Welcome to Data Structures(ECE20010/ITP20001)

- Youngsup Kim
  idebtor@gmail.com
- Jaehoon Lee
  21400575@handong.edu

Welcome to NowIC Jump Start

Part 1 - Data
Part 2 - Algorithmic Construct
Part 3 - Preprocess, Compile, Link

Youngsup Kim, idebtor@gmail.com, Handong Global University

The C Language Spirit

- Created by Dennis Ritchie and Ken Thompson between 1969 and 1973 at AT&T Bell Labs and used to re-implement the Unix operating system.
- Very flexible, very efficient, very liberal
  - Does not protect the programmers from themselves.
    Rationale: programmers know what they are doing even if looks bad enough to deserve a "Darwin award".
- Unix, most "serious" system software (servers, compilers, etc) and some programming languages themselves are written in C.
- Can do everything Java and C++ can. It'll just look uglier in C.

Programs

- Define
  - Data types and variables
  - Algorithms for manipulating those variables
- Example

```
int main(int argc, char *argv[]) {
  int  year;
  year = 2020;
  printf ("hello, world. This is %d\n", year);
  return 0;
}
```

Programs

- Define
  - Data types and variables
  - **Algorithms** for manipulating those variables
- Example

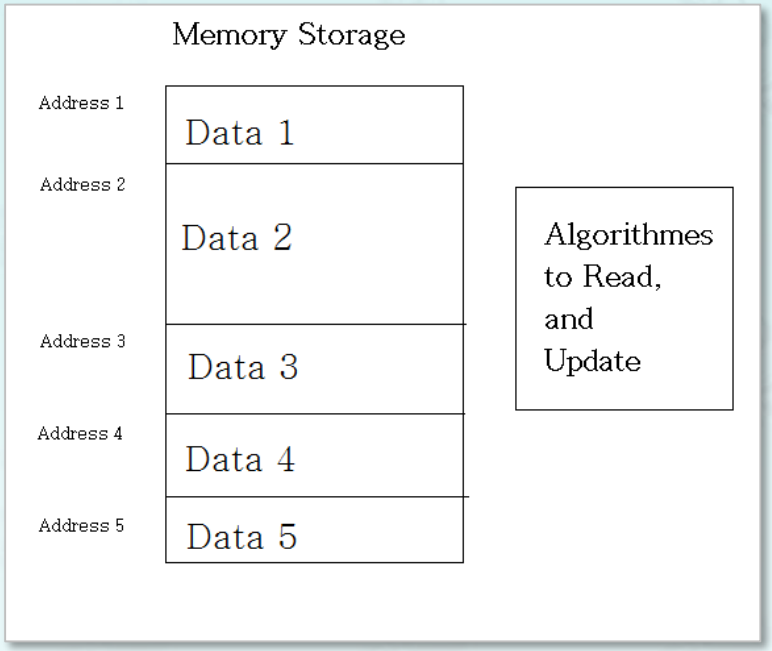
```
int main(int argc, char *argv[]) {
  int  year;
  year = 2020;
  printf ("hello, world. This is %d\n", year);
  return 0;
}
```

Declaration of variables:

Assignment statement

Format string

A Memory View of a Program



A **program** is a set of variables and a set of instructions that read/update them

Welcome to NowIC Jump Start

The C Programming Language

Part 0 – Why C?
Part 1 – Data
Part 2 – Algorithmic Construct
Part 3 – Preprocess, Compile, Link

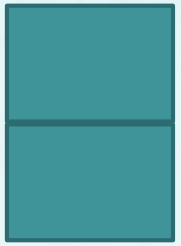
Youngsup Kim, idebtor@gmail.com, Handong Global University

Basic Data Types

- Integers : int (4byte)

- Characters : char(1byte)

- Floating point variables : float(4byte), double(8byte)

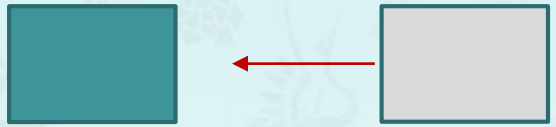


- Pointers (contain a memory address) :
  int*, char*, float*, void*, …

  - Arrays
  - Strings

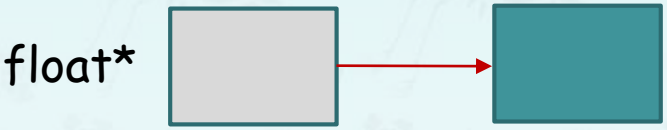
Basic Data Types

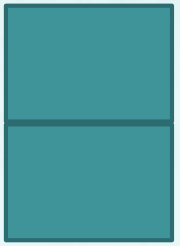
- Integers : int (4byte)



- Characters : char(1byte)

- Floating point variables : float(4byte), double(8byte)





- Pointers (contain a memory address) :
  int*, char*, float*, void*, …

  - Arrays
  - Strings

Data Variable Declaration

Format:

typename varname, varname, …;

```
int x, y;
float* p;
int z, *q;
char* c, m;
```

c is pointer to char
m is char, NOT a pointer!


```
int x, y;
float *p;
int z, *q;
char *c, m;
```

preferred

Pointers

- A pointer is **a variable** which points to data at a specific location in memory.
- A pointer has **a type**;
  this is the type of data it is pointing to.
- Key to doing many interesting things in C, such as functions that can change the value of a variable and dynamic memory management(more on memory in lecture)
- Can have a pointer to a pointer (to a pointer to a …)
- Can have a pointer to a **function**

More on Pointers

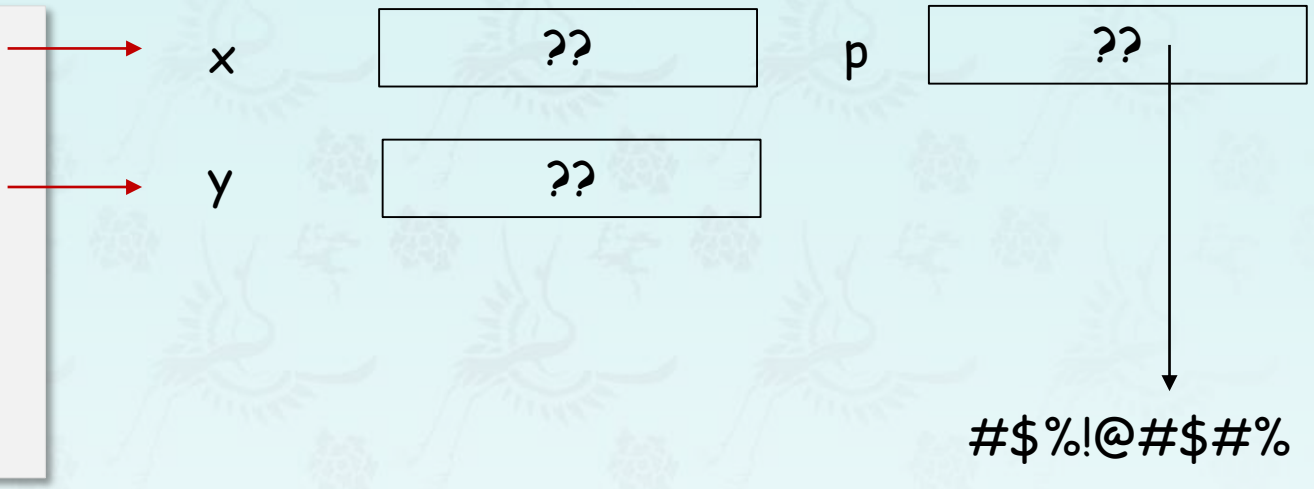
- The "&" (i.e. "**address of**") **operator** before a variable returns the memory address of the variable. This address is a pointer.

- The "*" (i.e. "**thing pointed to by**") **operator** before a pointer returns the variable that the pointer points to.

