An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 7

Functions

Call, and they shall return!

Based on material developed by Prof. Abhiram G. Ranade

Functions

- We have already seen a few mathematical functions:
  like `sqrt(x)`, `cosine(x)`, etc.
- Functions can also carry out tasks (rather than map inputs to outputs):
  like `forward(s)`, `left(d)`, etc.
- These functions are not built into C++, but implemented in some libraries (like simplecpp, or libraries included in simplecpp)
- Today: **How to implement your own functions**
  - Syntax and semantics of functions
  - Examples

Example: Prime-Factors Equivalence

- Let us say two numbers are *prime-factors equivalent (PFE)* if they have the same set of prime factors (ignoring multiplicities). A program to check PFE:

```
int a, b;
cin >> a >> b;
bool a_covers_b; // a_covers_b if every prime factor of b divides a.
bool b_covers_a; // similarly, b_covers_a

// TODO: code to evaluate a_covers_b
// TODO: code to evaluate b_covers_a

cout  << (a_covers_b && b_covers_a) ? "Equivalent!":"Not equivalent”
      << endl;
```

Example: Prime-Factors Equivalence

```
bool a_covers_b; // a_covers_b if every prime factor of b divides a.

// code to evaluate a_covers_b
a_covers_b = true;
for (int d=2; abs(b) > 1; (b%d==0) ? b/=d : d++) {
  if (b%d == 0 && a%d !=0) {
    a_covers_b = false;
    break;
  }
}
```

Example: Prime-Factors Equivalence

```
int a, b;
cin >> a >> b;
// code to evaluate a_covers_b
bool a_covers_b = true;
for (int d=2; abs(b) > 1; (b%d==0) ? b/=d : d++) {
  if (b%d == 0 && a%d != 0) {
    a_covers_b = false;
    break;
  }
}
bool b_covers_a;
// TODO: evaluate b_covers_a. Repeat the above snippet with a and b exchanged

cout  << (a_covers_b && b_covers_a) ? "PFE!":"Not PFE") << endl;
```

This alters b. Will need to change it to work on a copy of b.

Problems with duplicating code:

- Will have to carefully exchange a and b everywhere.
- While fixing bugs, will have to remember to fix it in both places
- If later modifying this code, will have to update in both places

Example: Prime-Factors Equivalence

type of output ("return value")

inputs ("parameters")

```
bool covers(int w, int x) {
  for (int d=2; abs(x) > 1; (x%d==0) ? x/=d : d++) {
    if (x%d == 0 && w%d != 0) return false;
  }
  return true;
}
```

Syntax: `return` *exp*;

Semantics: Evaluate *exp*, "return" it to the caller, and terminate the function execution

An expression, that evaluates to the return value. The arguments are copied to the function as its parameters. Their original values are not affected by the function call.

```
main_program {
  int a, b;
  cin >> a >> b;
  bool a_covers_b = covers(a,b);
  bool b_covers_a = covers(b,a);
  cout  << (a_covers_b && b_covers_a) ?
             "Equivalent!":"Not equivalent") << endl;
}
```

Example: Prime-Factors Equivalence

```
bool covers(int w, int x) {
  for (int d=2; abs(x) > 1; (x%d==0) ? x/=d : d++) {
    if (x%d == 0 && w%d != 0) return false;
  }
  return true;
}
```

```
bool PFE(int a, int b) {
  return covers(a,b) && covers(b,a);
}
```

```
main_program {
  int a, b;
  cin >> a >> b;
  cout  << PFE(a,b) ? "Equivalent!":"Not equivalent”) << endl;
}
```

Example: Prime-Factors Equivalence

Demo

```
int gcd(int a, int b) {
  // TODO
}
```

```
//remove all factors of w from x
int reduce(int w, int x) { // here assuming x!=0
  for(int g = gcd(w,x); g>1; x/=g, g = gcd(w,x)) {}
  return x;
}
```

Reimplemented

```
bool covers(int w, int x) {
   return reduce(w,x)==1;
}
```

```
bool PFE(int a, int b) {
  return covers(a,b) && covers(b,a);
}
```

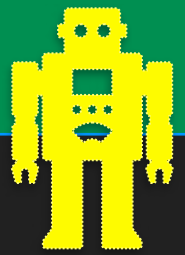
```
main_program {
  unsigned int a, b;
  cin >> a >> b;
  cout  << PFE(a,b) ? "Equivalent!":"Not equivalent") << endl;
}
```

No change!

