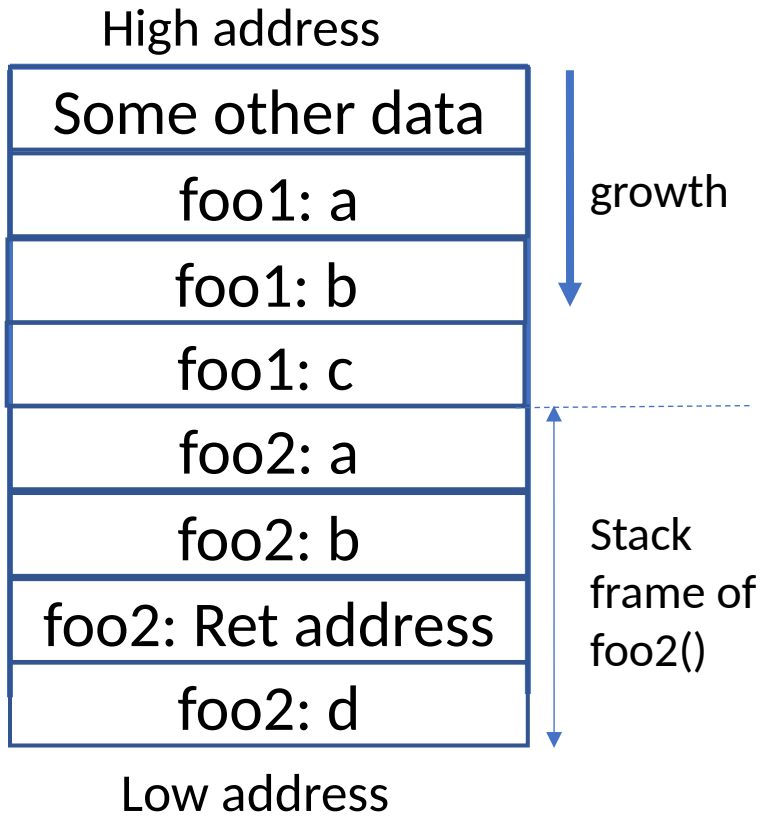
Stack during function calls

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

5. Variables within foo2(), which are called ‘local variables’ are stored.

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



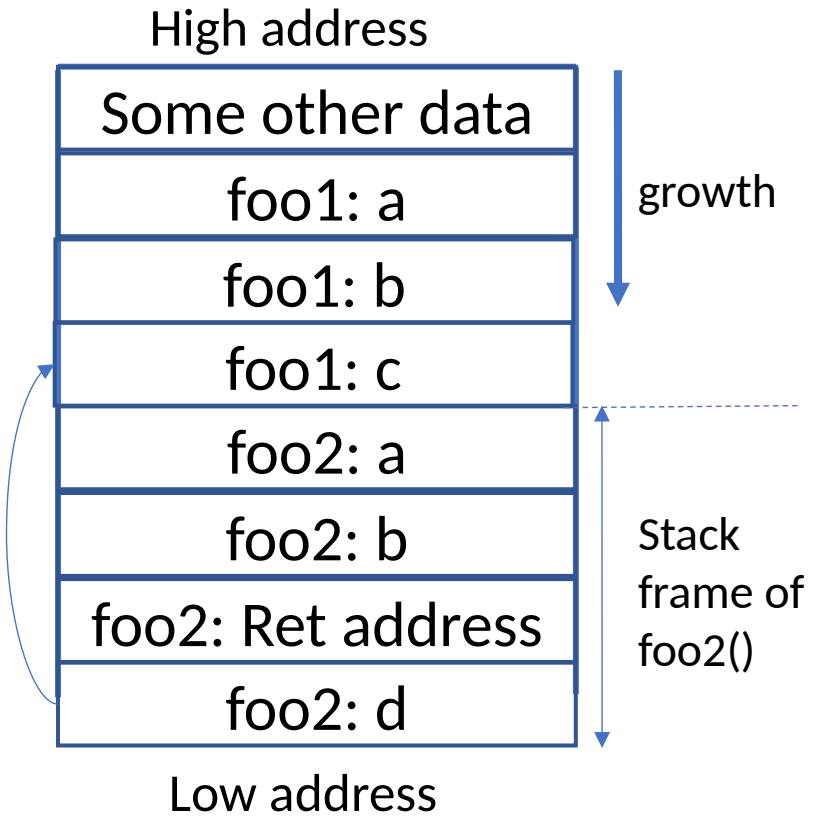
Stack during function calls

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

6. After foo2() finishes, local variable d is copied into c of foo1().

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



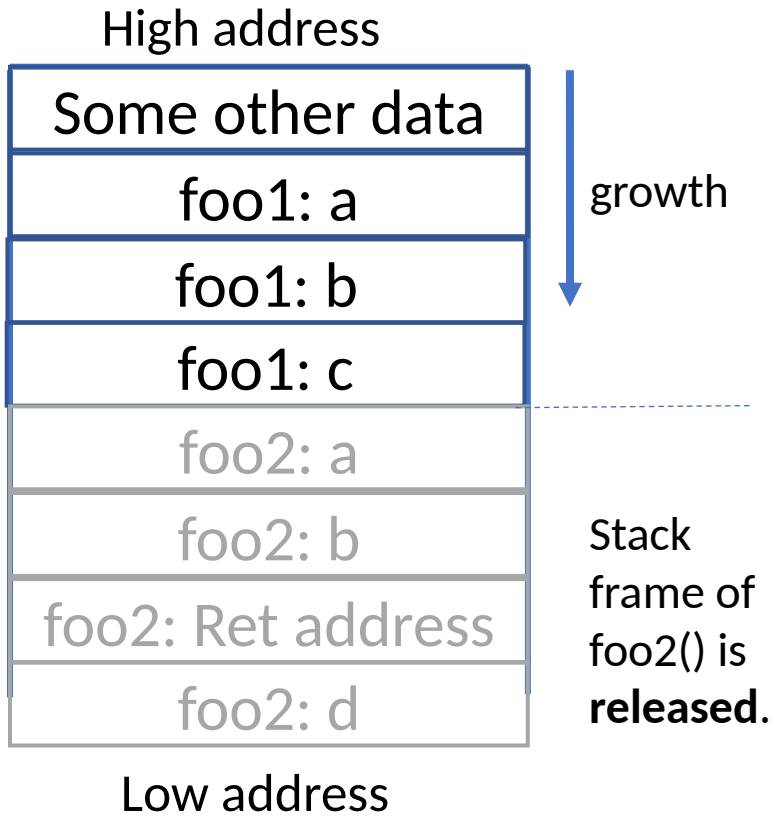
Stack during function calls

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

7. Following the return address, control-flow returns to foo1().
8. Stack frame for foo2() is released. Hence, all local variables die.

```
T foo2(T a, T b){
   T d;
   ...
   ...
   return d;
}
foo1(){
   T a, b, c;
   c = foo2(a, b);
   ...
}
```



Stack during function calls

Stack during function call: Scope

- Scope: part of the program where a variable can be used (or seen)

- Local variables within a function: scope is the function.
  They are allocated in the stack-frame of the function. After the function call, the stack-frame is released.

```
T foo2(T a, T b){
   T d; // scope of is foo2
   ...
   ...
   return d;
}
foo1(){
   T a, b, c; // scope of is foo1
   c = foo2(a, b);
   ...
}
```

Static variables

- Static variables are stored in the data segment, **not in the stack**.
- The data segment is active during the entire life-time of program
- So, static variables preserve their values even after they are out of their scope.

