C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

Mapping zyBooks Chapters

Topic	zyBooks Chapter
Functions	6.1, 6,15
Function Designs	6.18
Unit Testing (Functions)	6.7
Function Call Stack and Variable Scopes	6.8, 6.15
Argument Passing	6.10
Passing strings/vectors/arrays as Parameters	6.11, 6.12, 6.13, 6.14
Default Arguments	6.16
Overloading	6.17
Header Files	6.19, 6.20, 6.21
Makefile	16.4

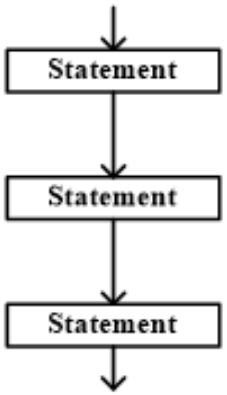
Self-study Chapters: 6.2, 6.3, 6.4, 6.5, 6.6, 6.9, 6.22, 6.23

Functions

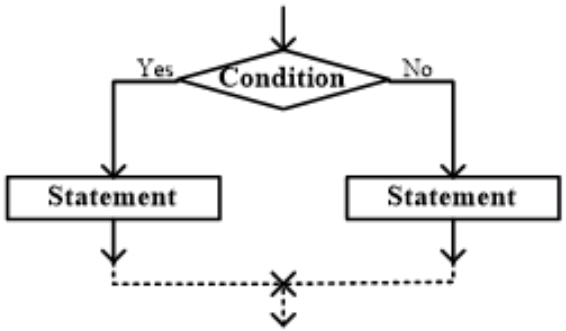
Structured Programming

- A programming paradigm
  - Improve the clarity quality and development time of computer program
  - Make extensive use of the **structured control flow**:

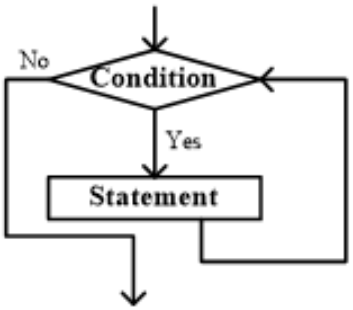
sequence



selection



iteration



  - Subroutines: functions, methods
  - Blocks: treat groups of statements as one statement

```
for (i = 0; i < 10 ; i ++)
{
    count = count + 1;
    num = num + i;
}
```
braces

```
while current <= n:
    sum = sum + current
    current = current + 1
```
indentation

Review - The First Program

- A simple C++ program form

```
directives

int main()
{
    statements
}
```

- Example

```
#include <iostream>
using namespace std;

int main() {
  int wage;

  wage = 20;

  cout << "Salary is ";
  cout << wage * 40 * 52;
  cout << endl;

  return 0;
}
```

A program starts in `main()` function

- Execute the statements within braces {}
- One statement at a time
- Each statement ends with a semicolon, as English sentences end with a period

Functions

- A C++ program is a collection of functions
- The form of a basic function:

```
return-val name-of-func (args) {
        body of function
}
```

Review - Functions

- What is a function?
  - A small program, with its own declarations and statements
- Benefits
  - Divide a program into small pieces that are easier for people to understand and modify
  - Avoid duplicating code that's used more than once
- The form of a function in C++

```
return-val name-of-function (list of formal parameters)
{
    body of function
}
```

- Example

```
int main()
{
  cout << "Hello World!"<< endl;
  return 0;
}
```

Review - Functions

- Example: a function returns computed square

```
#include <iostream>
using namespace std;

int ComputeSquare(int numToSquare) {
   return numToSquare * numToSquare;
}

int main() {
   int numSquared;

   numSquared = ComputeSquare(7);
   cout << "7 squared is " << numSquared << endl;

   return 0;
}
```

Function Declaration

- What happen if `ComputerSquare` is placed after `main`?

```
#include <iostream>
using namespace std;

int main() {
   int numSquared;

   numSquared = ComputeSquare(7);
   cout << "7 squared is " << numSquared << endl;

   return 0;
}

int ComputeSquare(int numToSquare) {
   return numToSquare * numToSquare;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:7:17: error: ‘ComputeSquare’ was not declared in this scope
    7 |    numSquared = ComputeSquare(7);
      |                 ^~~~~~~~~~~~~
```

Function Declaration

- What happen if `ComputerSquare` is placed after `main`?
  - When the compiler encounters the first call of `ComputerSquare` in main, it has no information about `ComputerSquare`.
  - It doesn't know how many parameters `ComputerSquare` has, what the types of these parameters are, or what kind of value `ComputerSquare` returns
- Function declaration (or is called function prototype)
  - Declare each function before calling it
  - Provides the compiler with a brief glimpse at ta function whose full definition will appear later.
  - The declaration of a function must be consistent with the function's definition

Function Declaration

- Add function declaration in the original program:

```
#include <iostream>
using namespace std;

int ComputeSquare(int numToSquare); /* declaration */

int main() {
   int numSquared;

   numSquared = ComputeSquare(7);
   cout << "7 squared is " << numSquared << endl;

   return 0;
}

int ComputeSquare(int numToSquare) { /* definition */
   return numToSquare * numToSquare;
}
```

Function Designs

Program and Function

- Macro Level
  - Programs consumes input and produces output
- Micro Level
  - Programs are made up of functions, which also consume input and produce output
- Function perform single tasks. Combining functions together produces a program

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- Function prototype:
  The contract - the function name, its inputs and outputs (including the data types of the input and output).

```
int round_sum(double x, double y);
```

  - Name of the function: `round_sum`. The name should describe what the function does
  - Inputs are `x` and `y` both are the type `double` (floating point)
  - Output is an integer (in C/C++ terminology, the function returns an `int`)