How Function Calls Work: The Stack

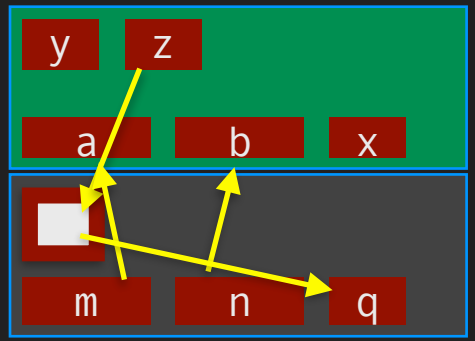
```
main_program {
  int m, n;
  bool q;
  //...
  q = PFE(m,n);
  //...
}
```

```
bool PFE(int a, int b) {
 bool x, y, z;
 // ...
 return z;
}
```



Processor

Memory



How Function Calls Work: The Stack

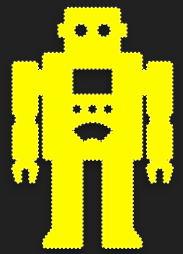
```
int gcd(int a, int b) {
  // ...
    return ans;
}
```

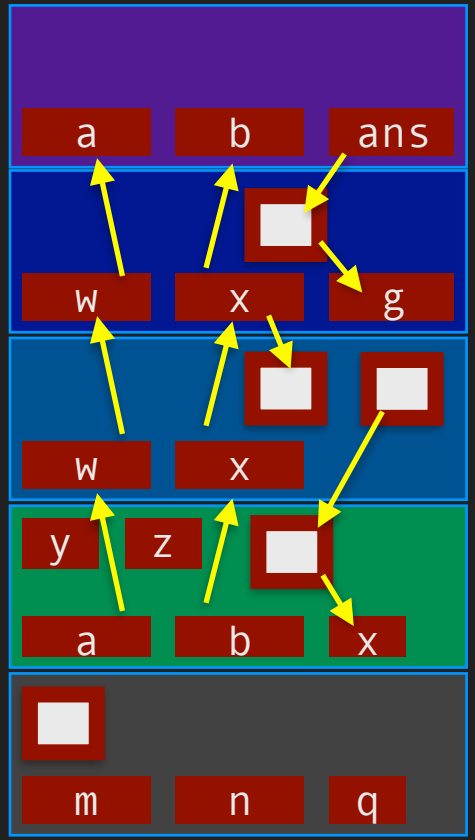
```
int reduce(int w, int x) {
  int g = gcd(w,x);
  // ...
  return x;
```

```
bool covers(int w, int x) {
  return reduce(w,x)==1;
}
```

```
bool PFE(int a, int b) {
  bool x, y, z;
  x = covers(a,b);
  y = covers(b,a);
  z = x && y;
  return z;
}
```

```
main_program {
  int m, n;
  bool q;
  //...
  q = PFE(m,n);
  //...
}
```





How Function Calls Work: The Stack

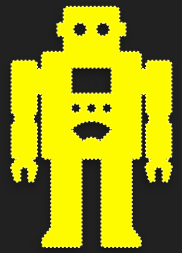
```
int gcd(int a, int b) {
  // ...
    return ans;
}
```

