Functions

Class 20

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 totally 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 top 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, a term you will have to know

formal parameters

```
unsigned get_rand_in_range(unsigned low, unsigned high)
{
  ...
  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

see program hello_multiple.cpp

Formal and Actual Parameter Names

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

formal parameters

```
unsigned get_rand_in_range(unsigned low, unsigned high)
{
  ...
  return value;
```

same

different

```
unsigned length = get_rand_in_range(low, biggest);
```

actual parameters

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 calling scope
  - a variable should have a name reflecting how it is used in the calling scope
  - the context of the formal parameter is the function
  - thinking only of the function as a microcosm, what is the best name for how that value is used only within the function, totally ignoring the calling scope

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 calling scope
  - a variable should have a name reflecting how it is used in the calling scope
  - the context of the formal parameter is the function
  - thinking only of the function as a microcosm, what is the best name for how that value is used only within the function, totally ignoring the calling scope
- do not artificially pick different names just to be different
- 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
- a reference variable is a variable that does not hold a value
- instead, a reference variable holds a reference to another variable which does hold a value
- a formal parameter can be declared as a reference parameter by using an ampersand: &

```
1  int main()
2  {
3    unsigned length = 12;
4    unsigned width = 8;
5
6    cout << "The rectangle is " << length <<
7      " long and " << width << " wide" << endl;
8
9    swap_values(length, width);
10
11   cout << "Now the rectangle is " << length <<