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- What is the purpose of the function?
  What is the problem the function is trying to solve?
  - Find the sum of two integers and round to the nearest integer
  - Purpose is written as a comment

```
/**
 * Find the sum of two integers and round
 * to the nearest integer
 */
int round_sum (double x, double y);
```

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- In addition to giving the general purpose of the function, you also comment the inputs and the output

```
/**
 * Find the sum of two integers and round
 * to the nearest integer
 * @param x the first addend
 * @param y the second addend
 * @param the rounded sum of x + y
 */
int round_sum (double x, double y);
```

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- Think up test cases for the function
  - Given input(s) what is the **expected** output.
  - This is a mental or paper and pencil work. Do not code
  - Reinforces to you how the function works.
  - Test Cases: 1.3 + 1.3 = 3, or 1.2 + 1.2 = 2.
- Now you are ready to write the body of the function
  - This is a translation of the problem into C syntax

```
int round_sum (double x, double y)
{
    /* round () is found in math .h */
    return (int) round (x + y);
}
```

- Compile the code to see if there are any errors

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- Now it is time to test the function with hard coded values

```
int main (void)
{
        double x = 1.3;
        double y = 1.3;
        cout << "rounded sum = ";
        cout << round_sum(x, y) << endl;
        return 0;
}
```

  - Compile and run program
  - Check that the function produces the expected output
  - If it doesn't, it is time to debug
  - If it does, time to run another test or move onto another function

Function Design

Problem: *Write a function that sums two numbers and rounds them to the nearest integer*

- Only after **ALL** functions have been tested and working properly, do you work on the input side of the program.
- Before you code input, test all functions with hard coded values

```
int main (void)
{
        double x = 1.3;
        double y = 1.3;
        cout << "rounded sum = ";
        cout << round_sum(x, y) << endl;
        return 0;
}
```

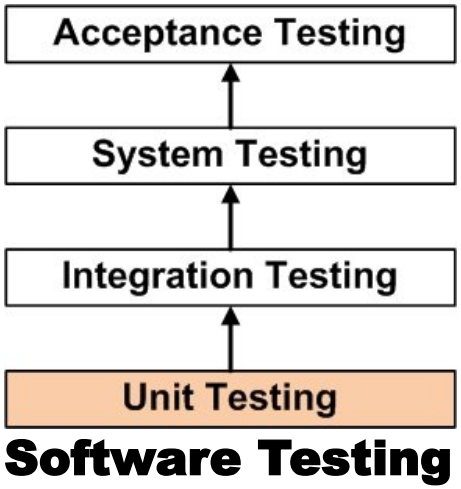
Change to user input statements

Function Design Summary

- What problem are you trying to solve? Functions solve simple problems.
- Write the contract (function prototype, name inputs, output)
- Write the purpose (comment function)
- Think of test cases (mental or paper and pencil work)
- Write body of function. Compile but do not run program.
- Test function with test cases and hard coded values. Compile and run program.
- If it works, move onto next function. If not, debug function.
- Only after successfully testing all functions, do you code input.

Unit Testing (Functions)

Unit Testing



- Definition: a software testing method
  - Individual units/components of a software are tested
- Unit
  - Is the smallest testable part of any software
  - Has one or a few inputs
  - Has a single output
- A unit in procedural programming
  - Maybe be an individual program, function, procedure, etc.
- A unit in object-oriented programming
  - A method, which may belong to a base/super class, abstract class, or derived/child class

http://softwaretestingfundamentals.com/unit-testing/

Testbench

- A unit test is typically conducted by creating a testbench (test harness)
  - A separate program whose sole purpose is to check that a function returns correct output values for a variety of input values
  - Each unique set of input values is know and a test vector
  - Example

```
cout << "0:0, expecting 0, got: "    << HrMinToMin(0, 0)  << endl;
cout << "0:1, expecting 1, got: "    << HrMinToMin(0, 1)  << endl;
cout << "0:99, expecting 99, got: "  << HrMinToMin(0, 99) << endl;
cout << "1:0, expecting 60, got: "   << HrMinToMin(1, 0)  << endl;
cout << "5:0, expecting 300, got: "  << HrMinToMin(5, 0)  << endl;
cout << "2:30, expecting 150, got: " << HrMinToMin(2, 30) << endl;
```

```
0:0, expecting 0, got: 0
0:1, expecting 1, got: 1
0:99, expecting 99, got: 99
1:0, expecting 60, got: 0
5:0, expecting 300, got: 0
2:30, expecting 150, got: 30
```

Error Handling

- `assert` macro
  - Statements used to test assumptions made by programmer
  - Write diagnostic information to the standard error file
- Example
  - https://www.geeksforgeeks.org/assertions-cc/

```
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    int x = 7;
    /* Some big code in between and let's say x is accidentally changed to 9 */
    x = 9;
    // Programmer assumes x to be 7 in rest of the code
    assert(x==7);
    /* Rest of the code */
    cout << "" << "end of program" << endl;
    return 0;
}
```

Testbench

- Testbench with assert

```
cout << "Testing started" << endl;

assert(HrMinToMin(0, 0)  == 0);
assert(HrMinToMin(0, 1)  == 1);
assert(HrMinToMin(0, 99) == 99);
assert(HrMinToMin(1, 0)  == 60);
assert(HrMinToMin(5, 0)  == 300);
assert(HrMinToMin(2, 30) == 150);
```

```
Testing started
Assertion failed: (HrMinToMin(1, 0) == 60), function main, file main.cpp, line 20.
```

- Testbench with branches

```
if ( HrMinToMin(0, 0) != 0 ) {
   cout << "0:0, expecting 0, got: " << HrMinToMin(0, 0) << endl;
}
```

