Functions

ESC101: Fundamentals of Computing

Nisheeth

Announcements

- Hope you enjoyed the exam
- Upcoming exams
  - Lab mid sem exam on 15th Feb 1000 - 1600
  - Theory mid sem exam on 21st Feb 1800-2000
- Lessons from MQ1
  - Be on time
  - Bring ID
  - Write roll numbers on every answer sheet

We have seen functions before

- main() is a special function. Execution of program starts from the beginning of main()
- scanf(…), printf(…) are standard input-output library functions
- sqrt(…), pow(…) are math functions in math.h

Writing our own functions..

A standard program for max of two numbers

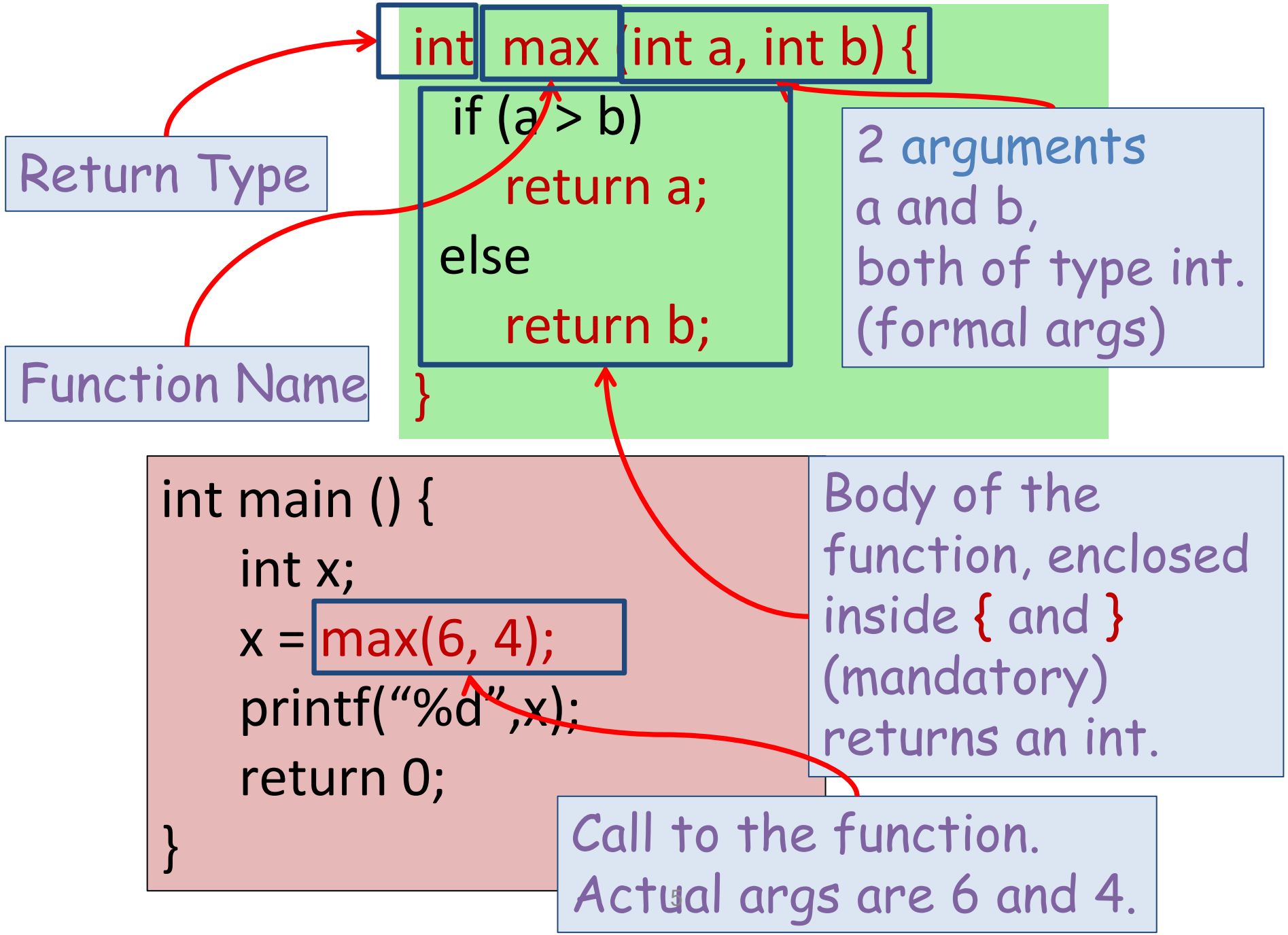
```
int main () {
    int x;
    int a,b;
    scanf("%d%d",&a,&b);
    if(a>b)
       x = a;
    else
       x = b;
    printf("%d",x);
    return 0;
}
```

```
int  max (int a, int b) {
  if (a > b)
      return a;
  else
      return b;
}
```

