



C Crash Course

Tarek Abdelzaher



The C Language Spirit

- Made by professional programmers for professional programmers
- 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" (see <http://www.darwinawards.com/>)
- UNIX and most "serious" system software (servers, compilers, etc) 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

```
main() {  
    int year;  
    year = 2007;  
    printf ("hello, world. This is %d\n", year);  
}
```



Programs

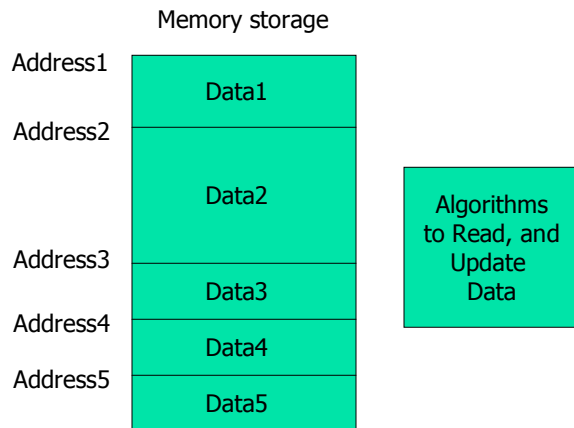
- Define
 - Data types and variables
 - Algorithms for manipulating those variables
- Example

```
main() {  
    int year;  
    year = 2007;  
    printf ("hello, world. This is %d\n", year);  
}
```

Declaration of variables:
typename varname;
←
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







PART I

Data











Basic Data Types

- Integers: `int`  (`double` )
- Characters: `char` 
- Floating point variables: `float` 
- Pointers (contain a memory address):
 - `int*`, `char*`, `float*`, `void*`, ...
 - Arrays
 - Strings



Basic Data Types

- Integers: `int`  (`double` )
- Characters: `char` 
- Floating point variables: `float` 
- 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;
```



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 a char, NOT pointer! Think: char *c, m;

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;      x ??      p ?? → #@#!&?  
int y;         y ??  
y=1;  
p = &x;  
*p = y+1;
```

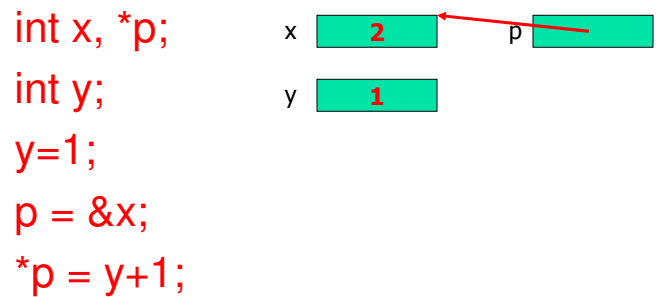
Tracing the Example

```
int x, *p;      x ??    p ?? → #@#!&?  
int y;         y 1  
y=1;  
p = &x;  
*p = y+1;
```

Tracing the Example

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

Tracing the Example



Allocating Memory to Hold Variables

```
pointerToBuffer = malloc (buffersize)
```

Example:

```
int x, *p;
...
p = (int*) malloc (sizeof (int))
*p = 10;
```



Example:

How much is y at the end:

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

```
x = 20;
```

```
*p = 10;
```

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



Example:

How much is y at the end:

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

```
x = 20;
```

```
*p = 10;
```

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

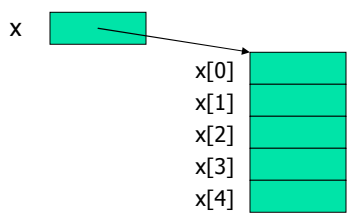
BAD!!
Dereferencing an uninitialized pointer
will likely segfault or overwrite
something!

**Segfault = unauthorized memory
access**

Arrays

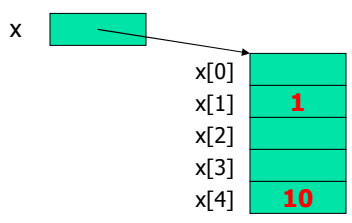
```
int x[5];
```

↑
Name of array (is a pointer)



Example

```
int x[5];  
x[1]=1;  
x[4]=10;
```





Array Name as Pointer

- What's the difference between the 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 the examples below:

Example 1:

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

NOTHING!!