```
foo(int b){
  int a=0;
  a = a+b;
  printf("a=%d", a);
}
main(){
  foo(1);
  foo(1);
}
```

Scope of ‘a’ is foo().
Outputs will be 1 and 1 both times.

```
foo(int b){
  static int a=0;
  a = a+b;
  printf("a=%d", a);
}
main(){
  foo(1);
  foo(1);
}
```

Static ‘a’ is preserved.
Outputs will be 1 and 2.

Global variables

- A global variable is declared outside all functions.
- It can be read or updated by all functions.
- Careful: Do not name any local var in the name of a global var.

```
int A_g = 5; // Global variable

int foo(int b){
   int a=0;  // Local variable
   a = a+b+A_g;
   return a;
}

main(){
   int a, b=1;  // Local variable
   a=foo(1);
}
```

```c
void swap(int x, int y){
   int temp;
   temp = x;
   x = y;
   y = temp;
}
int main(){
   int a=4, b=5;
   swap(a, b);
   printf("a=%d b=%d", a, b);
   return 0;
}
```

What values of a and b will be printed?

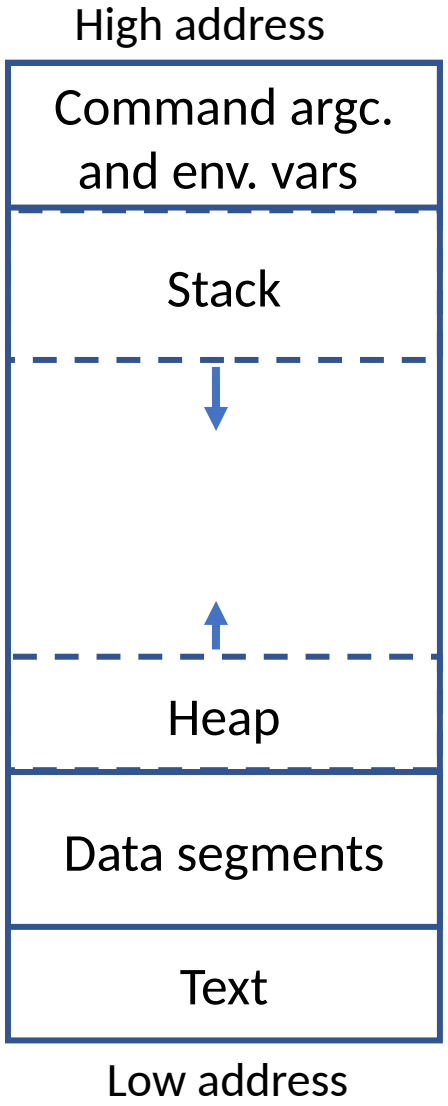
```c
void swap(int x, int y){
   int temp;
   temp = x;
   x = y;
   y = temp;
}
int main(){
   int a=4, b=5;
   swap(a, b);
   printf("a=%d b=%d", a, b);
   return 0;
}
```

Changes are local,
not visible from main().

The program will print a=4 and b=5.

Conclusions so far

- Local variables use Stack ‘temporarily’. They are active only within their functions.
- Static variables are stored in the Data segments. They preserve their values.
- Global variables are stored in the Data segments. They are accessible to all functions of the C program.



Use of Heap will be covered in Dynamic Memory Management.