We or someone else may have already written this "max" function and tested well (so very little chance of error)

```
int main () {
    int x;
    int a,b;
    scanf("%d%d",&a,&b);
    x = max(a, b);
    printf("%d",x);
    return 0;
}
```

A program with our own function


int max (int a, int b) {
if (a > b)
return a;
else
return b;
}
Return Type
Function Name
2 arguments
a and b,
both of type int.
(formal args)
int main () {
int x;
x = max(6, 4);
printf("%d",x);
return 0;
}
Body of the
function, enclosed
inside { and }
(mandatory)
returns an int.
Call to the function.
Actual args are 6 and 4.

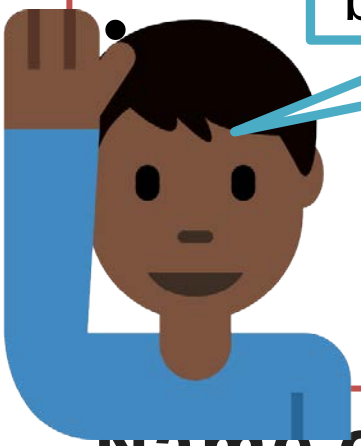
The Anatomy of a C Function

How we must speak to the compiler

- int is

How we usually speak to a human

- isUpperAlpha is a function that takes in a character (let us call that character x) as input and gives an output

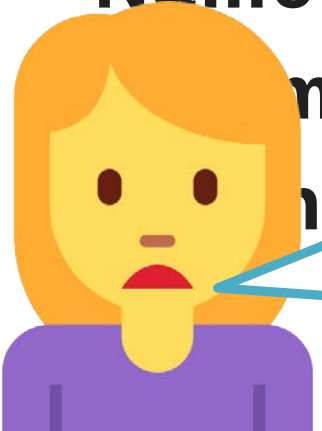


Yes you can! But you have to be a bit clever about doing so

We will teach you 3 ways to return more than one output in this course

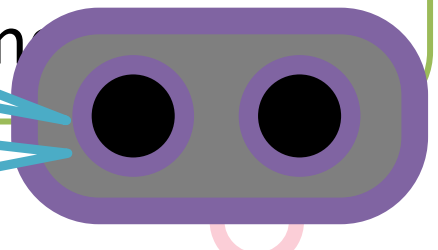
Programmers often call the process of giving inputs to a function as *passing arguments to the function*

Inputs to a function are called its *arguments*
A function *returns* its output



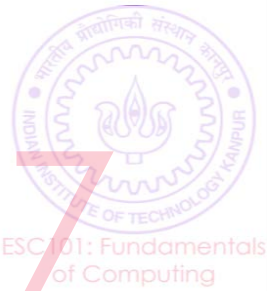
So I cant write a function that returns 2 integers – say x and y coordinates?

A function may have many inputs but only one output



Why use functions?

- Break up complex problem into small sub-problems.
- Solve each of the sub-problems separately as a function, and combine them together in another function.
- The main tool in C for modular programming.



Functions help us write compact code

Example : Maximum of 3 numbers

```
int main(){
   int a, b, c, m;

   /* code to read
    * a, b, c */

   if (a>b){
     if (a>c) m = a;
     else m = c;
   }
   else{
     if (b>c) m = b;
     else m = c;
   }

   /* print or use m */

   return 0;
}
```

```
int max(int a, int b){
   if (a>b)
     return a;
   else
     return b;
}

int main() {
   int a, b, c, m;

   /* code to read
    * a, b, c */

   m = max(a, b);
   m = max(m, c);
   /* print or use m */

   return 0;
}
```

This code can scale easily to handle large number of inputs (e.g.: max of 100 numbers!)