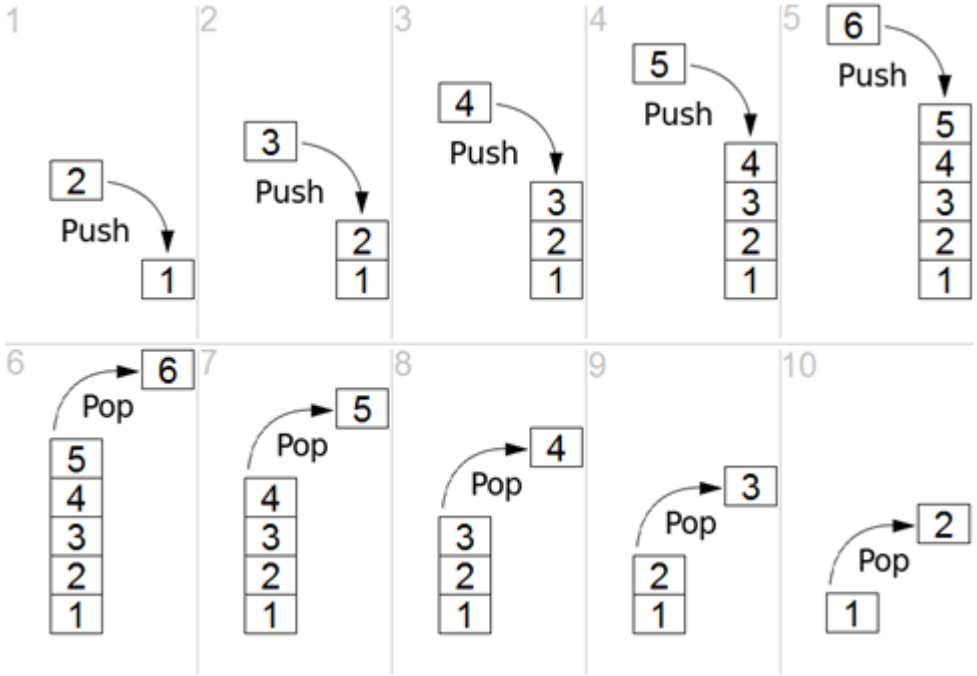
Test Vectors

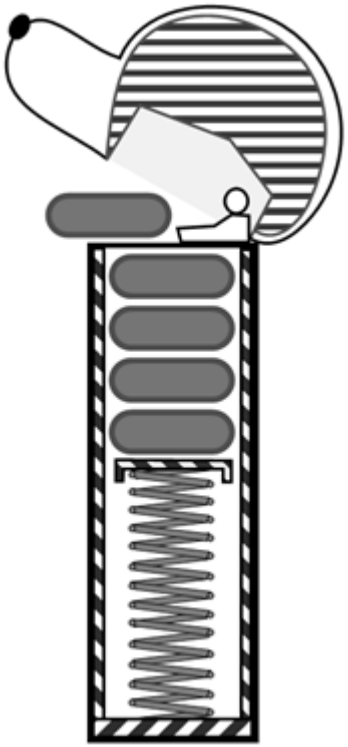
- A programmer should choose test vectors that thoroughly exercise a function
- Good test vectors examples
  - Include a number of normal cases that represent a rich variety of typical input values
    - Example: Mixing small and large numbers
  - Include border cases that represent fringe scenarios
    - Example: 0, huge positive number, huge negative numbers, etc.

Function Call Stack and Variable Scopes

Function Call Stack and Stack Frames

- Stack
  - A collection of elements with two principle operations
    - Push: ADd an element to the collection
    - Pop: Remove the last element that was added
  - LIFO: Last In First Out





Function Call Stack and Stack Frames

- Function call stack (program execution stack)
  - Support the function call/return mechanism
  - When a function calls another function, an entry is pushed onto the stack
    - The entry is called stack frame, containing the return address that the called function needs in order to return to the calling function
    - Contains some additional information such as local (automatic) variables
  - When the called function returns, the stack frame for the function call is popped.
    - The local variables are no longer exists when the stack frame is popped out.