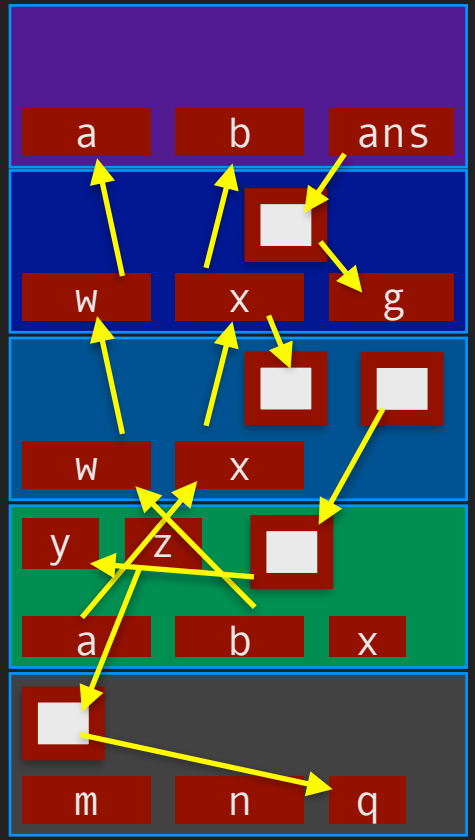
```
int reduce(int w, int x) {
  int g = gcd(w,x);
  // ...
  return x;
```

```
bool covers(int w, int x) {
  return reduce(w,x)==1;
}
```

```
bool PFE(int a, int b) {
 bool x, y, z;
 x = covers(a,b);
 y = covers(b,a);
 z = x && y;
 return z;
}
```

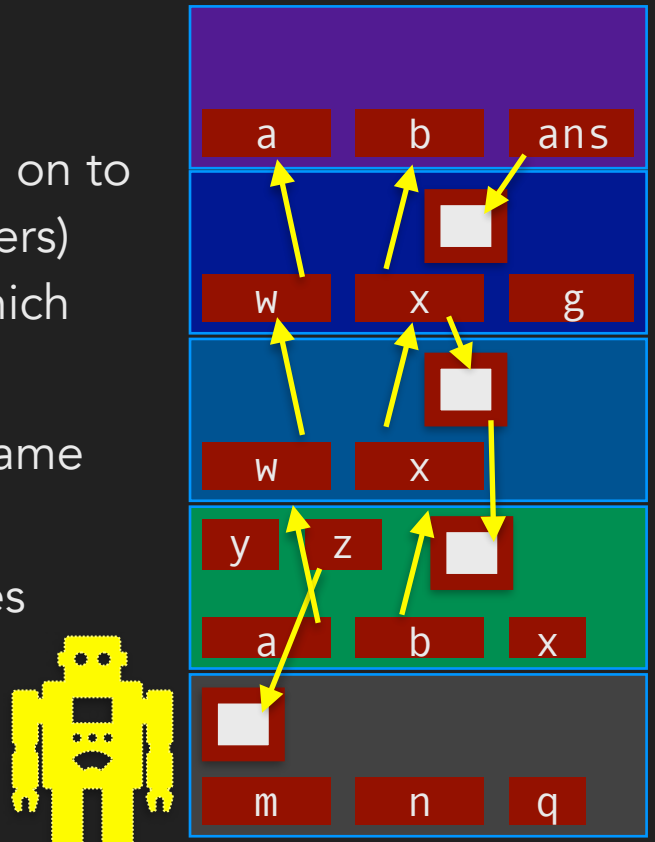
```
main_program {
  int m, n;
  bool q;
  //...
  q = PFE(m,n);
  //...
}
```