Example 2:

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

**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's the difference between the examples below

Example 1

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

NOTHING!!

Example 2:

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

**Incrementing a pointer
increases the address it points
to by the given number of
elements (of the right type)**



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;
```

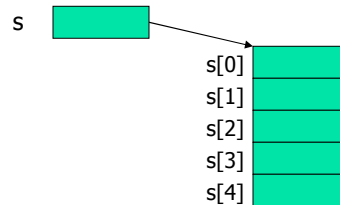
Strings

(Null-terminated Arrays of Char)

- Example

`char s[5];`

- Strings are arrays that contain the string characters followed by a "Null" character to indicate end of string.
 - Do not forget to leave room for the null character



Strings

(Null-terminated Arrays of Char)

- Example

`char s[5];`

- Strings are arrays that contain the string characters followed by a "Null" character to indicate end of string.
 - Do not forget to leave room for the null character
- Question: How many characters can the string below hold?

`char s[5];`



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);
```



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

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




strcat

- Syntax: `strcat(ptr1, ptr2);`
where `ptr1` and `ptr2` are pointers to `char`

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

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



strcat

Example

- 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 *empPointer;
```

```
empPointer = (employee*) malloc (sizeof(employee))  
john.salary = 100;  
empPointer->salary= 200;
```



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 *age, months;  
float *exactAge;
```

```
age = malloc (sizeof(int));  
exactAge = malloc (sizeof(float));  
*age = 11;  
months = 3;  
exactAge = (float*) age;  
*exactAge = *exactAge + ((float)months)/12;
```



PART II

Algorithms

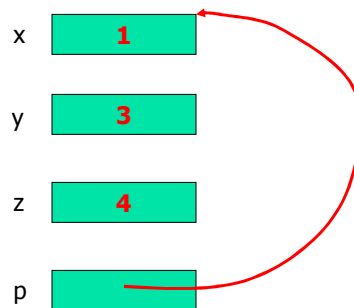
- 
- ### Main Algorithmic Constructs
- Assignment
 - Input/Output
 - if
 - for
 - while
 - switch

Assignment

```
int x, y;  
int *p;  
x=1;  
y=3;  
z=x+y;  
p= &x;  
*p= 2;
```

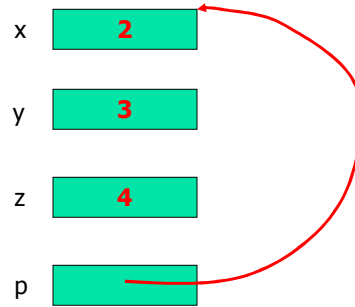
Assignment

```
int x, y;  
int *p;  
x=1;  
y=3;  
z=x+y;  
p= &x;  
*p= 2;
```



Assignment

```
int x, y;  
int p;  
x=1;  
y=3;  
z=x+y;  
p= &x;  
*p= 2;
```



Increment and Decrement Operators

Example 1:

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

Example 2:

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

Increment and Decrement 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

Increment and Decrement Operators on Pointers

Example 1:

```
int a[2];  
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?



Increment and Decrement Operators on Pointers

Example 1:

```
int a[2];  
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);
```

←
Must be a pointer that points to the memory buffer where input is going to be stored



if

```
if (condition) something;  
else somethingelse;
```

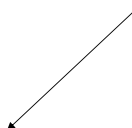
```
if (condition) {  
    something1;  
    something2;  
    ...  
}  
else {  
    somethingelse1;  
    somethingelse2;  
    ...  
}
```



if Example