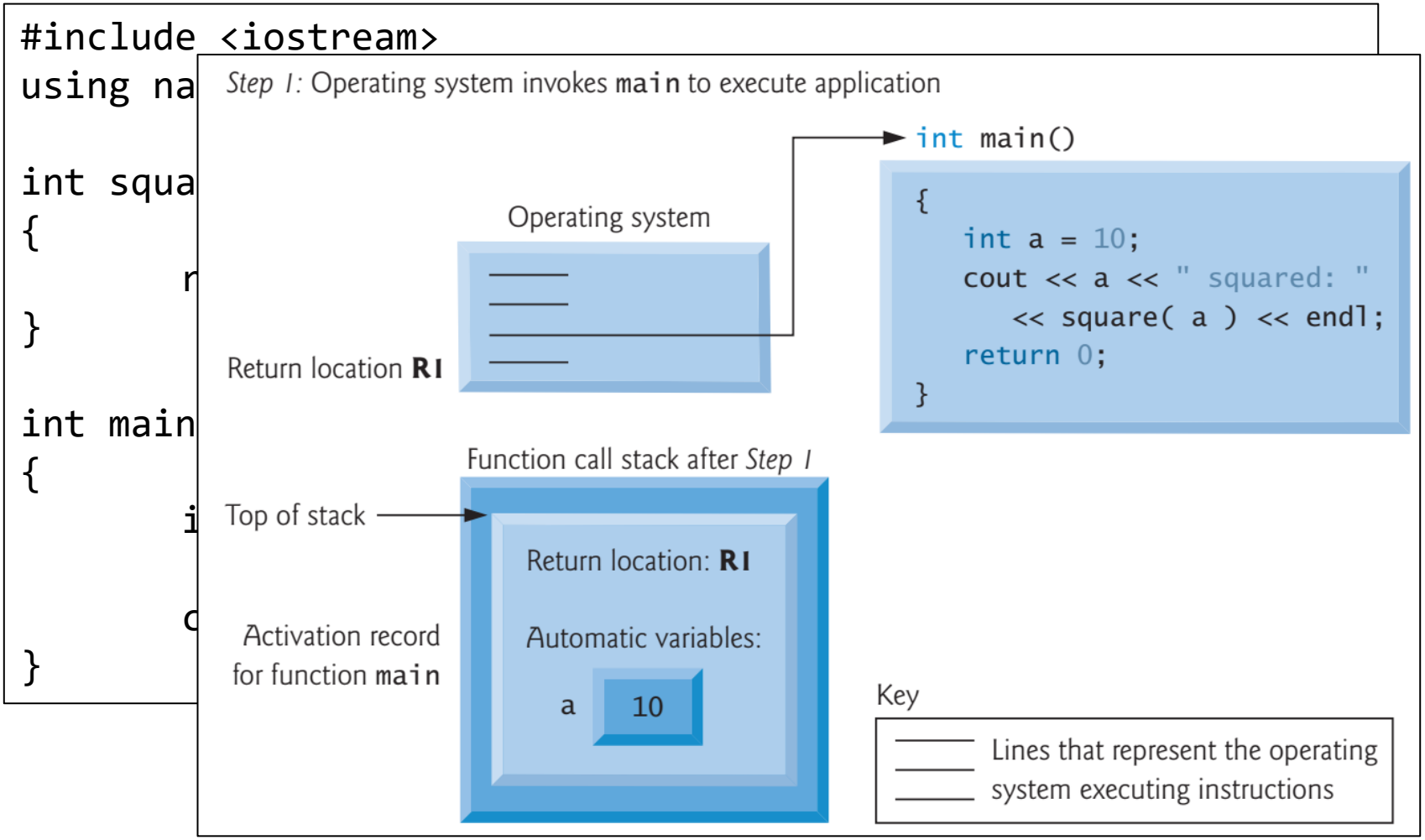
Function Call Stack and Stack Frames

- Example

```
#include <iostream>
using namespace std;

int square( int x )
{
        return x * x;
}

int main()
{
        int a = 10;

        cout << a << " squared: " << << endl;
}
```