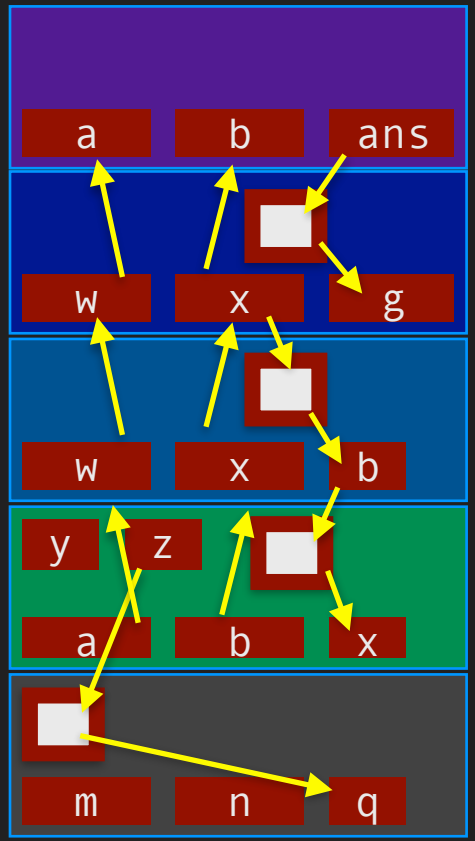
How Function Calls Work: The Stack

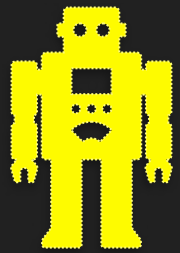
- When a function is called, it gets its own piece of memory at the top of the stack: its "frame"
- The inputs to the function (arguments) are copied on to the corresponding variables in its frame (parameters) from the frame below it (frame of the function which called it)
- While executing, the function uses only its own frame (but later: *global variables*, *references*, *pointers*)
  - Temporary expression values as well as variables
- When the function returns, the return value is copied into the frame below, and then its own frame is discarded



How Function Calls Work: The Stack

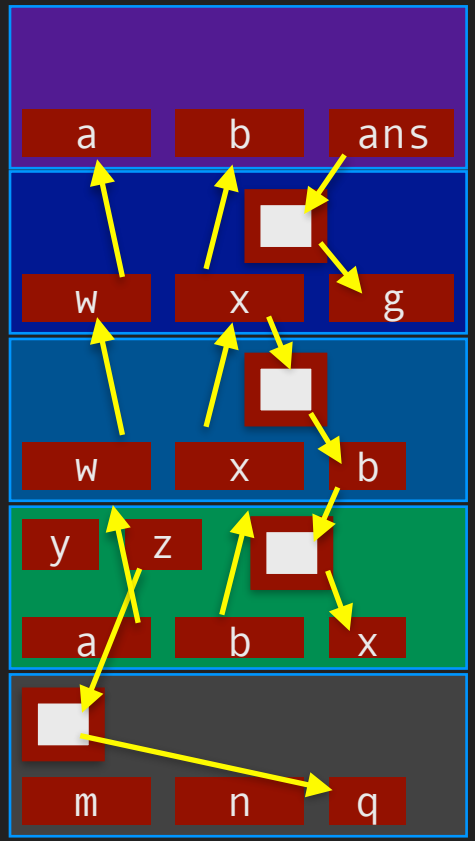
- A function must be *declared* (if not defined) before it is used in another function. Declaration specifies input/output types:
  - e.g., `int gcd(int a, int b);`
- Variables in a function are "local" to that function
  - Can have variables with same name in different functions
- It is allowed to define "global variables" outside functions, which can be accessed by all functions
  - Typically not a good design (error prone, hard to debug). Instead pass them around as needed
    (or use mechanisms to limit how they can be accessed — later!)

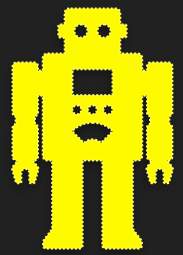




How Function Calls Work: The Stack

- `main_program` is also a function: `int main( )` (corresponding to the bottom frame in the stack)
- `main's` return value is typically used as an "error code" by the shell
- Even if no explicit `return` statement, when the control reaches the end of the function, it implicitly returns the integer `0` (taken as no error by shells)
- Can explicitly return a non-zero value to indicate an error to the shell





Functions that Return Nothing

- Functions can also carry out tasks (rather than just map inputs to outputs): like `forward(s)`, `left(d)`, etc.
- In C++, they have the same syntax as functions that return a value, but the return type is declared as `void`
  - The return statement for such functions is simply `return;` (rather than `return` *exp*`;`)
  - The return statement is optional: When control reaches (falls through) the end of the function, it returns
- A void type expression (obtained from invoking a function with return type void) cannot be stored or operated up on

Example: Drawing Olympic Rings

Demo

```
void move(double dx, double dy) {
  // move relative to current posn/angle
}
```

```
void movex(double dx) {
  move(dx,0);
}
```

```
void circle(double radius) {
  // draw a circle and come
  // back to original posn/angle
}
```

```
void movey(double dy) {
  move(0,dy);
}
```

```
int main() {
  ...
  circle(r); movex(2*r+s); circle(r); movex(2*r+s); circle(r);
  movex(-r-s/2); movey(-r-d); circle(r); movex(-2*r-x); circle(r);
}
```