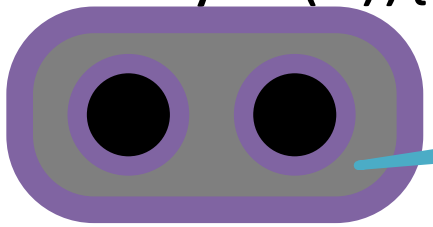
Other benefits of writing functions

- **Code Reuse**: Allows us to reuse a piece of code as many times as we want, without having to write it.
  - Think of the `printf` function!
- **Procedural Abstraction**: Different pieces of your algorithm can be implemented using different functions.
- **Distribution of Tasks**: A large project can be broken into components and distributed to multiple people.
- **Easier to debug**: If your task is divided into smaller subtasks, it is easier to find errors.
- **Easier to understand**: Code is better organized and hence easier for an outsider to understand it.

Other benefits of writing functions

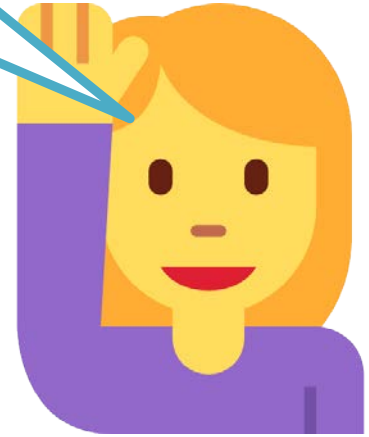
- **Allows you to think very clearly**
- E.g. if you want to do somet… number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    …
}
```

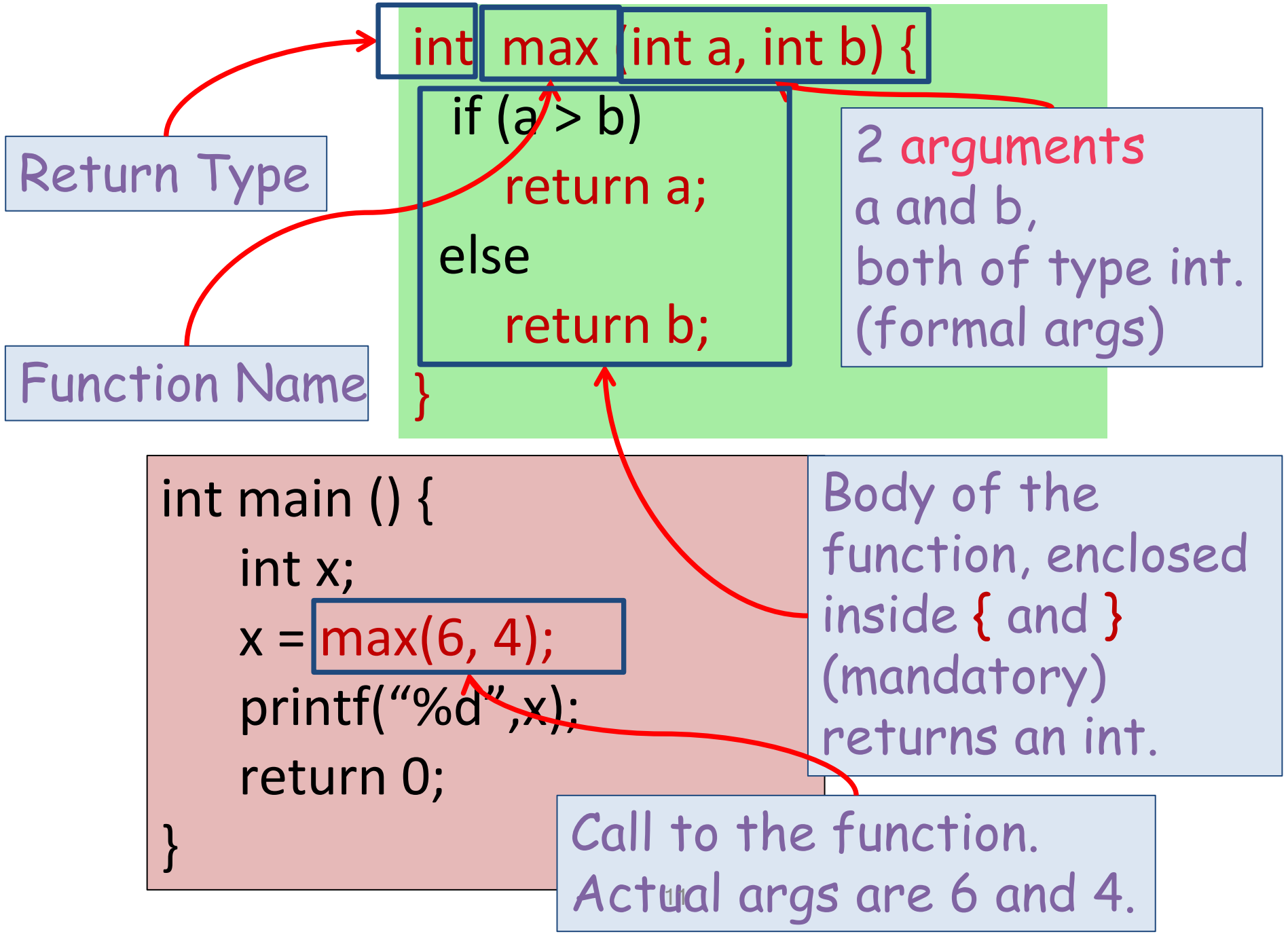


E.g. in this case, primality testing is one module, checking for divisibility by 11 is another module

Writing code that has modules is a type of *modular programming* – it is the the industry standard!



- Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺
- You can break your code into chunks – called modules
- Each module handled using a separate function


int max (int a, int b) {
if (a > b)
return a;
else
return b;
}
Return Type
Function Name
2 arguments
a and b,
both of type int.
(formal args)
int main () {
int x;
x = max(6, 4);
printf("%d",x);
return 0;
}
Body of the
function, enclosed
inside { and }
(mandatory)
returns an int.
Call to the function.
Actual args are 6 and 4.

Function Terminology

Same rule as variable names

Function Name: must be a valid identifier abc, a124, _ab1. Ideally, should reflect what the function does

Arguments: can be int, long, float, double, char

Can also have pointers and even arrays as input – soon!

Return type: what does the function *return*

When you use a function, we say you have *called* that function. If the function outputs something, we say the function *returned* that output back to you

Function Practice Exercises

Define a function to input two integers, output their max

Define a function to print Hello World

Define a function to output 1 if input is prime else 0

Define a function to input two integers and print Hello World if their max is prime

Define a function to print the max of 3 numbers

Define a function to input a character, output its upper case version if lower case else output the character itself

Function Declaration?

We declare variables before using them. For example

int x; x = 2;

Do we have to declare functions before using them?

Not necessary. Optional in modern C

Also known as function "prototype"

If you do, here is what a function's declaration looks like