```
int a
int b
...
if (a == b) printf ("Equal.");
else printf ("Not Equal.");
```

Equality operator



Relational (Condition) Operators

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to



Equality versus Assignment

- Note that `(a=b)` is an assignment, whereas `(a==b)` is a condition that evaluates to true when `a` and `b` are equal and to false otherwise.
- What's wrong with:
`if (a = b) printf ("Equal.");`
`else printf ("Not Equal.");`



Multiple Conditions

And: `((condition) && (condition))`
Or: `((condition) || (condition))`

Example:

```
int age;  
float price, paid;  
...  
if ((age >= 21) && (paid >= price))  
    beerPurchaseAllowed();
```



for

Example

```
int sum;  
sum=0;  
  
for (i=0; i<10; i++) {  
    sum=sum+i;  
}  
printf ("Sum = %d\n", sum);
```

Initialization

Continue while this condition holds

Do this after each iteration



while

```
While (condition) {  
    dothis;  
    dothat;  
}
```



while

Example

```
float budget, leftover, price;
budget=100;
leftover=budget;
while (leftover>0) {
    scanf ("Item price %f", &price);
    leftover = leftover - price;
}
```



switch

```
int countA, countB, countC;
char c;
...
switch(c) {
    case 'a': countA++; break;
    case 'b': countB++; break;
    case 'c': countC++; break;
    default: printf("Unknown character\n"); break;
}
```



switch

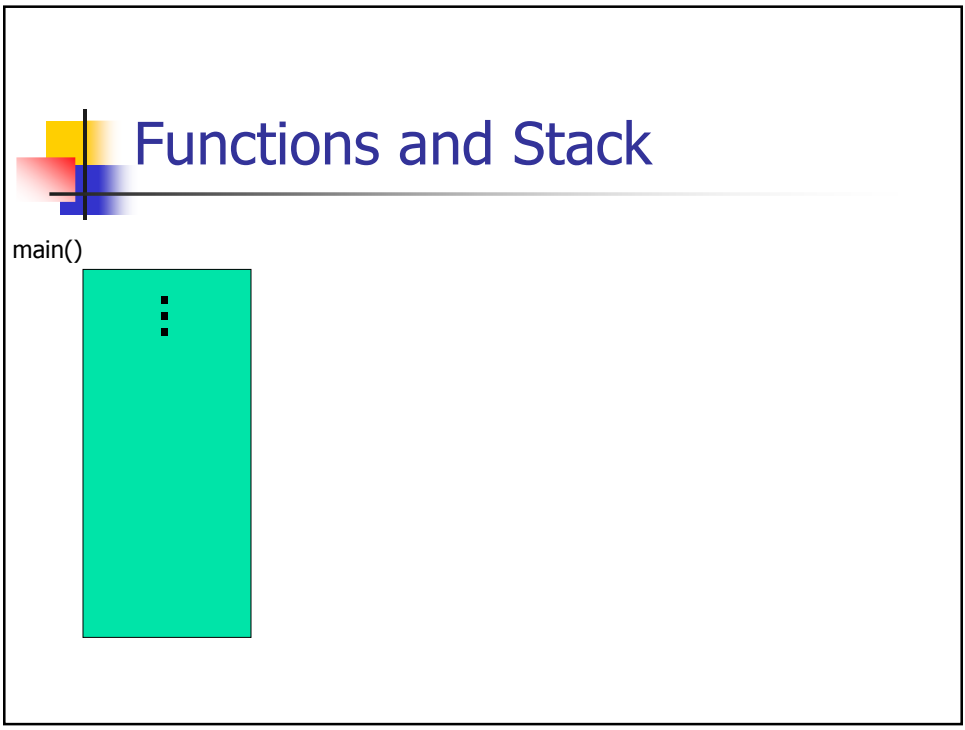
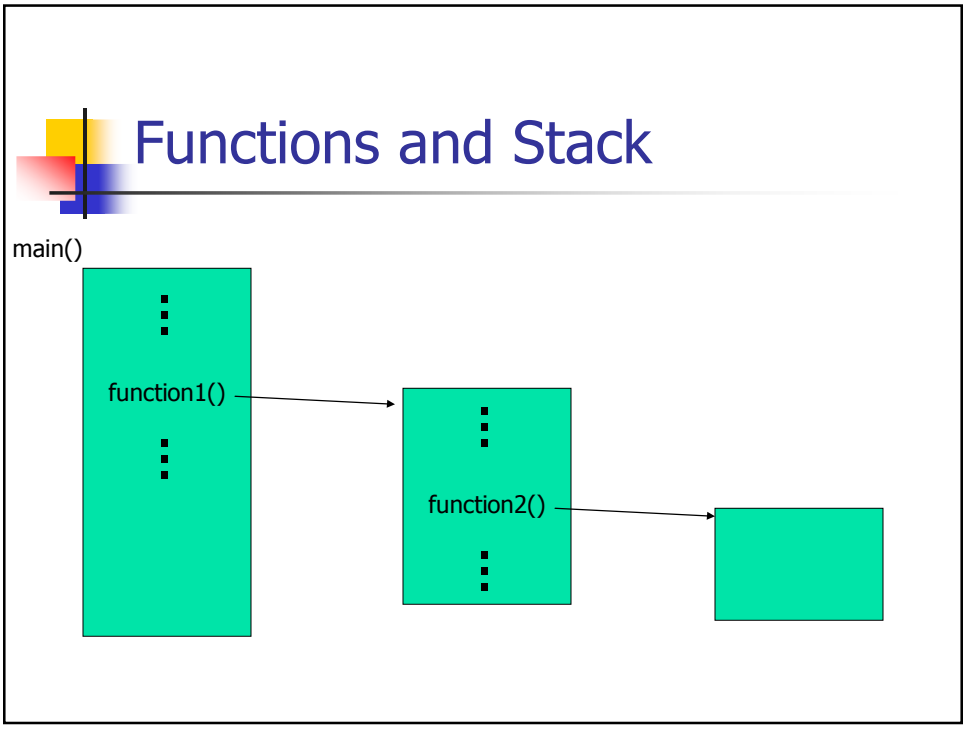
```
int countA, countB, countC;
char c;
...
switch(c) {
    case 'a': countA++; break;