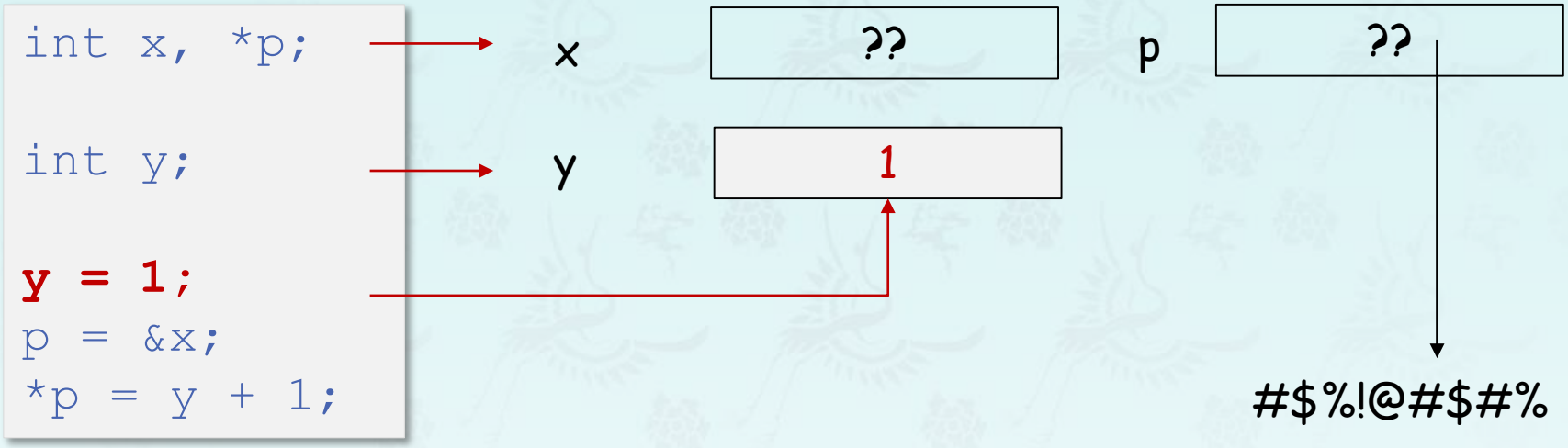
```
int x, *p;
int y;
y = 1;
p = &x;
*p = y + 1;
```

Tracing the Example

```
int x, *p;

int y;

y = 1;
p = &x;
*p = y + 1;
```

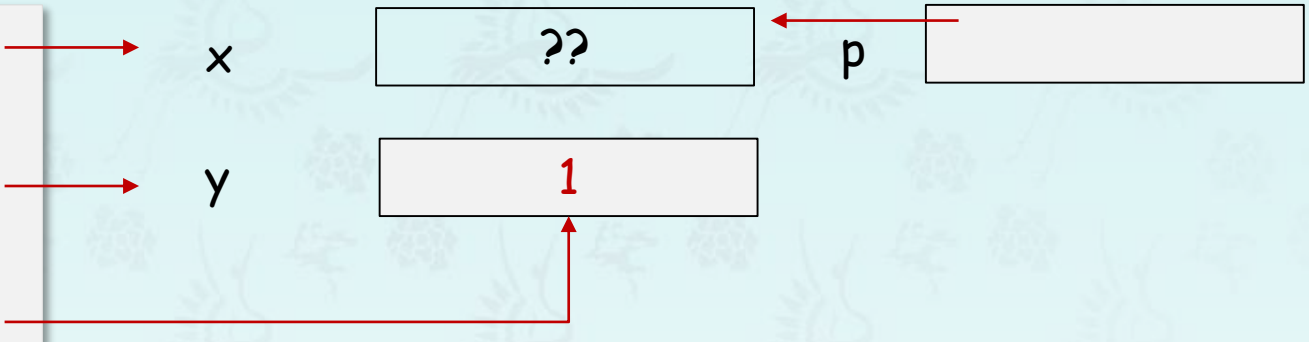


Tracing the Example



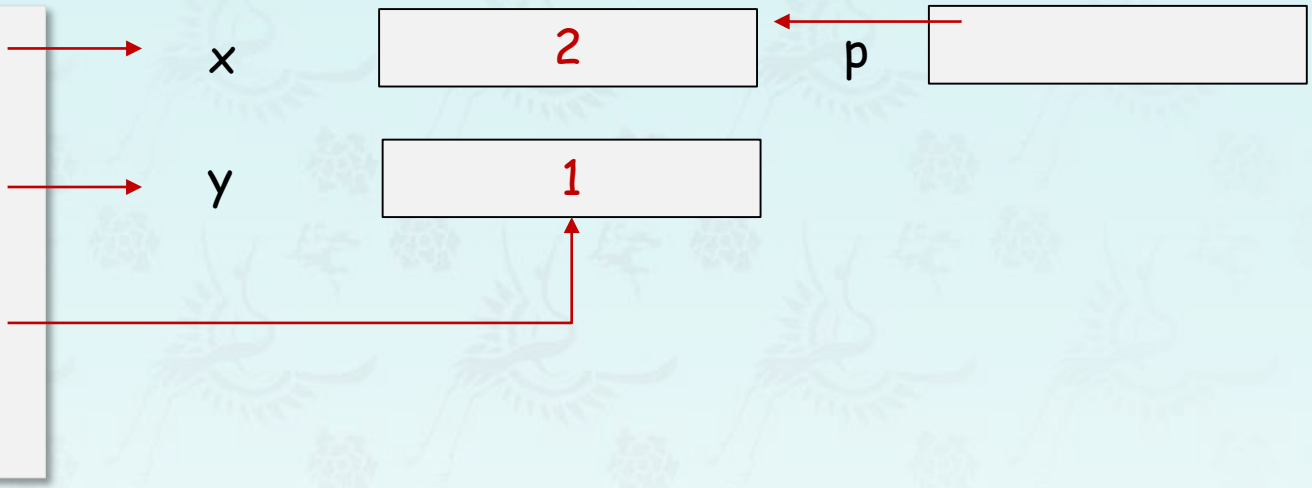
Tracing the Example

```
int x, *p;

int y;

y = 1;
p = &x;
*p = y + 1;
```



Tracing the Example

```
int x, *p;

int y;

y = 1;
p = &x;
*p = y + 1;
```



Example:

What is y at the end?

```
int y, x, *p;

x = 20;

*p = 10;

y = x + *p;
```

Example

What is x and y at the end?

```
int x = 1, y = 2;

int *p1, *p2;          // declares two pointers to ints

p1 = &x;       // p1 contains the address of x

y = *p1;       // * dereferences p1,

p2 = p1;       // p2 points to the same thing as p1

*p2 = 4;       // x is now
```

Example

What is x and y at the end?

```
int x = 1, y = 2;

int *p1, *p2;       // declares two pointers to ints

p1 = &x;      // p1 contains the address of x

y = *p1;      // * dereferences p1, so y = 1

p2 = p1;      // p2 points to the same thing as p1

*p2 = 4;      // x is now 4
```

Example:

What is y at the end?

```
int y, x, *p;

x = 20;

*p = 10;

y = x + *p;
```

Example:

What is y at the end?

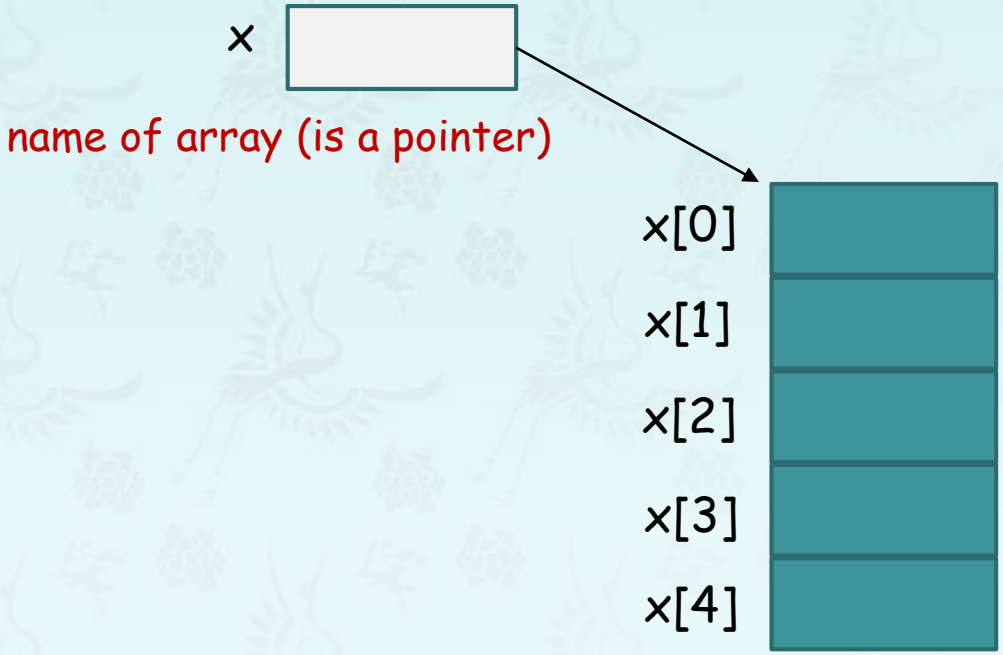
```
int y, x, *p;

x = 20;

*p = 10;

y = x + *p;
```

Oops!!
Dereferencing an uninitialized pointer will likely segfault or overwrite something!