Function Call Stack and Stack Frames

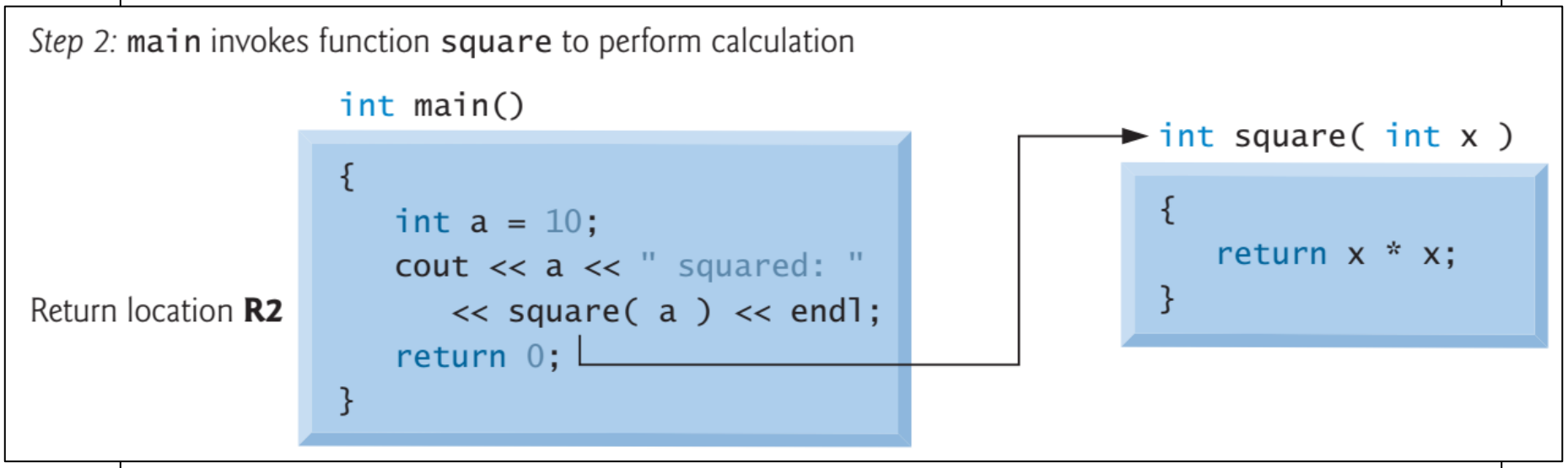
- Example



Function Call Stack and Stack Frames

- Example

```
#include <iostream>
using namespace std;
```

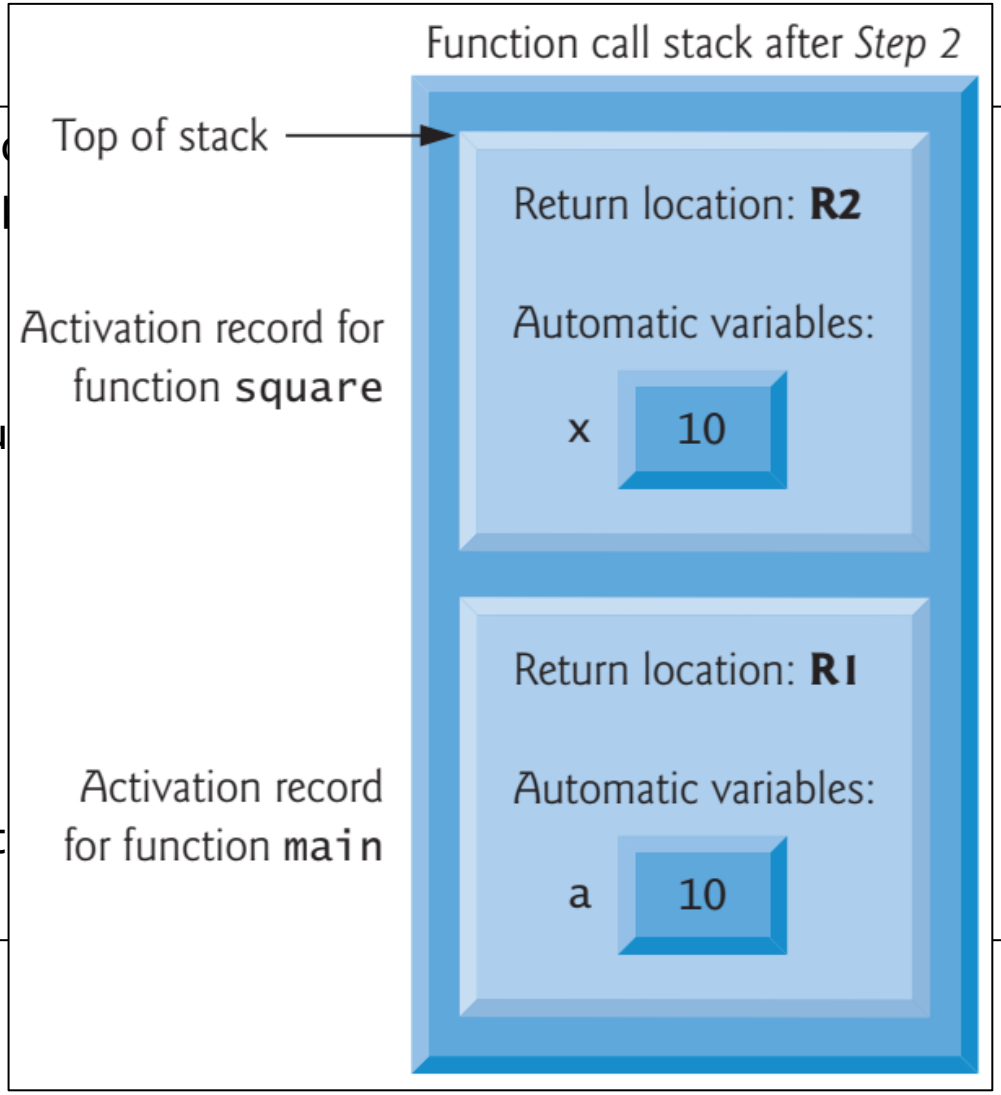



```
        cout << a << " squared: " << << endl;
}
```