```
return_type function_name (comma_separated_list_of_args);
```

```
int max(int a, int b);
int max(int x, int y);
int max(int , int);
```

All 3 declarations are equivalent. Variable names don't matter, and are optional. Note the semi-colon

Header files usually contains function declarations

Position of declaration must be before the first call to the function in the code, and also not inside any function

Prototype

Function call

Definition

```c
#include <stdio.h>

int checkPrimeNumber(int n);

int main() {
        int n1, n2, i, flag;
        printf("Enter two positive integers: ");
        scanf("%d %d", &n1, &n2);
        printf("Prime numbers between %d and %d are: ", n1, n2);

        for(i=n1+1; i<n2; ++i) {
        // i is a prime number, flag will be equal to 1
                flag = checkPrimeNumber(i);
                if(flag == 1)
                        printf("%d ",i);
        }
        return 0;
}

// user-defined function to check prime number
int checkPrimeNumber(int n) {
        int j, flag = 1;
        for(j=2; j <= n/2; ++j) {
                if (n%j == 0) {
                        flag =0;
                        break;
                }
        }
        return flag;
}
```

"Position" of a Function

If not declared already, the called function must be defined before where it is called. Can define it below the calling function only if the called function's return type is int (else compiler assumes int return type and will complain if it finds some other return type)

```
int  max (int a, int b) {
   if (a > b)
       return a;
   else
       return b;
}
```

```
int main () {
      int x;
      x = max(6, 4);
      printf("%d",x);
      return 0;
}
```