Segfault = unauthorized memory access

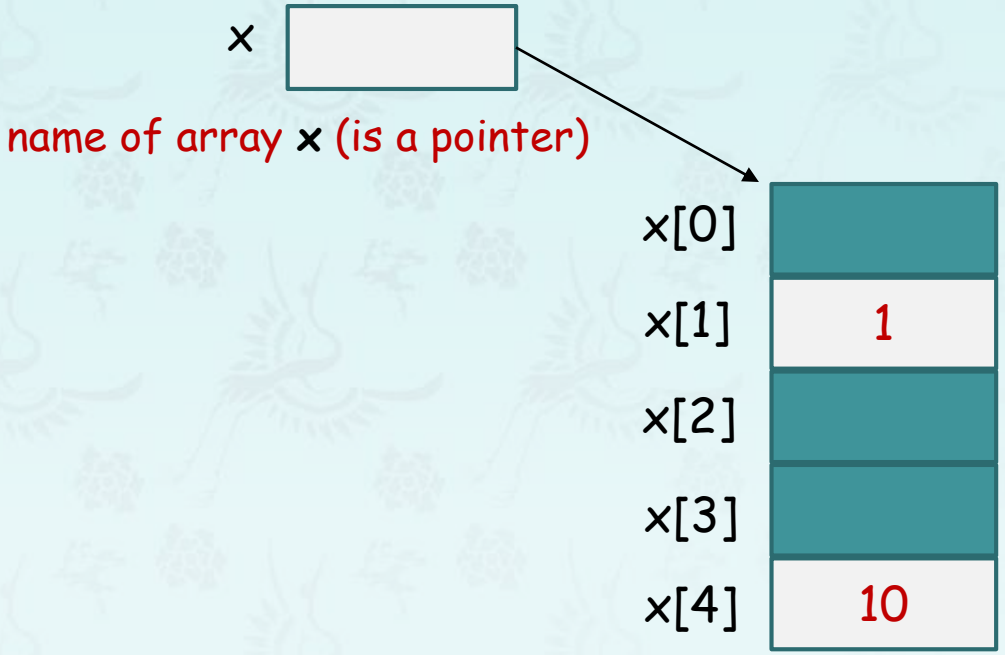
Arrays

int x[5];



Arrays

```
int x[5];
```




```
x[1] = 1;
x[4] = 10;
```

Array Name as Pointer

What's the difference between two examples below?

Example 1:

```
int x[10];
int *y;
y = x;
```

Example 2:

```
int x[10];
int *y;
y = &x[0];
```

Array Name as Pointer

What's the difference between two examples below?

Example 1:

```
int x[10];
int *y;
y = x;
```

Example 2:

```
int x[10];
int *y;
y = &x[0];
```

NOTHING!!

x (the array name) is a pointer to the beginning of the array, which is &x[0];

Question:

What's the difference between

```
int* array;
int arr[5];
```

What's wrong with:

```
int arr[5];
arr[1] = 1;
arr[2] = 2;
....
arr[5] = 5;
```

Question:

What is the value of a[3] at the end?

```
int a[4];
int *p;

a[0] = 4; a[1] = 3; a[2] = 10;
p = a;
*(p+2) = 20;
a[3] = a[1] + a[2];
```

Question:

What's the difference between the examples below

Example 1

```
int arr[5];
arr[3] = 6;
```

Example 2

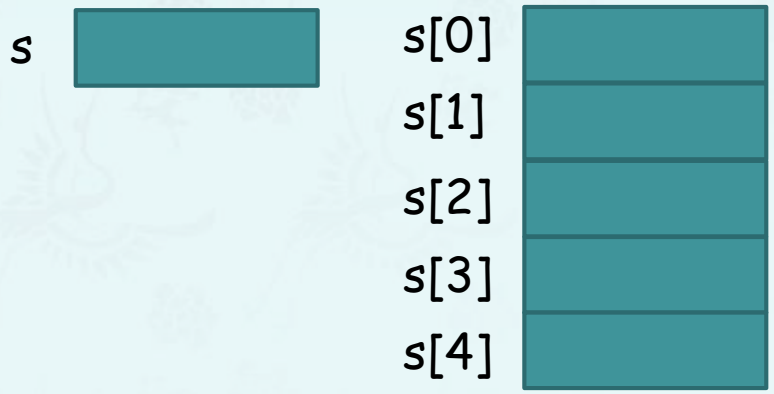
```
int arr[5];
*(arr + 3) = 6;
```

Strings
(Null-terminated Arrays of Char)

String is an array that contains characters followed by a "Null" character to indicate end of string.

- Do not forget to leave room for the null character.

Example: char s[5];

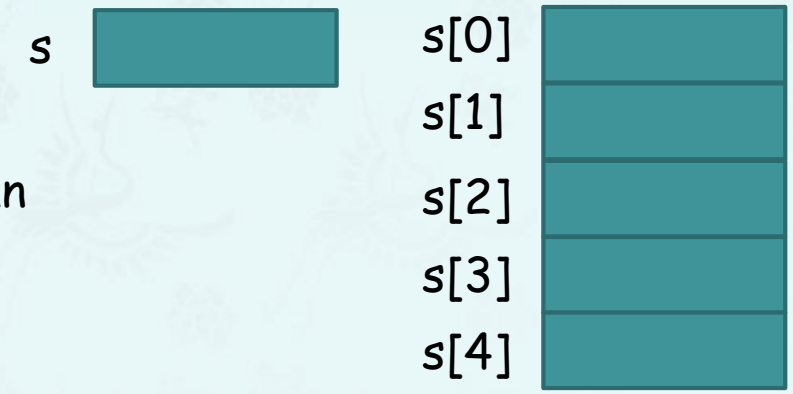


Strings
(Null-terminated Arrays of Char)

String is an array that contains characters followed by a "Null" character to indicate end of string.

- Do not forget to leave room for the null character.

Example: char s[5];



Question : How many characters can the string below hold?

Conventions

Strings
- "string"
- "c"

Character
- 'c'

String Operations

- strcpy
- strlen
- strcat
- strcmp

strcpy, strlen

Syntax:

strcpy(ptr1, ptr2);

where ptr1 and ptr2 are pointers to char

value = strlen(ptr);

where value is an integer and
ptr is a pointer to char

Example:

```
int len;
char str[15];
strcpy (str, "Hello, World!");
len = strlen(str);
```

Memory Management

Variables can be static, local, or malloc'ed

- **Static variables** live in special section of program, only 1 copy
- **Local variables** allocated automatically when a function is called, deallocated automatically when it returns

Memory Management

Variables can be static, local, or **malloc'ed**

- Dynamic storage is managed through the function malloc() and free()
- malloc returns a pointer to a chunk of memory in the heap
- Use when we don't know how big an array needs to be, or
  we need a variable that doesn't disappear when a function returns.

