Section 3
Memory Management

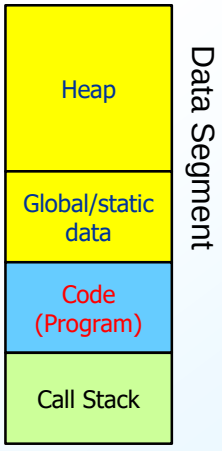
1. Stack and heap
2. Dynamic memory allocation
3. Linked lists

Section 3.1
Stack and Heap

1. Overview
2. Function call stack
3. Heap
4. Memory allocation

3.1.1 Overview

- OS allocates four areas of memory on program startup
  - code segment (text segment)
    - program instructions
  - data segment
    - global memory
  - function call stack
    - local data
  - heap (part of data segment)
    - dynamically allocated memory



Overview (cont.)

- Code segment
  - program instructions
  - addresses of functions
- Data segment
  - global variables
  - **static** variables
  - literals

- are local to a function/block but survive the scope (namely, {})

Overview (cont.)

- Function call stack
  - manages order of function calls
  - stores local variables
- Heap
  - part of the data segment
  - stores all dynamically allocated memory
    - memory allocated at runtime

3.1.2 Function Call Stack

- What is a stack?
  - data structure
  - collection of related data

- Stack data structure
  - analogous to a pile of dishes
  - order is last-in, first-out (LIFO)
    - last item added (pushed) is the first item removed (popped)

Function Call Stack (cont.)

- What is the function call stack?
  - used to manage the function call and return mechanism
- Function call stack contains:
  - automatic variables
    - local variables
    - function parameters
  - return address in calling function

Function Call Stack (cont.)

- Control flow: order in which instructions are executed
- Control flow in single-threaded C program
  - begins at first instruction in `main` function
  - continues sequentially to next instruction
  - function calls:
    - control is transferred to first instruction of called function
    - upon return, control is transferred back to calling function

Function Call Stack (cont.)

- Function call and return mechanism
  - when a function is called
    - an activation record is *pushed* onto the stack
    - activation record also known as **stack frame**
  - activation record contains:
    - address of instruction in calling function where control will return
    - automatic variables in called function
  - when the called function returns
    - its activation record is *popped* off the stack
      - automatic variables are lost!
    - control transfers to the return address in the calling function

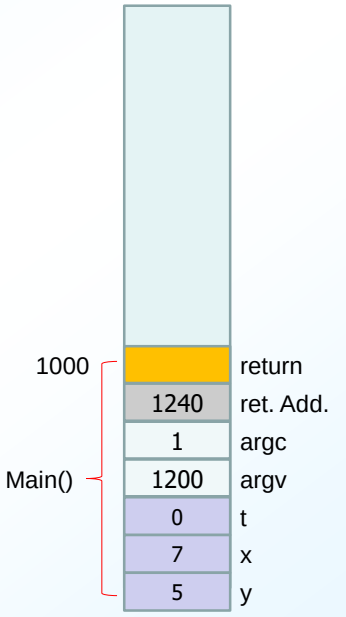

```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```

Frame Stack



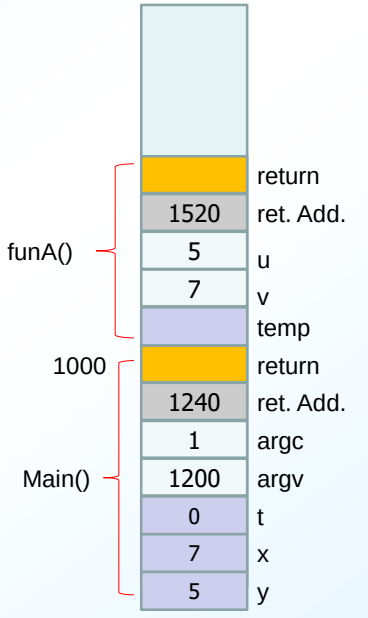
```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```

Frame Stack



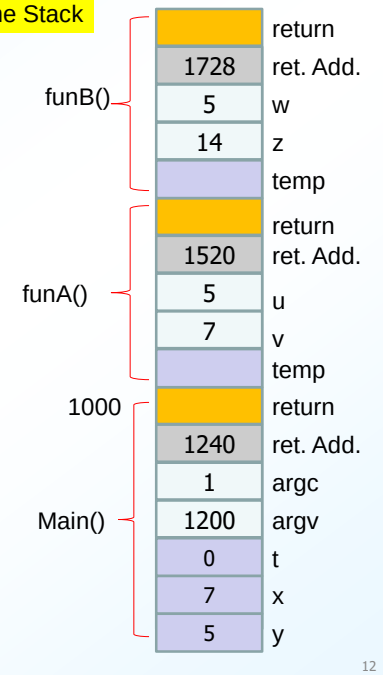

```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```

Frame Stack

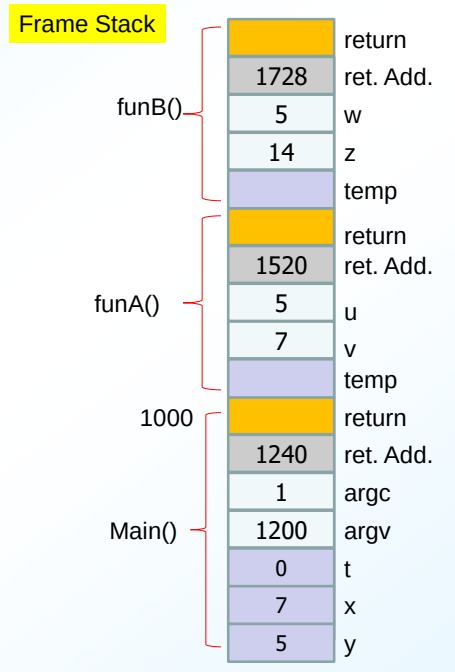


