Functions

Exam Announcement

Global Variables

- we have repeatedly stated that a variable's scope extends from point of declaration to the closest closing curly brace
- but the variable message has no closing curly brace

```
1  #include <iostream>
2  string message = "Hello, world!";
3
4  int main()
5  {
6     cout << message << endl;
7     return 0;
8  }
```

- message is a global variable
- this is legal and compiles
- but is extremely dangerous
- is not allowed by good programming practice

Global Variables in Gaddis

- Gaddis talks about global variables because they are part of the language
- but we will never use them
- Gaddis says you should "avoid" using global variables, but our position is much stronger: never, ever use global variables!
- all variables must be local, declared within a function

Global Constants

- in contrast to global variables, global constants are acceptable
- global constants are safe because they are constant
- global constants are visible in every function in the program

```
1  #include <iostream>
2  const string MESSAGE = "Hello, world!";
3
4  int main()
5  {
6     cout << MESSAGE << endl;
7     return 0;
8  }
```

- MESSAGE is in scope and visible in every function

Global Constants

- just because you can declare global constants does not mean you should
- declare a global constant only if it will be used in more than one function
- a constant that is used in only one function should be declared at the beginning of that function

Local Variable Lifetime

- when a function is executing, its formal parameters are in scope throughout the function body
- its local variables follow the rules of scoping we have already seen; e.g., from the point of declaration to the closest closing curly brace

Local Variable Lifetime

- when a function is executing, its formal parameters are in scope throughout the function body
- its local variables follow the rules of scoping we have already seen; e.g., from the point of declaration to the closest closing curly brace

- when the function terminates, all memory associated with its formal parameters and local variables vanishes

Local Variable Lifetime

- when a function is executing, its formal parameters are in scope throughout the function body
- its local variables follow the rules of scoping we have already seen; e.g., from the point of declaration to the closest closing curly brace

- when the function terminates, all memory associated with its formal parameters and local variables vanishes

- if the function is called again in the same program
  - the formal parameters are re-initialized by the current call's actual parameters
  - the local variables have no memory of the last time the function ran; it's always like the first time

Omitted

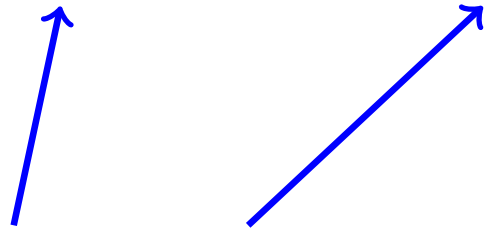
- these are topics we will not cover at this time:
  - 6.11 static local variables. This is a very important topic that will be discussed in CS181
  - 6.12 default arguments. They're useful, but we just don't need them right now.
  - 6.14 function overloading. Also can be very useful, but at this point it's hard to come up with realistic examples. We'll get to this later.
  - 6.15 exit(). At this point, using it would be just the same as a return 0 from the middle of the main function. We will discuss error handling in detail later.

Pass By Value

- an argument's value is copied into the formal parameter
- this is called pass by value: an important term

formal parameters

```
unsigned get rand in range(unsigned low, unsigned high)
{  // do some operation
   return value;
}
unsigned length = get rand in range(1, MAX LENGTH);
```