Memory Management

```
int main() {
    int x = 5;          // x is on the stack

    // y is a pointer to a chunk of memory
    // big enough to hold one int
    int *y = (int *) malloc (sizeof(int));

    // double is a pointer to a chunk of memory
    // big enough to hold 10 doubles
    double *z = (double*) malloc (10 * sizeof(double) );
    assert(z != NULL);   // something went wrong…

    // we can access the memory z points to
    // as though z was an array
    z[5] = 1.1;
}
```

Memory Management

- What happens to memory given out by malloc when we're done with it?
- Use function free() to free memory. free() takes a pointer given out by malloc, and frees the memory given out so it can be used again.
- Forgetting to call free is a cause of a significant percentage of memory leaks…

Memory Management

```
// arrays made without malloc are freed automatically
void ok() {
    int arr[10];
    return ;
}

/* arr is never freed; since function returned,
   we lost the only pointer we had to the memory
   we malloc'ed! */
void leaky() {
    int *arr = (int *) malloc (10 * sizeof(int) );
    return;
}
```

strcpy, strlen

What's wrong with

```
char str[5];
strcpy(str, "Hello");
```

strncpy

Syntax:
strncpy(ptr1, ptr2, num);
where ptr1 and ptr2 are pointers to char
num is the number of characters to be copied.

Example:

```
int len
char str1[15], str2[15];
strcpy(str1, "Hello, World!");
strncpy(str2, str1, 5);
```

strncpy

Syntax:
strncpy(ptr1, ptr2, num);
where ptr1 and ptr2 are pointers to char
num is the number of characters to be copied.

Example:

```
int len
char str1[15], str2[15];
strcpy(str1, "Hello, World!");
strncpy(str2, str1, 5);
```

Caution: strncpy blindly copies the characters. It does not voluntarily append the string-terminating null character.

strcat

Syntax: strcat(ptr1, ptr2);

where ptr1 and ptr2 are pointers to char

Concatenates the two null terminates strings yielding one string(pointed to by ptr1).

```
char S[25]="world!";
char D[25]="Hello, ";
strcat(D, S);
```

strcat

What's wrong with:

```
char S[25]="world!";
strcat("Hello, ", S);
```

strcmp

- Syntax: diff = strcmp(ptr1, ptr2);
  where diff is an integer and ptr1 and ptr2 are pointers to char.
- Returns zero if strings are identical

```
int diff;
char s1[25] = "pat";
char s2[25] = "pet";
diff = strcmp(s1, s2);
```

Structures

```
struct employee {
        char name[10];
        int salary;
        int year, month, day;
}

struct employee john;
struct employee *peter;

john.salary = 100;

peter = (employee *) malloc (sizeof(employee));
peter->salary = 200;
```

Structures

```
struct employee {
        char name[10];
        int salary;
        int year, month, day;
}

struct employee john;
struct employee *peter;

john.salary = 100;

peter = (employee *) malloc (sizeof(employee));
        salary = 200;
```

Structures

```
struct employee {
        char name[10];
        int salary;
        int year, month, day;
}

struct employee john;
struct employee *peter;

john.salary = 100;

peter = (employee *) malloc (sizeof(employee));
(*peter).salary = 200;
```

Structures

```
struct employee {
        char name[10];
        int salary;
        int year, month, day;
}

struct employee john;
struct employee *peter;

john.salary = 100;

peter = (employee *) malloc (sizeof(employee));
peter->salary = 200;
```

- With pointers to a structure, use → the **dereference operator** to access members**, exclusively.**

Structures

Functions can return structures

```
point_t makePoint(int x, int y) {
    point_t p;
    p.x = x;
    p.y = y;
    return p;
}
```

```
struct point_t
{
    int x, y;
}
```

User-defined types inside a struct

```
struct rect {
    point_t ll;   // lower left
    point_t ur;  // upper right
}
```

Type Casting

Re-interpret a parameter as a different type

```
int age, months;
float exactAge;

age = 11;
months = 3;
exactAge = (float) age;
exactAge = exactAge + ((float)months)/12;
```

Type Casting

Does this example work?

```
int    months
int    *age;
float  *exactAge;

age = malloc (sizeof(int));
exactAge = malloc(sizeof(float));

*age = 11;
months = 3;

exactAge = [redacted]age;
*exactAge = [redacted]+ ((float) months) / 12;
```

Type Casting

Does this example work?

```
int    months
int    *age;
float  *exactAge;

age = malloc (sizeof(int));
exactAge = malloc(sizeof(float));

*age = 11;
months = 3;

exactAge = (float *) age;
*exactAge = *exactAge + ((float) months) / 12;
```

Welcome to NowIC Jump Start

Part 0 – Why C?
Part 1 – Data
Part 2 – Algorithmic Construct
Part 3 – Preprocess, Compile, Link

Youngsup Kim, idebtor@gmail.com, Handong Global University

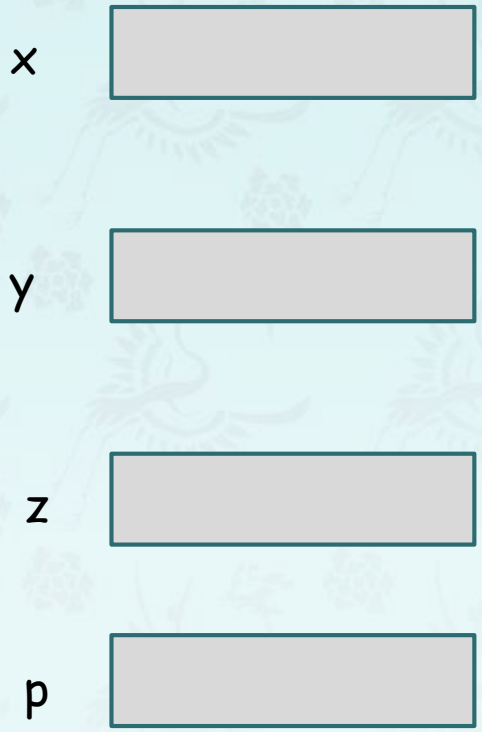
Algorithmic Constructs

- Assignment
- Input/Output
- if
- for
- while
- switch

not discussed in class

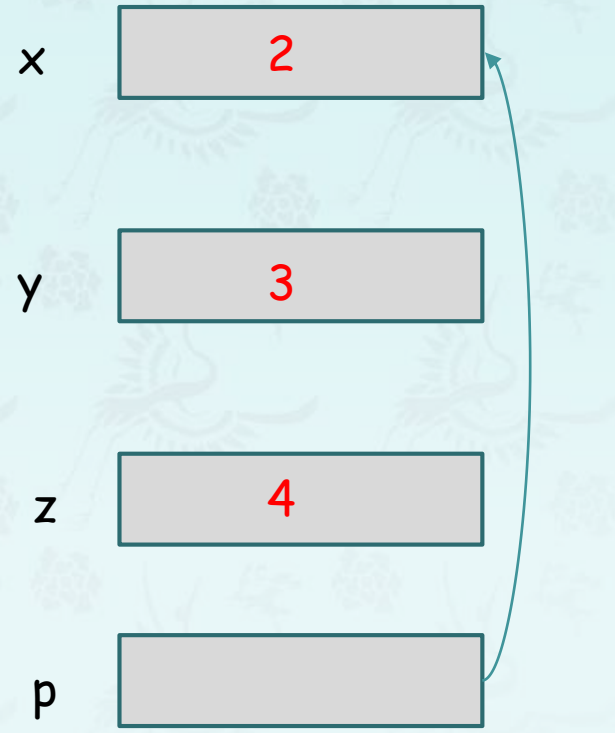
Assignment

```
int x, y;
Int *p;

x = 1;
y = 3;

z = x + y;
p = &x;
*p = 2;
```



Assignment

```
int x, y;
int* p;

x = 1;
y = 3;

z = x + y;
p = &x;
*p = 2;
```



Inc and Dec Operators

Example 1:

```
int x, y, z, w;
y = 10; w = 2;
x = ++y;
z = --w;
```

X = ?
Z = ?

Example 2:

```
int x, y;
y = 10; w=2;
x = y++;
z = w--;
```

X = ?
Z = ?

Inc and Dec Operators

Example 1:

```
int x, y, z, w;
y = 10; w = 2;
x = ++y;
z = --w;
```

First increment/ decrement then assign result x is 11, z is 1

Example 2:

```
int x, y;
y = 10; w=2;
x = y++;
z = w--;
```

First assign then increment/ decrement x is 10, z is 2

Inc/Dec Operators on Pointers

Example 1:

```
int a[3];
int number1, number2, *p;

a[0] = 1; a[1] = 10; a[2] = 100;

p = a;
number1 = *p++;
number2 = *p;
```

What will number1 and number2 be at the end?