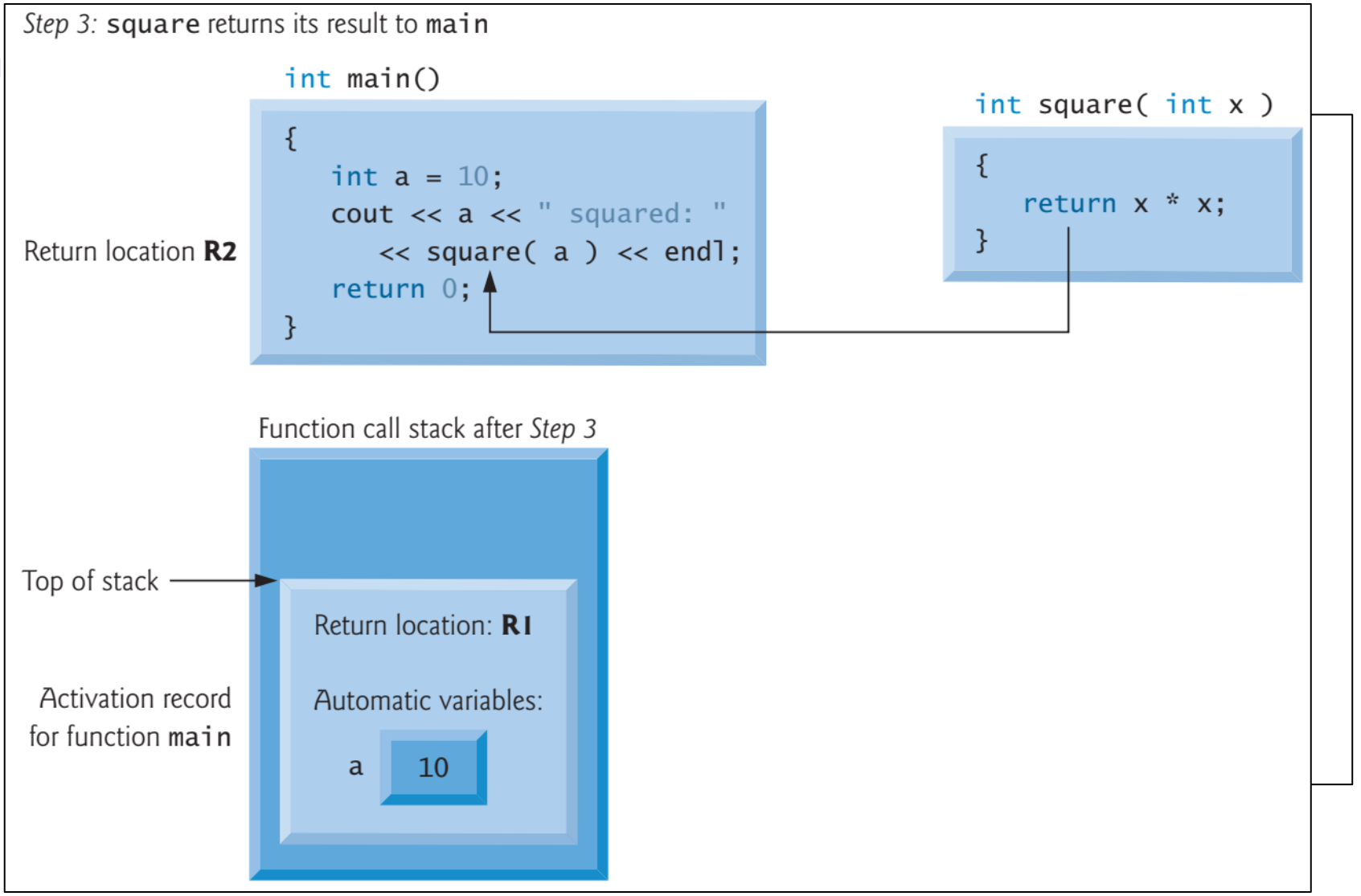
Function Call Stack and Stack Frames

- Example

```
#include <i
using namesp

int square(
{
        retu
}

int main()
{
        int

        cout
}
```



Function Call Stack and Stack Frames



Variable Scopes

- The name of a defined variable or function item is only visible to part of a program
- Local variable
  - A variable declared in a function has scope limited to inside that function
  - The scope starts after the declaration until the function's end

Variable Scopes

- Global variables
  - A variable declared outside any function
  - Variable's scope extends after the declaration to the file's end
  - If a function's local variable (including a parameter) has the same name as a global variable, then in that function the name refers to the local item and the global is inaccessible
  - Are typically limited to `const` variables like the number of centimeters per inch above
  - Good practice is to minimize the use of non-`const` global variables
  - Global Variables Are Bad
    http://c2.com/cgi/wiki?GlobalVariablesAreBad
  - Global Variables Are Evil
    http://koopman.us/bess/chap19_globals.pdf

Argument Passing

Swap

- How do you swap two variables?

```
#include <iostream>
using namespace std;


void print_vars(int a, int b)
{
            cout << "a = " << a << "\tb = " << b << endl;
}

int main(void)
{

            int a = 6;
            int b = 9;

            print_vars(a,b);

            /* swap */

            print_vars(a,b);

            return 0;
}
```

Swap

- How do you swap two variables?

```
int main(void)
{
        int tmp;
        int a = 6;
        int b = 9;

        print_vars(a,b);

        /* swap */
        tmp = a;
        a = b;
        b = tmp;

        print_vars(a,b);

        return 0;
}
```

- How about building a swap function and swap the data in the function

Swap

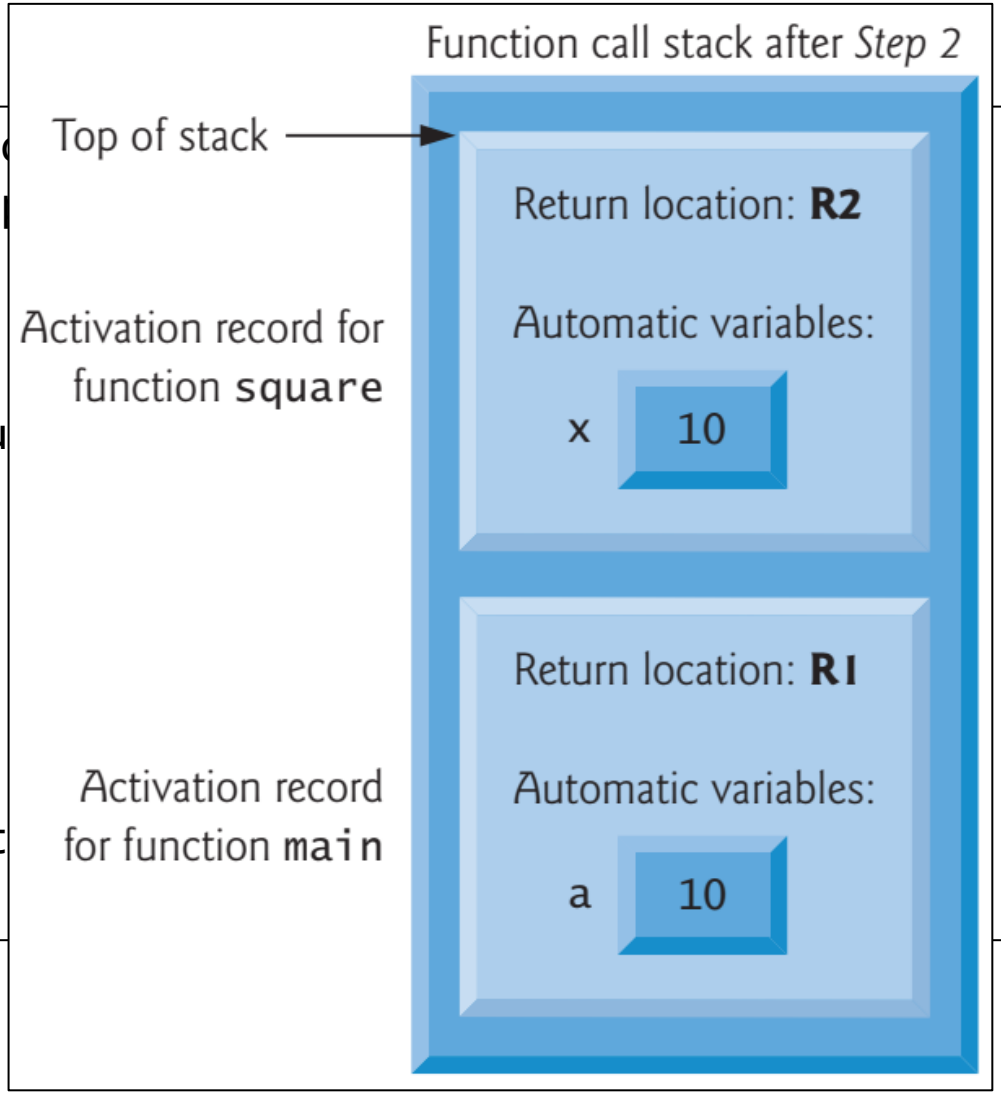
```
void swap(int a, int b)
{
        int tmp = a;
        a = b;
        b = tmp;
        cout << "in function swap: ";
        print_vars(a,b);
}

int main(void)
{
        int a = 6;
        int b = 9;
        cout << "in main - before swap: ";
        print_vars(a,b);

        swap(a, b);

        cout << "in main - after swap: ";
        print_vars(a,b);

        return 0;
}
```

```
in main - before swap: a = 6    b = 9
in function swap: a = 9 b = 6
in main - after swap: a = 6     b = 9


...Program finished with exit code 0
Press ENTER to exit console.
```