```
int main () {
      int x;
      x = max(6, 4);
      printf("%d",x);
      return 0;
}
```

```
int  max (int a, int b) {
   if (a > b)
       return a;
   else
       return b;
}
```

Arguments and Return types

You can define a function that takes in no input and gives no output

Even void print(){ … } works

```
void print(void){
    printf("Hello World");
}
```

You can define a function that takes inputs but gives no output

```
void sum(int a, int b){
    printf("Sum %d", a+b);
}
```

You can define a function that takes no input but gives an output

Even char getFirstAlpha(){ … } works

```
char getFirstAlpha(void){
    return 'A';
}
```

More on Arguments

Argument name can be any valid variable name

Another type of scope rule for variables. Not "block" based but function based

Can reuse a variable name even if this name used in main or another function

Calling a function is like creating a clone of Mr C. This clone starts afresh, with any inputs you have given. The clone forgets all old variable names and values

Will see more about this "cloning" behaviour later

scope of m1, a1, b1

```
int max(int a, int b) {
    int m = 0;
    if (a > b) m = a;
    else m = b;
    return m;
}
```

scope of m2, a2, b2

```
int min(int a, int b) {
    int m = 0;
    if (a < b) m = a;
    else m = b;
    return m;
}
```

```
int main() { ... }
```

More on Arguments

If you have promised to give a function two integers, please give it two integers

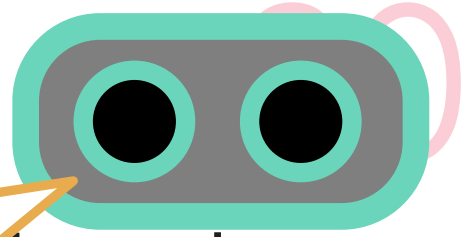
If you give it only one or three integers, compilation error

If you give it two floats or else one char and one int, automatic typecasting will take place

Be careful to not make typecasting errors

More o

For functions that do not need to return anything i.e. void return type, you can either say return; or else not write return at all inside the function body in which case the entire body will get executed



May write re function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

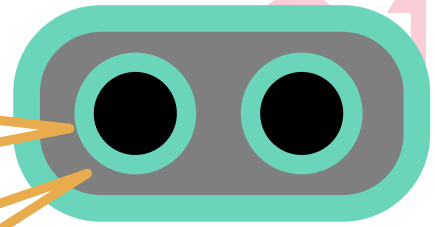
The clone dies and the original Mr C takes over ☺

If you return a float/double value from a function with int return type, automatic typecasting will take place.

Be careful to not make typecasting mistakes

More on Return

main() is also a function with return type int



main() is like a reserved function name. Cannot name your function main

The value that is re[illegible]ast as a normal variable of [illegible]

You can freely use returned values in expressions

- Be careful of type though

```
int sum(int x, int y){
    return x + y;
}
```

```
int main(){
    printf("%d", sum(3,4) - sum(5,6));
    return 0;
}
```

Can even use within printf

Function and Expression

A function call is an *expression*. Can be used anywhere an expression can be used subject to type restrictions

Example below: assume we have already written the max and min functions for two integer arguments

printf("%d", max(5,3));	prints 5
max(5,3) – min(5,3)	evaluates to 2
max(x, max(y, z)) == z	checks if z is max of x, y, z
if (max(a, b)) printf("Y");	prints Y if max of a and b is not 0.

Nested Function Calls

Not just main function but other functions can also call each other

A declaration or definition (or both) must be visible before the call

Help compiler detect any inconsistencies in function use

Compiler warning, if both (decl & def) are missing

```
#include<stdio.h>
int min(int, int); //declaration
int max(int, int); //of max, min

int max(int a, int b) {
  return (a > b) ? a : b;
}

// this "cryptic" min, uses max :-)
int min(int a, int b) {
  return a + b – max (a, b);
}

int main() {
 printf("%d", min(6, 4));
}
```

Benefits of writing functions

Functions allow you to reuse code

Some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again

printf() and scanf() are also functions. Think of how much we use them in every single program

We are reusing code that some helpful C expert wrote in the printf(), scanf(), sqrt(), abs() and other functions

If some piece of code keeps getting used in your program again and again – put it inside a function!

We reused code in today's codes – didn't have to rewrite code – may make mistakes if you write same code again