Inc/Dec Operators on Pointers

Example 1:

```
int a[3];
int number1, number2, *p;

a[0] = 1; a[1] = 10; a[2] = 100;

p = a;
number1 = *p++;
number2 = *p;
```

Hint:
++ increments pointer p, not variable *p

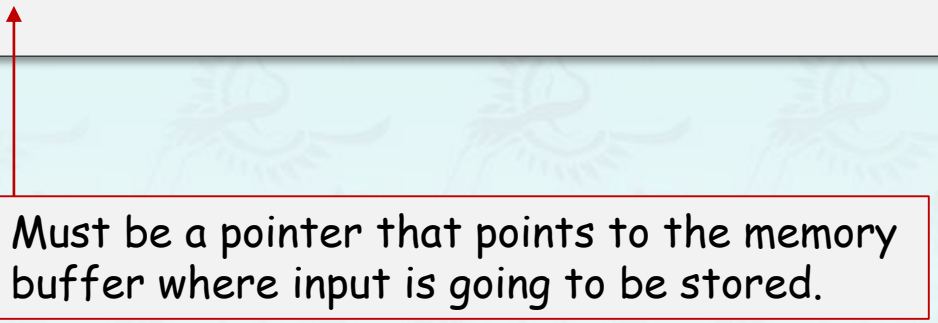
What will number1 and number2 be at the end?

Output

```
int age, weight;
float height;

age = 100;
weight = 300;
height = 5.5;

printf("Hi there! ");
printf("I am %d years old. I weigh %d lbs ", age, weight);
printf("I am %f ft tall. \n", height);
```

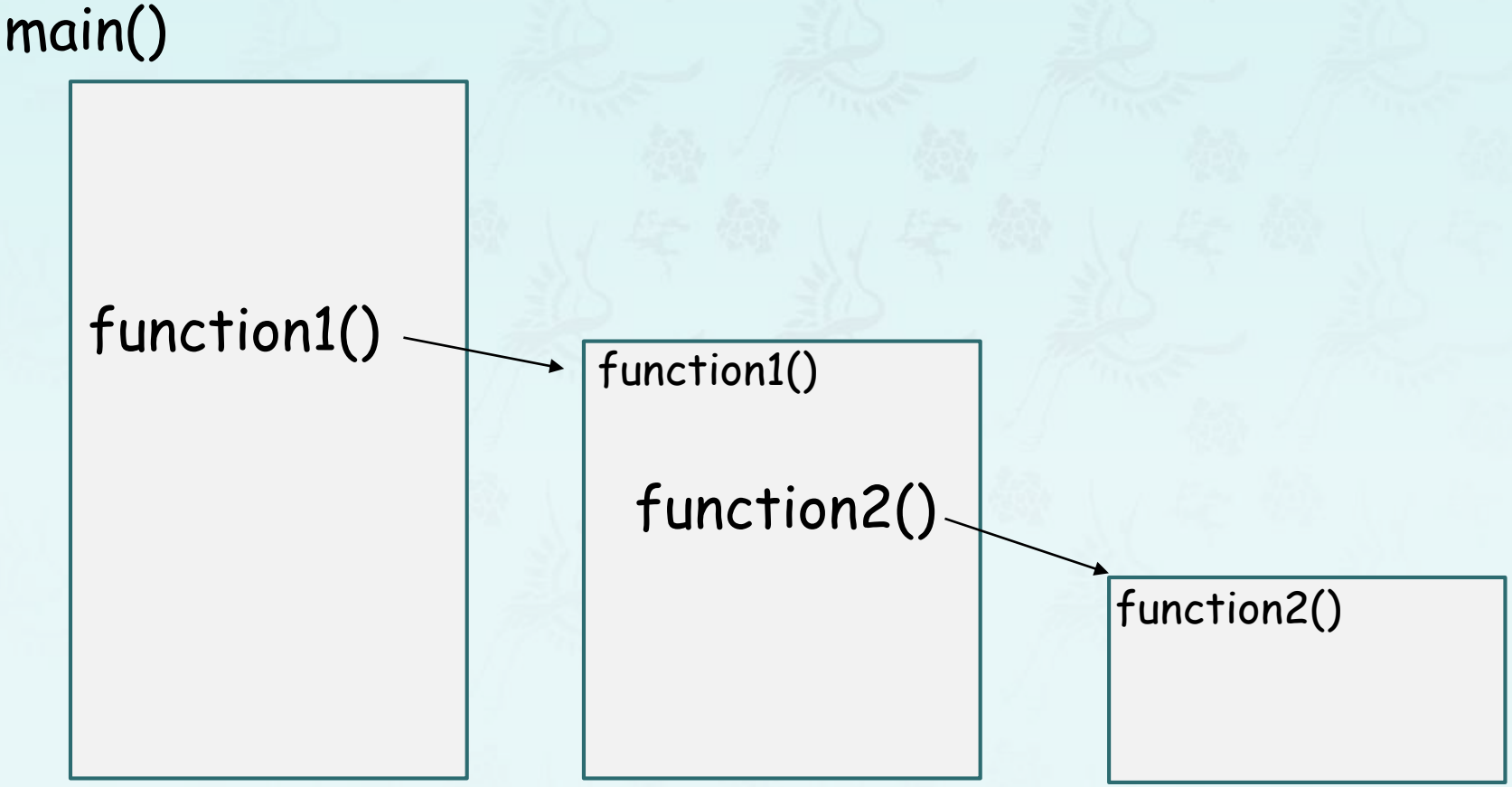
Input

```
int x;

printf(" Input your age here: ");

scanf("%d", &x);
```