Review - Function Call Stack and Stack Frames

- Example

```
#include <i
using namesp

int square(
{
        retu
}

int main()
{
        int

        cout
}
```



Passing Arguments to Functions

- Ways to pass arguments to functions
  - Pass by value
    - A copy of the argument's value is made and pass to the called function
    - Changes to the copy do not affect an original variables value in the caller
  - Pass by reference
    - The caller allows the called function to modify the original variable's value
- In C, all arguments are passed by value
  - Pointers is used to achieve pass-by reference by using the address operator and the indirection operator.
  - Array arguments are automatically passed by reference for performance reason

Reference Parameters in C++

- Performing pass-by-reference in C++
  - The caller gives the called function the ability to access the caller's data directly, and to modify the data
  - Indicating that a function parameter is passed by reference:
    `type& variable`
  - Example:
    `int& count`
    is pronounced "`count` is a reference to an `int`."
- Update the swap function

```
void swap(int& a, int& b)
{
          int tmp = a;
          a = b;
          b = tmp;
          cout << "in function swap: ";
          print_vars(a,b);
}
```

```
in main - before swap: a = 6    b = 9
in function swap: a = 9 b = 6
in main - after swap: a = 9     b = 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Reference Variables

- Reference can also be used as aliases for other variables
- Example

```
int count = 1;       // declare integer variable count
int& cRef = count;   // create cRef as an alias for count
++cRef;              // increment count (using its alias cRef)
```

- Reference variables **must** be initialized in their declaration
- Example of an incorrect reference usage

```
int x = 3;
int& y;         // Error: y must be initialized

cout << "x = " << x << endl << "y = " << y << endl;
y = 7;
cout << "x = " << x << endl << "y = " << y << endl;
```

Avoid Assigning Pass-by-Value Parameters

- Assigning a parameter can reduce code slightly, but is widely considered a lazy programming style

```
int IntMax(int numVal1, int numVal2) {
   if (numVal1 > numVal2) {
      numVal2 = numVal1; // numVal2 holds max
   }

   return numVal2;
}
```

- Assigning a parameter can mislead a reader into believing the argument variable is supposed to be updated
- Assigning a parameter also increases likelihood of a bug caused by a statement reading the parameter later in the code but assuming the parameter's value is the original passed value

Returning a Reference from a Function

- Functions can return references, but this can be dangerous
- Dangling reference
  - When returning a reference to a variable declared in the called function
  - The reference refers to an automatic variable that's discarded when the function terminates
  - The variable is said to be "undefined"
    → is called dangling reference
    → the program's behavior is unpredictable
  - Unless that variable is declared static

Passing `strings`/`vectors`/`arrays` as Parameters

Review - Reference Parameters in C++

- Performing pass-by-reference in C++
  - The caller gives the called function the ability to access the caller's data directly, and to modify the data
  - Indicating that a function parameter is passed by reference:
    `type& variable`
  - Example:
    `int& count`
    is pronounced "`count` is a reference to an `int`."
- Update the swap function

```
void swap(int& a, int& b)
{
          int tmp = a;
          a = b;
          b = tmp;
          cout << "in function swap: ";
          print_vars(a,b);
}
```

```
in main - before swap: a = 6    b = 9
in function swap: a = 9 b = 6
in main - after swap: a = 9     b = 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Passing Arguments to a Function

- Basic Rules
  - Pass all built-in types by value (things you don't need an #include for, like int, double, char, bool)
  - Pass everything else by reference, like string, vector, etc. (by `const` reference if the function should not change the argument passed, which is most of the time)

Passing a `string`/`vector` to a Function

- Example

```
// Function replaces spaces with hyphens
void StrSpaceToHyphen(string& modStr) {
   unsigned int i;   // Loop index

   for (i = 0; i < modStr.size(); ++i) {
      if (modStr.at(i) == ' ') {
         modStr.at(i) = '-';
      }
   }
}
```

  - The passed string is updated by the function: using reference (`string&`)

Passing a `string`/`vector` to a Function

- Example

```
void PrintVals(const vector<int>& vctrVals) {
   unsigned int i;  // Loop index

   // Print updated vector
   cout << endl << "New values: ";
   for (i = 0; i < vctrVals.size(); ++i) {
      cout << " " << vctrVals.at(i);
   }
   cout << endl;
}
```

  - The passed vector is not updated: using constant reference (`const vector<int>&`)

Passing an Array to a Function

- Using [ ] to indicate an array parameter
- Example

```
double CalculateAverage(double scoreVals[], int numVals) {
   int index;
   double scoreSum = 0.0;

   for (index = 0; index < numVals; ++index) {
      scoreSum = scoreSum + scoreVals[index];
   }

   return scoreSum / numVals;
}
```

Passing an Array to a Function

- Prevent the function to modify the array parameter: using `const`
- Example

```
double CalculateAverage(const double scoreVals[], int numVals) {
   int i;
   double scoreSum = 0;

   for (i = 0; i < numVals; ++i) {
      scoreSum = scoreSum + scoreVals[i];
   }

   return scoreSum / numVals;
}
```

Passing an Array to a Function

- Use the following program to test the size of the array

```
int main(void)
{
    int data[] = {43, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9, 12, 52};

    cout << "data occupies " << sizeof(data) << " bytes" << endl;
    cout << "data has " << (sizeof(data) / sizeof(int)) << " elements" << endl;
    cout << "data[0] occupies " << sizeof(data[0]) << " bytes" << endl;

    return 0;
}
```

  - Array declaration allocates a contiguous block of memory
  - The size of the array is 13 `int`s, which occupies 52 bytes (13 $\times$ 4 bytes).