12     " long and " << width << " wide" << endl;
13   return 0;
14 }
15
16 void swap_values(unsigned& value_a,
17                  unsigned& value_b)
18 {
19   unsigned temp = value_a;
20   value_a = value_b;
21   value_b = temp;
22 }
```

About to Run

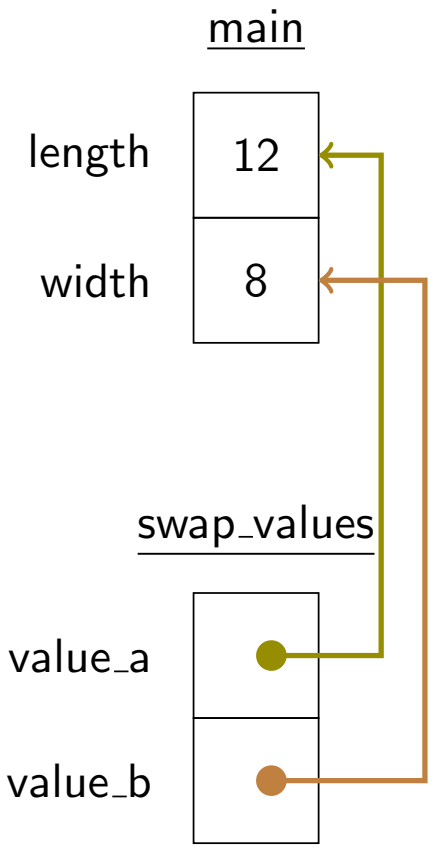
Line 9

main

length	12
width	8

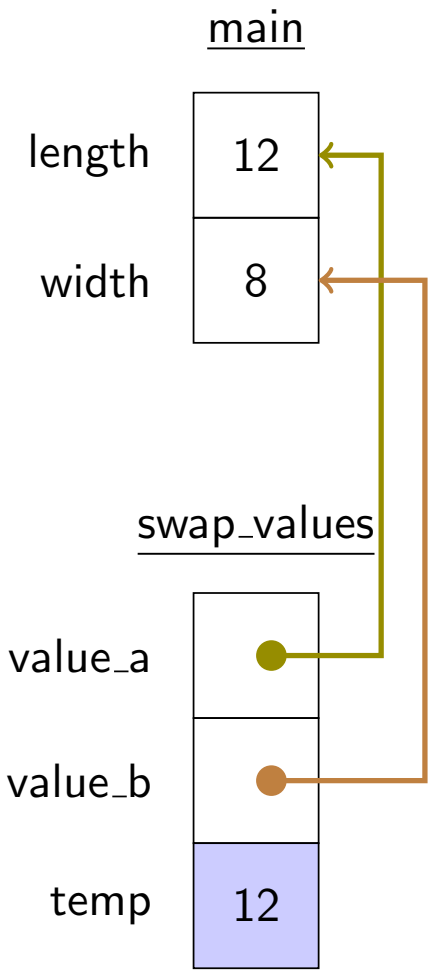
```
1  int main()
2  {
3    unsigned length = 12;
4    unsigned width = 8;
5
6    cout << "The rectangle is " << length <<
7      " long and " << width << " wide" << endl;
8
9    swap_values(length, width);
10
11   cout << "Now the rectangle is " << length <<
12     " long and " << width << " wide" << endl;
13   return 0;
14 }
15
16 void swap_values(unsigned& value_a,
17                  unsigned& value_b)
18 {
19   unsigned temp = value_a;
20   value_a = value_b;
21   value_b = temp;
22 }
```

About to Run
Line 19



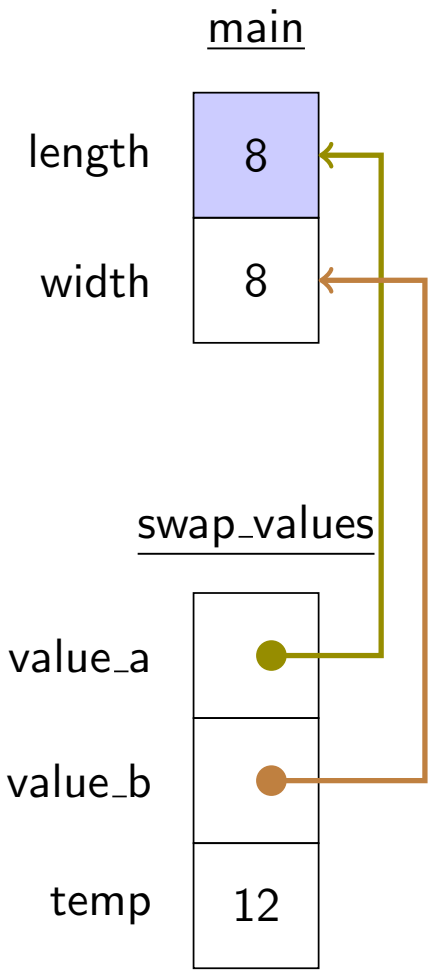
```
1  int main()
2  {
3    unsigned length = 12;
4    unsigned width = 8;
5
6    cout << "The rectangle is " << length <<
7      " long and " << width << " wide" << endl;
8
9    swap_values(length, width);
10
11   cout << "Now the rectangle is " << length <<
12     " long and " << width << " wide" << endl;
13   return 0;
14 }
15
16 void swap_values(unsigned& value_a,
17                  unsigned& value_b)
18 {
19   unsigned temp = value_a;
20   value_a = value_b;
21   value_b = temp;
22 }
```

About to Run
Line 20



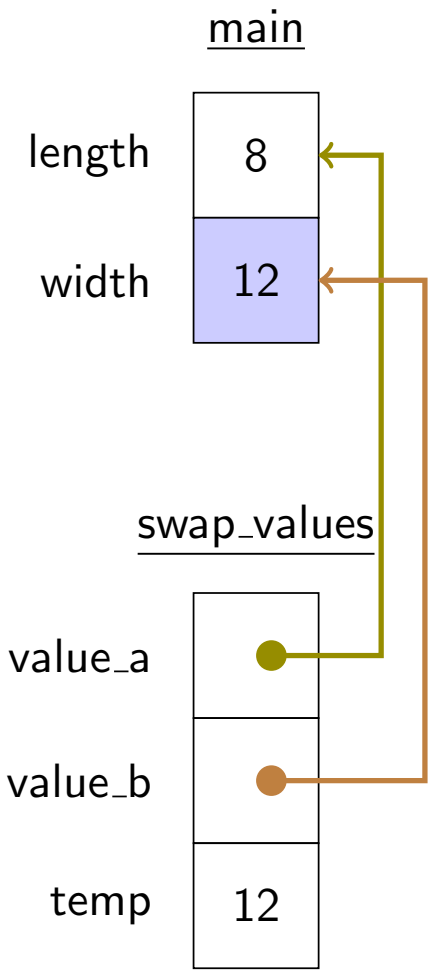
```
1  int main()
2  {
3    unsigned length = 12;
4    unsigned width = 8;
5
6    cout << "The rectangle is " << length <<
7      " long and " << width << " wide" << endl;
8
9    swap_values(length, width);
10
11   cout << "Now the rectangle is " << length <<
12     " long and " << width << " wide" << endl;
13   return 0;
14 }
15
16 void swap_values(unsigned& value_a,
17                  unsigned& value_b)
18 {
19   unsigned temp = value_a;
20   value_a = value_b;
21   value_b = temp;
22 }
```

About to Run
Line 21