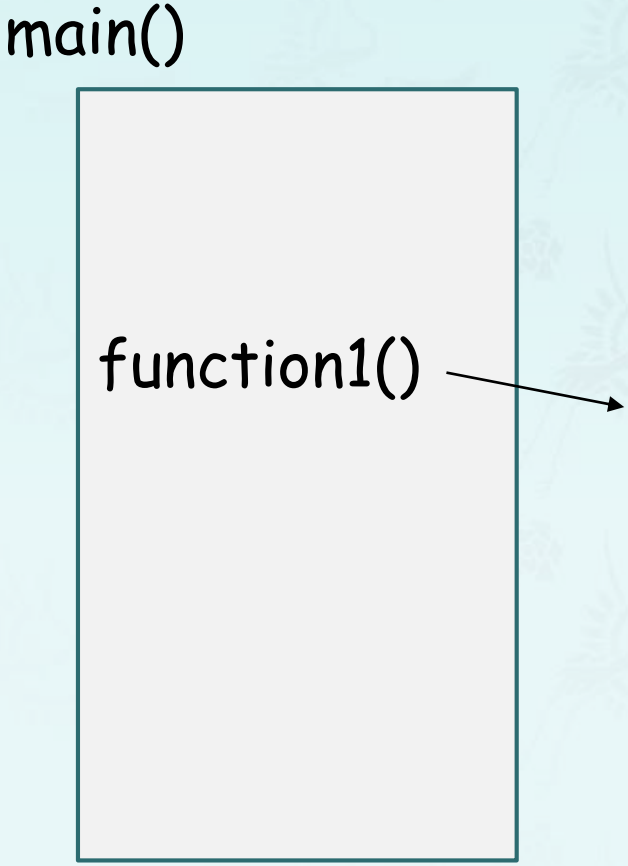


Functions, Scope and Stack

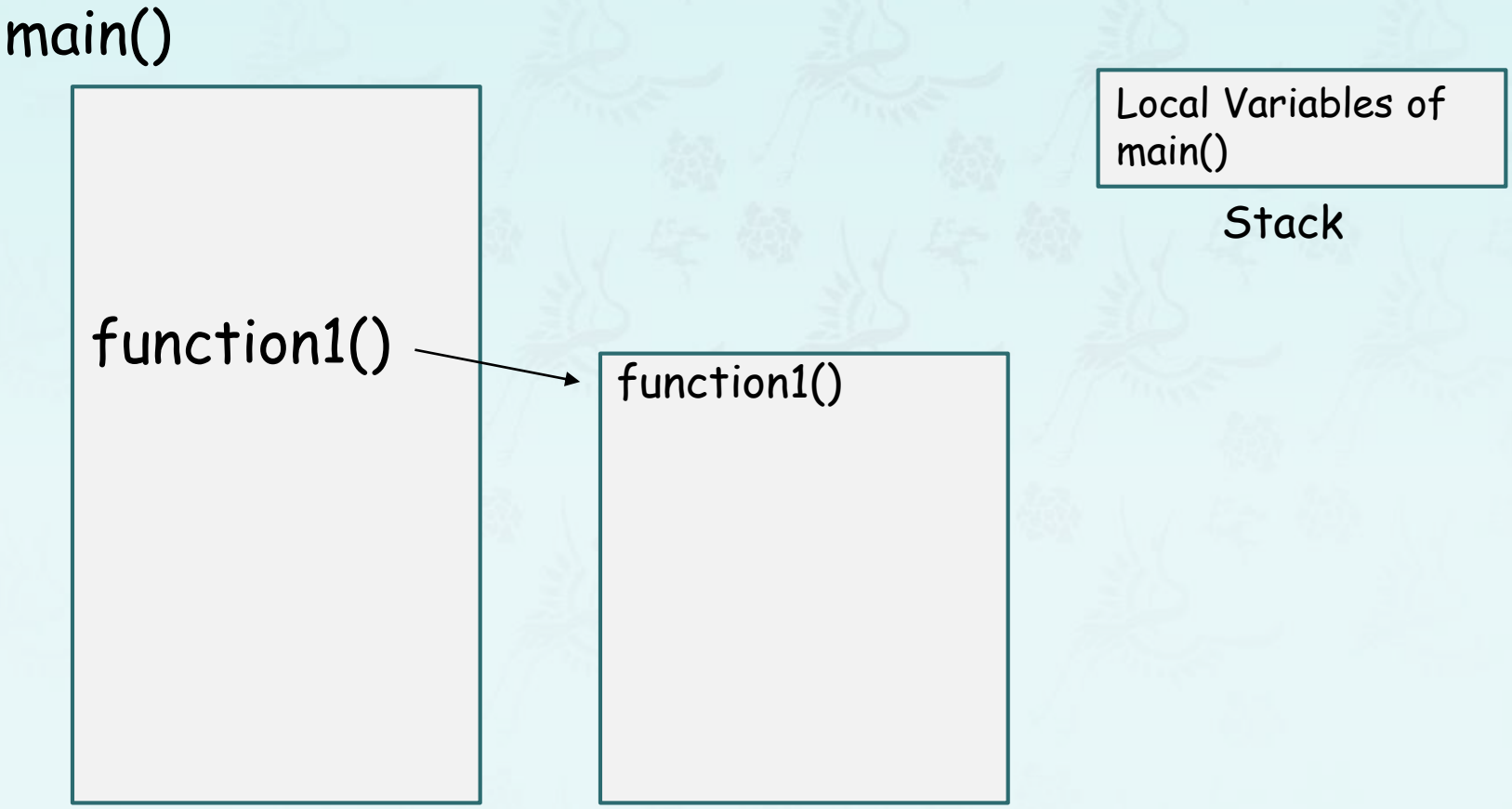
Functions and Stack



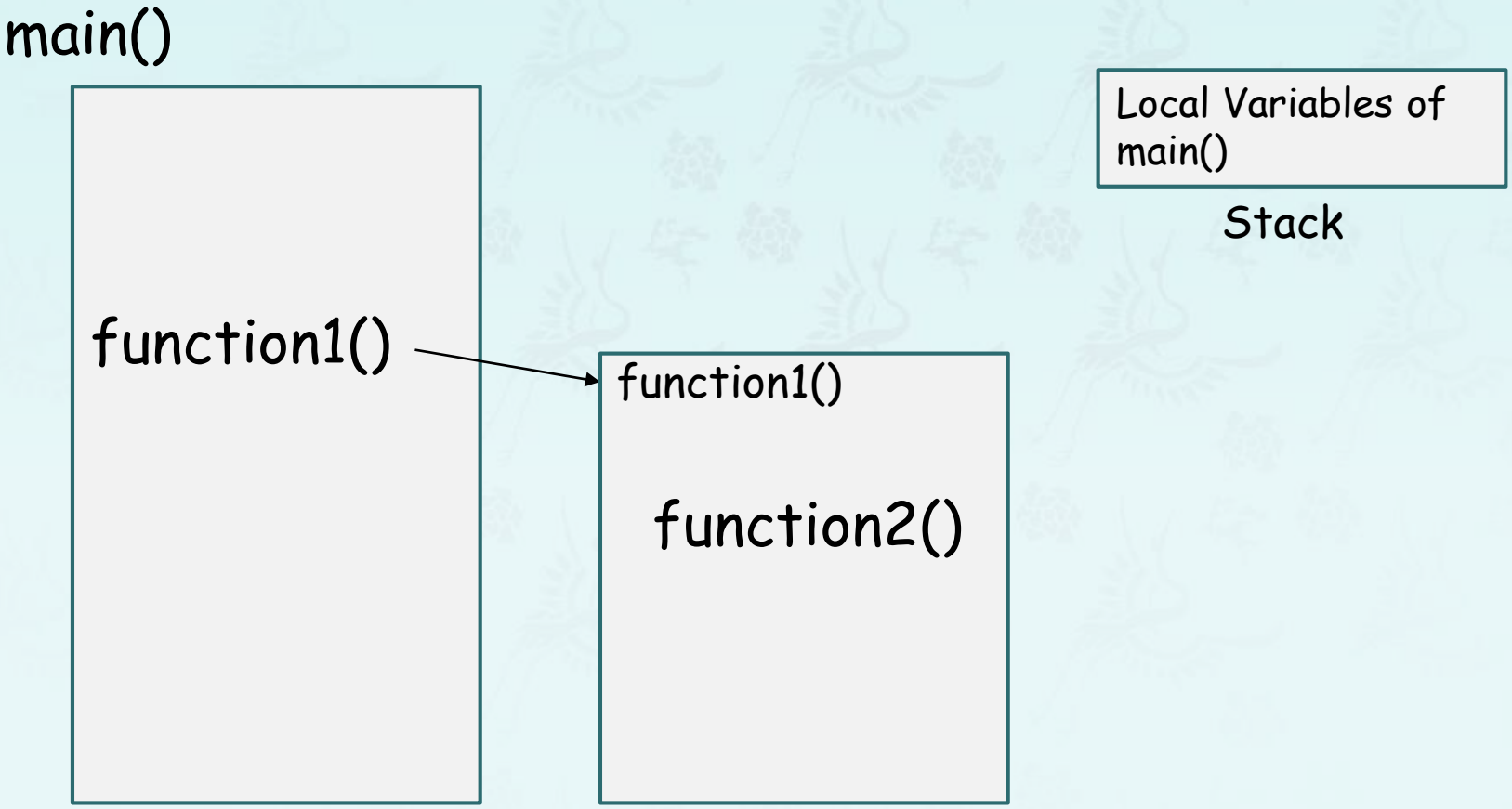
Functions and Stack



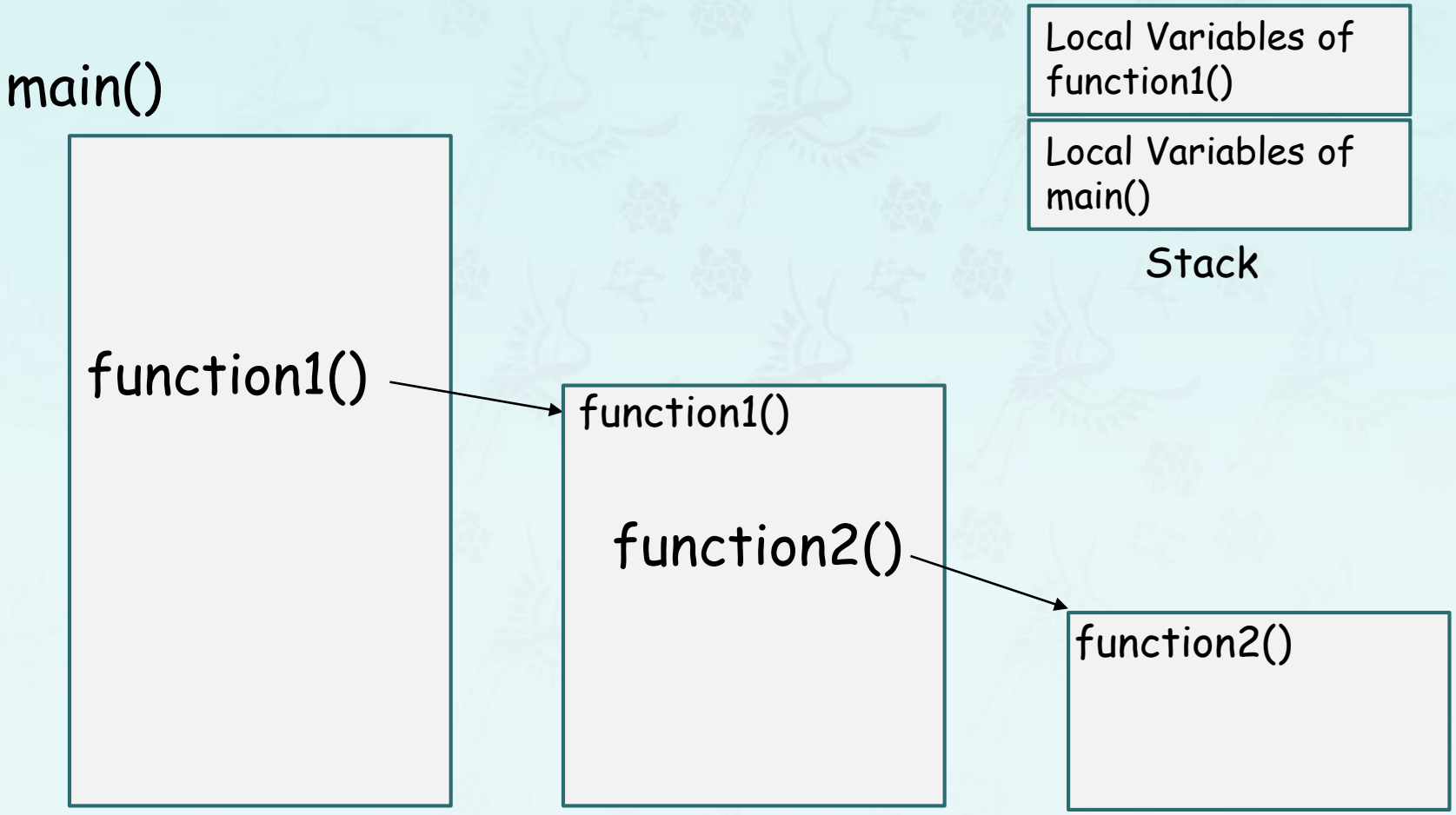
Functions and Stack



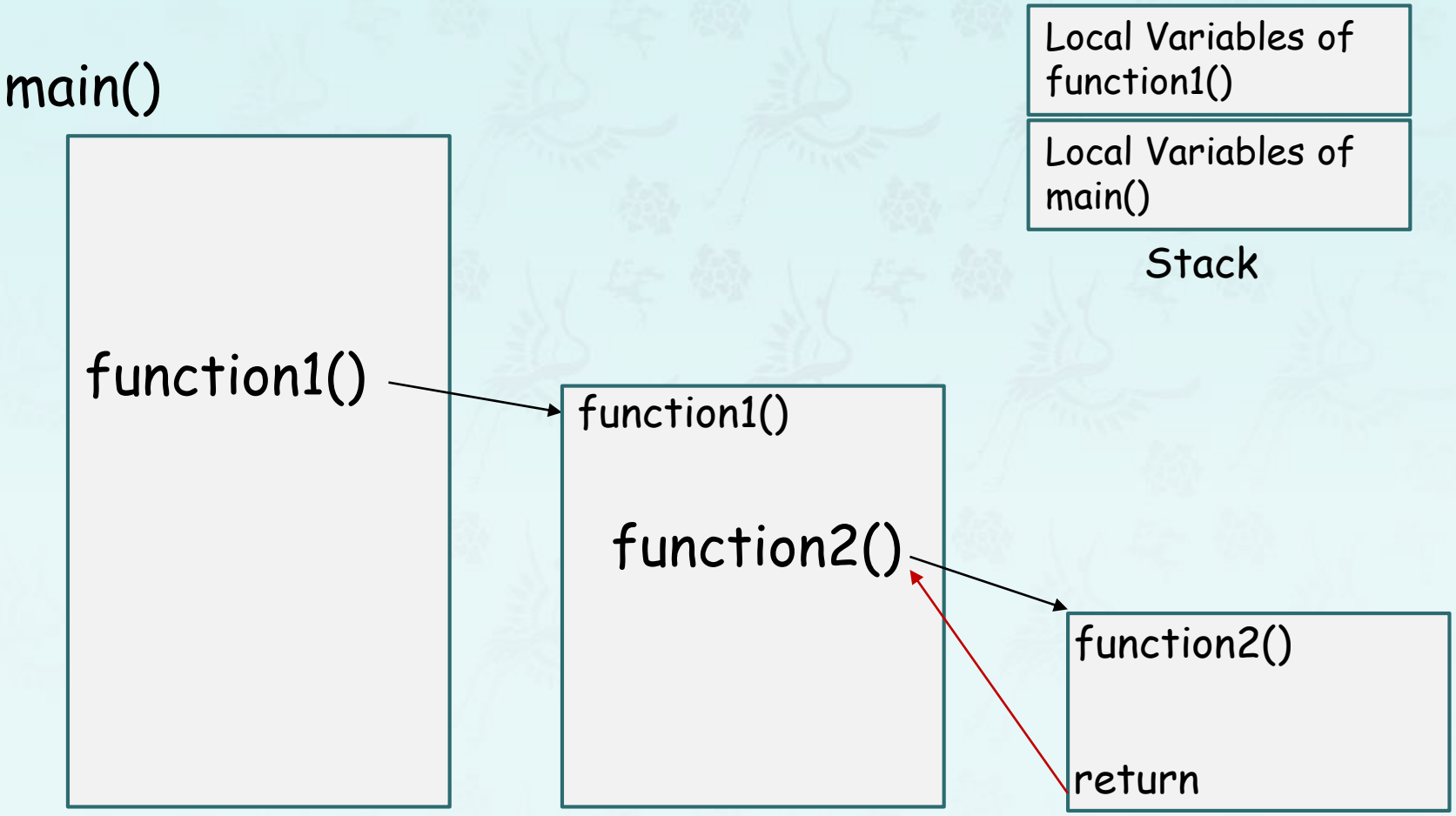
Functions and Stack



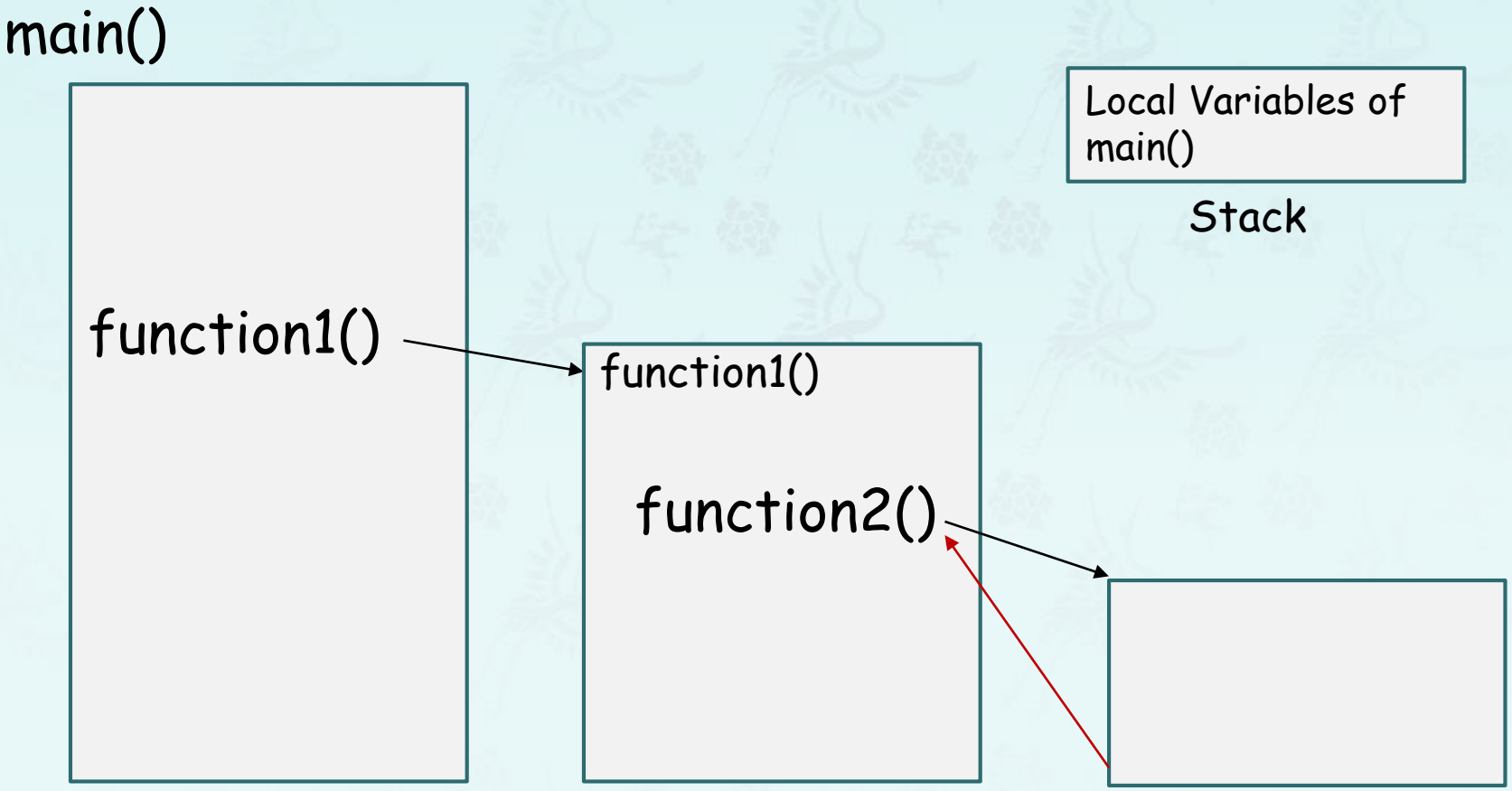
Functions and Stack



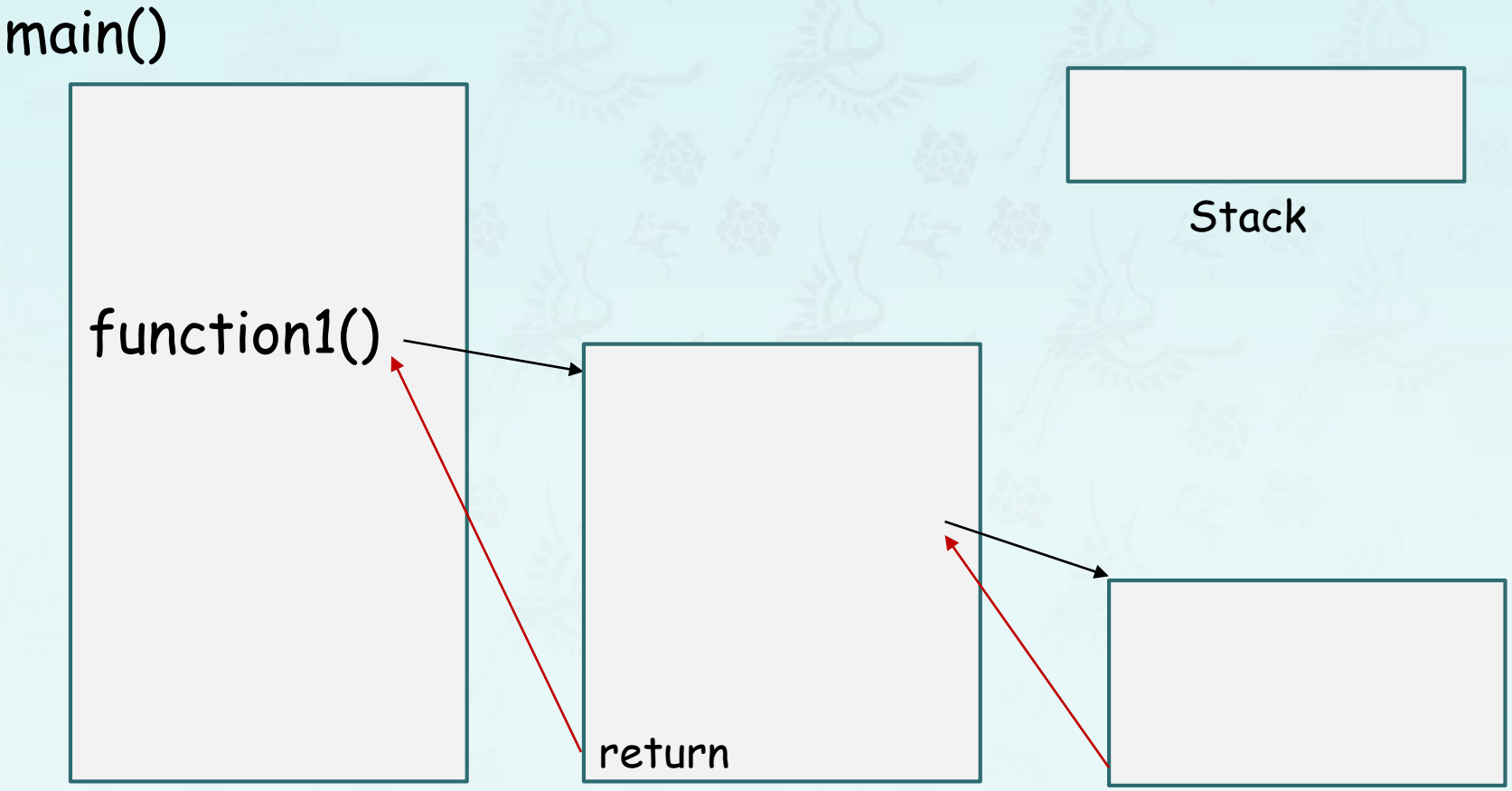
Functions and Stack



Functions and Stack



Functions and Stack



Function calls and Parameter Passing

```
int update (number) {
    int extra;
    extra = 8;
    number = number + extra;
    return (number);
}

main() {
    int x;
    x = 1;
    x = update(x);
    printf("Result = %d\n", x);
}
```