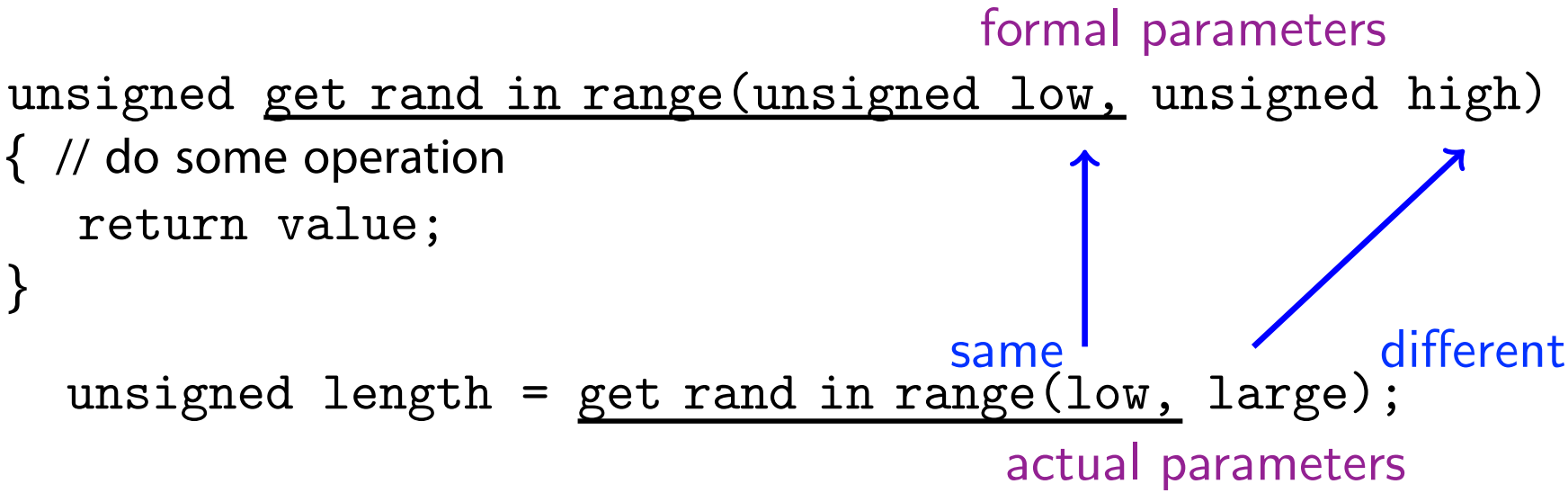


actual parameters

- once inside the function, the formal parameter (with its copied-in value) can be used as a variable
- it is pre-initialized by the call process with the value of the actual parameter

Formal and Actual Parameter Names

- students are often confused by whether formal and actual parameter names should be the same or different



Formal and Actual Parameter Names

- there is not one right answer
- the solution is to use the best name in the context
  - the context of the actual parameter is the its scope
  - a variable should have a name reflecting how it is used in its scope
  - the context of the formal parameter is the function
  - Use a name depending on how that value is used only within the function.
- the formal and actual parameters are in different scopes, so their names do not collide

Pass by Reference

- C++ has a second parameter-passing method
- reference variable
- a reference variable holds a reference to another variable and uses its value
- a formal parameter can be declared as a reference parameter by using an ampersand: &

```
1  int main()
2  {
3    int length = 12;
4    int width = 8;
5
6    cout << "Rectangle length: " << length
7         << " width " << width << endl;
8
9    int p_meter = 0, area=0;
10   calc_rect(length, width, p_meter, area);
11   cout << "Perimeter is: "<< p_meter<<" "
12        << "area is: "<< area<< endl;
13   return 0;
14 }
15
16
17 void calc_rect(int l, int w, int &p, int &a)
18 {
19
20   p = 2 * (l + w);
21   a = l * w;
22 }
```

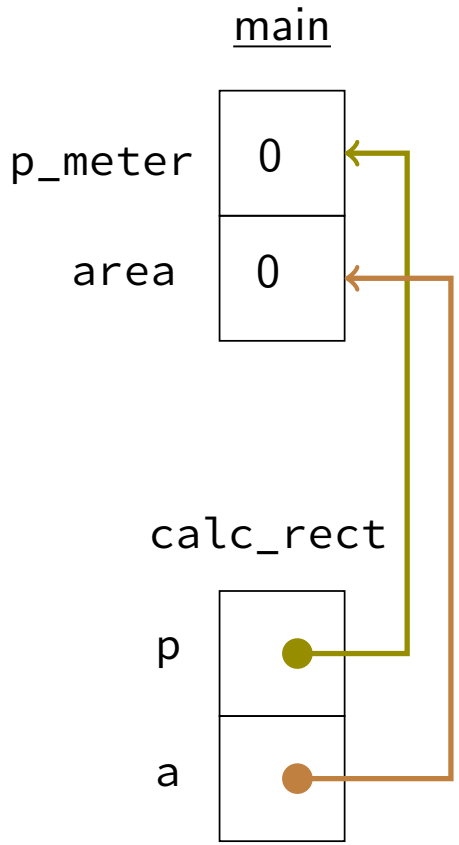
About to Run
Line 10

main

p_meter	0
area	0

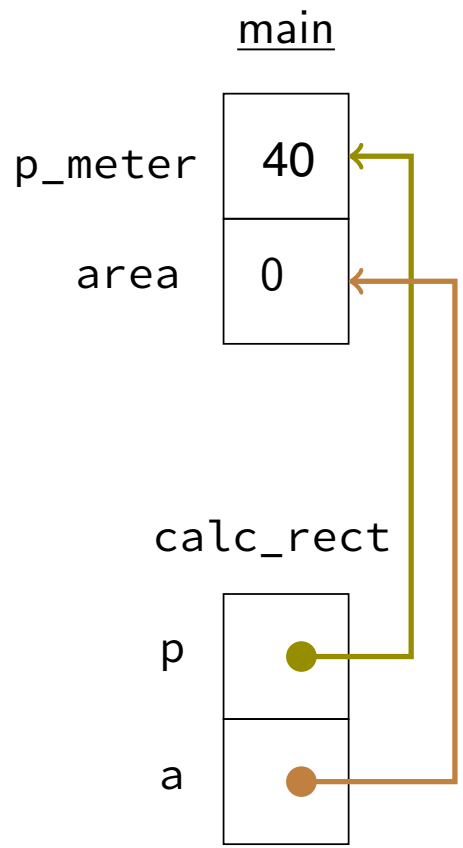
```
1  int main()
2  {
3
4    int length = 12;
5    int width = 8;
6
7    cout << "Rectangle length: " << length
8            << " width " << width << endl;
9
10   int p_meter = 0, area=0;
11   calc_rect(length, width, p_meter, area);
12   cout << "Perimeter is: "<< p_meter<<" "
13        << "area is: "<< area<< endl;
14   return 0;
15 }
16
17
18 void calc_rect(int l, int w, int &p, int &a)
19 {
20   p = 2 * (l + w);
21   a = l * w;
22 }
```