```
1  int main()
2  {
3    unsigned length = 12;
4    unsigned width = 8;
5
6    cout << "The rectangle is " << length <<
7      " long and " << width << " wide" << endl;
8
9    swap_values(length, width);
10
11   cout << "Now the rectangle is " << length <<
12     " long and " << width << " wide" << endl;
13   return 0;
14 }
15
16 void swap_values(unsigned& value_a,
17                  unsigned& value_b)
18 {
19   unsigned temp = value_a;
20   value_a = value_b;
21   value_b = temp;
22 }
```

About to Run
Line 22



```
1   int main()
2   {
3     unsigned length = 12;
4     unsigned width = 8;
5
6     cout << "The rectangle is " << length <<
7       " long and " << width << " wide" << endl;
8
9     swap_values(length, width);
10
11    cout << "Now the rectangle is " << length <<
12      " long and " << width << " wide" << endl;
13    return 0;
14  }
15
16  void swap_values(unsigned& value_a,
17                   unsigned& value_b)
18  {
19    unsigned temp = value_a;
20    value_a = value_b;
21    value_b = temp;
22  }
```

About to Run
Line 11

main

length	8
width	12

Call by Reference

- a reference variable is an alias for another variable
- any change made to the reference variable is actually done to "real" variable
- by using a reference parameter, a function may change the value of a variable that exists in a different function's scope
- a function may use call-by-value for one of its parameters and call-by-reference for a different parameter

Call by Reference

- a reference variable is an alias for another variable
- any change made to the reference variable is actually done to "real" variable
- by using a reference parameter, a function may change the value of a variable that exists in a different function's scope
- a function may use call-by-value for one of its parameters and call-by-reference for a different parameter

- based on how we are calling getline in the current lab, what must the function signature of getline be?
  `while (getline(data_file, student_name))`

Call by Reference

- a reference variable is an alias for another variable
- any change made to the reference variable is actually done to "real" variable
- by using a reference parameter, a function may change the value of a variable that exists in a different function's scope
- a function may use call-by-value for one of its parameters and call-by-reference for a different parameter

- based on how we are calling getline in the current lab, what must the function signature of getline be?
  `while (getline(data_file, student_name))`
  `bool getline(ifstream& x, string& y)`
  (we don't know what x and y actually are)

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

```
swap_values(length, width); // ok!
swap_values(MAX_LENGTH, MAX_WIDTH); // error! constants
swap_values(5, 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 scope does not expect it to change
  - when the function needs to return zero or one values to the calling scope
  - 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 scope
  - when the function needs to return more than one value to the calling scope

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);`
4. random and inconsistent

- Gaddis does the first
- the second makes the most sense, so that's what I do
- the third avoids choosing
- the fourth is clearly wrong

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

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