Note: This variable is local to the function. It is allocated on stack and will be removed when the function returns.

This statement tells the function what to return.

Function calls and Parameter Passing

What's wrong with:

```
void update (number) {
        int extra;
        extra = 8;
        number = number + extra;
}

main() {
    int x;
    x = 1;
    x = update(x);
    printf("Result = %d\n", x);
}
```

Function calls and Parameter Passing

What's wrong with:

```
void update (number) {
        int extra;
        extra = 8;
        number = number + extra;
}

main() {
    int x;
    x = 1;
    x = update(x);
    printf("Result = %d\n", x);
}
```

Note: This variable is local to the function. It is allocated on stack and will be removed when the function returns.

No "return". It is OK.
But, the function does not update x.

Pointers

```
void swap (int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int a = 1, b = 2;
swap(a, b);          // a and b did not get swapped
```

Pointers - Debugging

```
void swap (int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int a = 1, b = 2;
swap(a, b);          // a and b are now swapped
```

Pointers - Debugging

```
void swap (int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int a = 1, b = 2;
swap(&a, &b);       // a and b did get swapped
```

Special Example:

```
mint ain(int argc, char **argv)
{
...
}
```

Number of program arguments

An array of strings with:

argv[0] program name
argv[1] first argument
argv[2] second argument
etc.

char **argv is the same as
char *argv[]