```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```

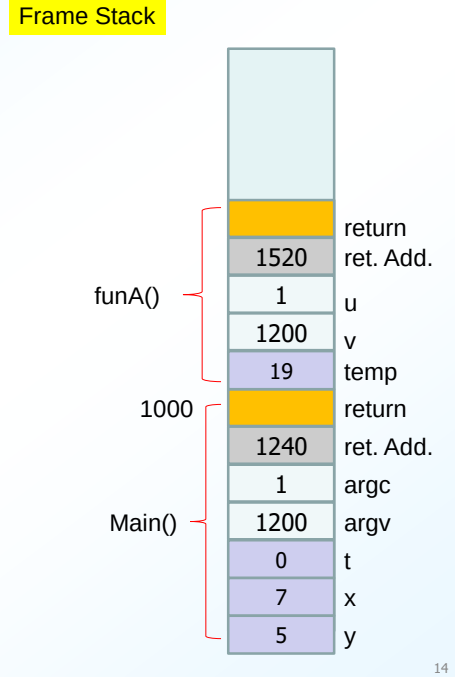



```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```

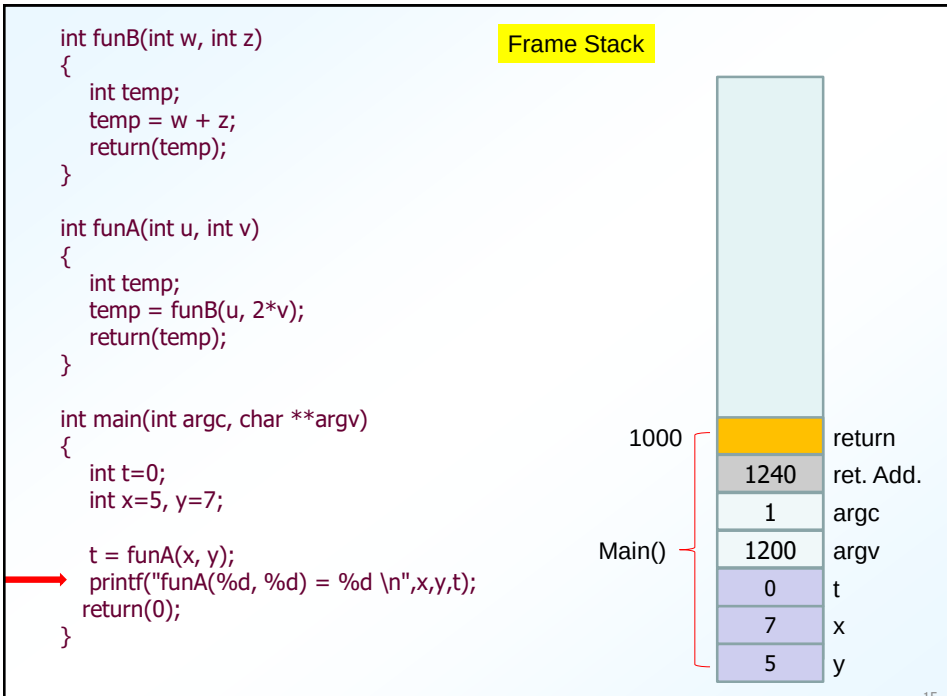


```
int funB(int w, int z)
{
    int temp;
    temp = w + z;
    return(temp);
}

int funA(int u, int v)
{
    int temp;
    temp = funB(u, 2*v);
    return(temp);
}

int main(int argc, char **argv)
{
    int t=0;
    int x=5, y=7;

    t = funA(x, y);
    printf("funA(%d, %d) = %d \n",x,y,t);
   return(0);
}
```



3.1.3 Heap

- What is the heap?
  - block of memory used for dynamic allocation
- Terminology
  - static: at compile time
  - dynamic: at runtime

Heap (cont.)

- Common problem with dynamically allocated memory
  - memory leaks
- What is a memory leak?
  - dynamically allocated memory that is not deallocated
- Result
  - program crashes when it runs out of heap memory

Heap (cont.)

- Preventing memory leaks
  - explicitly deallocate dynamically allocated memory
    - neither OS nor compiler will do it for you
  - never lose or overwrite pointer into heap
    - be careful with pointers into heap stored on function call stack 
    - hints:
      - deallocate before return to calling function
      - return pointer value to calling function

3.1.4 Memory Allocation

- What is memory allocation?
  - reserving (allocates) bytes in memory
- Types of memory allocation
  - static
    - allocated at compile time
  - dynamic
    - allocated at runtime

Static Memory Allocation

- How to allocate memory at compile time
  - programmer declares variables
  - compiler reserves number of bytes according to data type
- Problems
  - once allocated, memory **cannot** be resized !
  - may not know how much memory is needed for some variables
  - May not have enough memory
    - functions are limited by frame stack size

Static Memory Allocation (cont.)

- Solution
  - wait until runtime to reserve number of bytes in memory
  - how? dynamic memory allocation