    case 'b': countB++; break;
    case 'c': countC++; break;
    default: printf("Unknown character\n"); break;
}
```

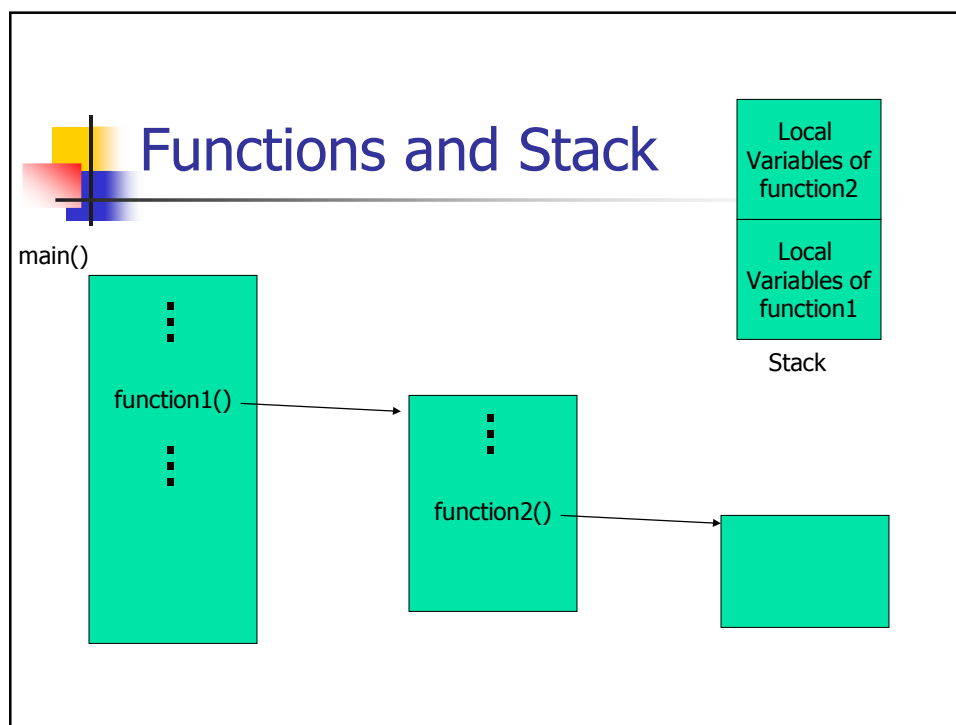
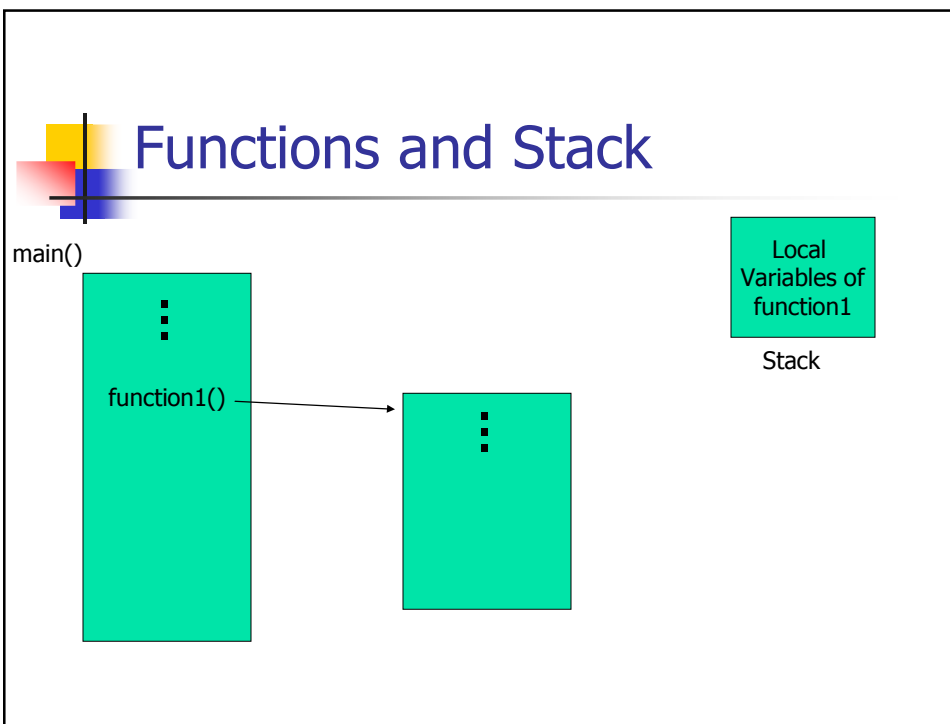
Must be something you can enumerate like an integer or character

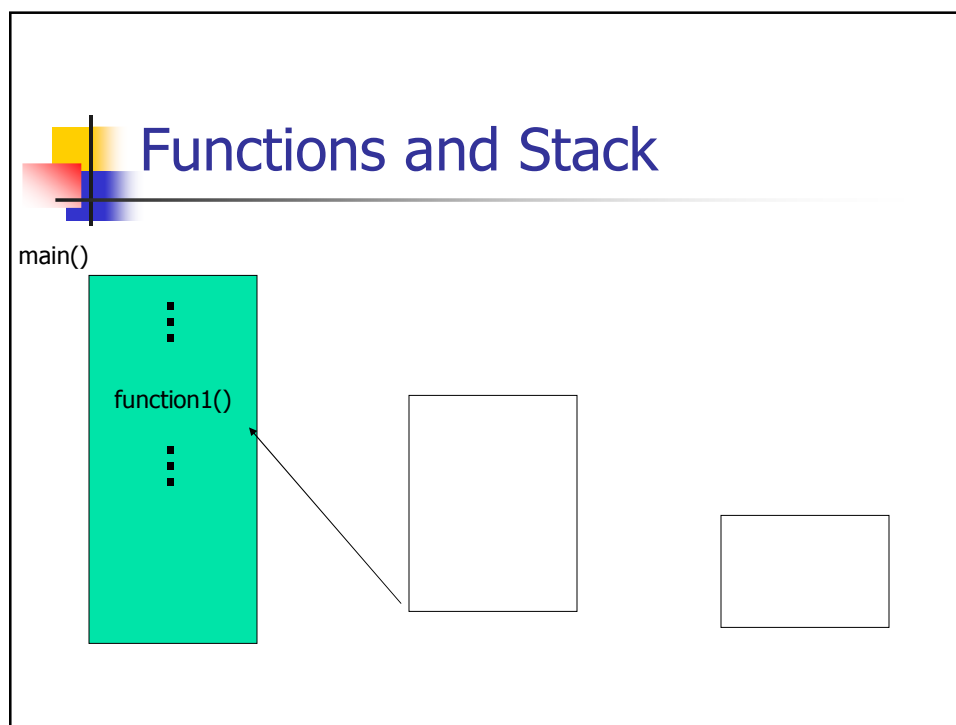
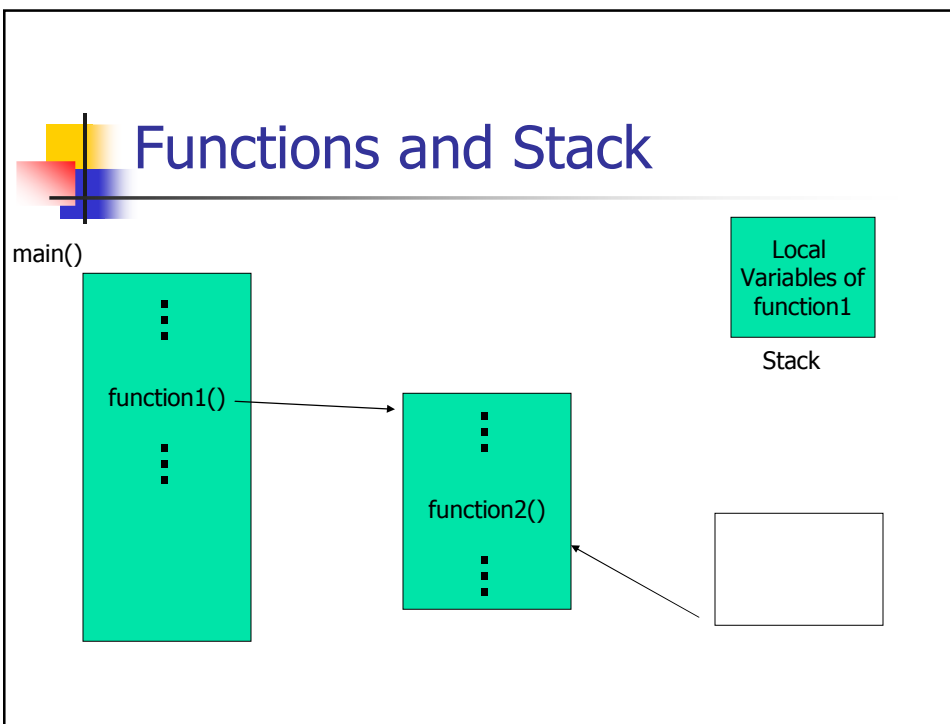
Exists statement. If you forget it, execution will fall through to next case.



Functions, Scope and Stack







Function Calls and Parameter Passing

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

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

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

This statement tells the function what to return

Example:

- What's wrong with:

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

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

```
main () {
  int x
  ...
  x = update (x);
}
```



Example:

- What's wrong with:

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

```
main () {
  int x
  ...
  x = update (x);
```

← No "return". The function does not update x



Example

What's the difference between:

Example 1

```
int update (this)
  int this; {
    this = this * 2;
  }
```

Example 2

```
int update (this)
  int* this {
    *this = (*this) * 2;
  }
```



Special Example:

Number of program arguments

`main(argc, argv)`

`int argc;`

`char **argv;`

{

...

}

An array of strings with:

`argv[0]` program name

`argv[1]` first argument

`argv[2]` second argument

etc