About to Run
Line 19



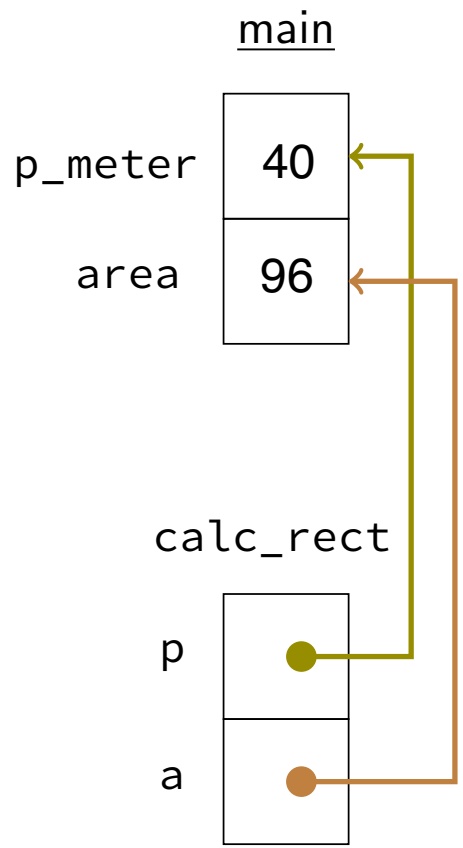
```
1   int main()
2   {
3
4     int length = 12;
5     int width = 8;
6
7     cout << "Rectangle length: " << length
8             << " width " << width << endl;
9
10    int p_meter = 0, area=0;
11    calc_rect(length, width, p_meter, area);
12    cout << "Perimeter is: "<< p_meter<<" "
13         << "area is: "<< area<< endl;
14    return 0;
15  }
16
17
18  void calc_rect(int l, int w, int &p, int &a)
19  {
20    p = 2 * (l + w);
21    a = l * w;
22  }
```

After running Line 19



```
1   int main()
2   {
3
4     int length = 12;
5     int width = 8;
6
7     cout << "Rectangle length: " << length
8             << " width " << width << endl;
9
10    int p_meter = 0, area=0;
11    calc_rect(length, width, p_meter, area);
12    cout << "Perimeter is: "<< p_meter<<" "
13         << "area is: "<< area<< endl;
14    return 0;
15  }
16
17
18  void calc_rect(int l, int w, int &p, int &a)
19  {
20    p = 2 * (l + w);
21    a = l * w;
22  }
```

After running
Line 20



Call by Reference

- a reference variable is an alias for another variable
- any change made to the reference variable is also done to the original variable
- may use call-by-value for one of its parameters and call-by-reference for a different parameter

Arguments of Reference Parameters

- only variables may be used as arguments for reference parameters
- any attempt to pass a non-variable argument
  - a literal
  - a constant
  - an expression
- `is an error`

```
calc_perimeter(length, width, p_meter); // ok!
calc_perimeter(length, width, PERIMETER); // error! constants
swap_values(length, width, 10); // error! literals
```

Call by Value or Reference

- when should you use call by value vs. call by reference?
- use call by value
  - when the function needs a value but the calling function does not expect the value to change
  - when the arguments are literals, constants, or expressions
- use call by reference
  - when the function needs to change a variable that exists in the calling function
  - when the function needs to return more than one value to the calling function

A Note on Style

for a reference parameter declaration, where exactly does the ampersand go?

1. attached to the parameter name	int foo(int &bar);
2. attached to the type	int foo(int& bar);
3. attached to neither one	int foo(int & bar);

- All of the above approaches work the same.

  Use one style that you prefer.

Concluding note: Function Design

- a best practice of programming is that a function should do only one thing
- it may take a number of steps to do it
- but only one overall task should be accomplished

Concluding note: Function Design

- a best practice of programming is that a function should do only one thing
- it may take a number of steps to do it
- but only one overall task should be accomplished

- a function named
  - compute_average_and_assign_grade

  represents poor design
- this should be written as two functions
  - compute_average
  - assign_grade