Passing an Array to a Function

- Use the following program to test the size of the array in the function

```
void foo(int a[])
{
    cout << endl << "in function foo" << endl;
    cout << "data occupies " << sizeof(a) << " bytes" << endl;
    cout << "data has " << (sizeof(a) / sizeof(int)) << " elements" << endl;
    cout << "data[0] occupies " << sizeof(a[0]) << " bytes" << endl;
}

int main(void)
{
    int data[] = {43, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9, 12, 52};
    foo(data);

    return 0;
}
```

  - Ignore the warning and execute the file

Passing an Array to a Function

- Why does the function report a size of 2?
  - To save memory, the **address** of the first element of the array is passed to the formal parameter
    → Rather than making a copy of the array and then making the assignment to the function parameter, the compiler is passing the address of the first element of the array
  - Imagine if you had an array that occupied 100 MB. If every time you called a function with the array, and it had to make a copy of the array. It would be slow and you would waste a lot of memory
  - Passing the address of the array speeds things up and allows the function to access an already allocated region of memory
  - Address are 8 bytes on a 64-bit machine (4 bytes on a 32-bit machine)
  - That's why the size of the array a in function `foo()` is 8 bytes. It is the size of the array's first element's address

Passing an Array to a Function

- Correct way to pass an array to a function
  - The length of the array needs to be a function parameter as well

```
int foo (int a[], size_t size)
{
        int i = 0;

        for (i = 0; i < size ; i++) {
                /* do something with the array ,
                   element by element a[i] */
        }
        return 0;
}
```

`int` or `unsigned int` can also be the data type of array size

  - You have to know the array length before you call the function.
  - The correct way to call a function is to determine the size of the array and pass the size as a parameter to the function

Passing an Array to a Function

- Example

```
#include <iostream>
using namespace std;

int foo(int a[], size_t size)
{
    int i = 0;
    for(i = 0; i < size; i++) {
        if (a[i] % 7 == 0)
            cout << a[i] << " is divisible by 7" << endl;
    }
    return 0;
}

int main(void)
{
    size_t size;
    int data[] = {42, 5, 67, 44, 33, 25, 54, 33, 17, 6, 9, 12, 52, 66};
    size = sizeof(data) / sizeof(int);
    foo(data, size);

    return 0;
}
```

Passing a C String to a Function

- C String is a character array
- Example

```
void StrSpaceToHyphen(char modString[]) {
   int i;       // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }
}
```

Passing a C String to a Function

- Using pointer annotation

```
void StrSpaceToHyphen(char* modString) {
   int i;       // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }
}
```

- Arrays and pointers are intimately related in C
  - An array name can be thought of as a constant pointer.
  - Pointers can be used to do any operation involving array indexing

Default Arguments

Default Arguments

- Pass a default value to the parameter
  - When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrite the function all and inserts the default value of that argument
- Must be the rightmost (trailing) argument
  - If an omitted argument is not the rightmost argument in the argument, then all arguments to the right of that argument also must be omitted
- Must be specified with the first occurrence of the function name
  - Typically in the function prototype
  - If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header
- Types of default values
  - Can be any expression, including constants, global variables, or function calls

Default Arguments

- Example:

```
void PrintDate(int currDay, int currMonth, int currYear, int printStyle = 0) {
   if (printStyle == 0) {
      cout << currMonth << "/" << currDay << "/" << currYear;
   } else if (printStyle == 1) {
      cout << currDay << "/" << currMonth << "/" << currYear;
   } else {
      cout << "(invalid style)";
   }
}

int main() {
   PrintDate(30, 7, 2012, 0);
   cout << endl;

   PrintDate(30, 7, 2012); // Uses default value for printStyle
   cout << endl;

   return 0;
}
```

Default Arguments

- Example:

```
void PrintDate(int currDay = 1, int currMonth = 1, int currYear = 2000,
                                                    int printStyle = 0) {
   if (printStyle == 0) {      // American
      cout << currMonth << "/" << currDay << "/" << currYear;
   }
   else if (printStyle == 1) { // European
      cout << currDay << "/" << currMonth << "/" << currYear;
   }
   else {
      cout << "(invalid style)";
   }
}
```

```
PrintDate(30, 7, 2012, 0); // No defaults
PrintDate(30, 7, 2012);    // Defaults:                                style=0
PrintDate(30, 7);          // Defaults:                    year=2000, style=0
PrintDate(30);             // Defaults:         month=1, year=2000, style=0 (strange, but valid)
PrintDate();               // Defaults: day=1, month=1, year=2000, style=0
```

Overloading

Review - Function Declaration

- Add function declaration in the original program:

```
#include <iostream>
using namespace std;

int ComputeSquare(int numToSquare); /* declaration */

int main() {
   int numSquared;

   numSquared = ComputeSquare(7);
   cout << "7 squared is " << numSquared << endl;

   return 0;
}

int ComputeSquare(int numToSquare) { /* definition */
   return numToSquare * numToSquare;
}
```

Function Signatures

- In regular function
  - The name and the parameter-type-list of a function
- In a class member
  - The name and the parameter-type-list of a function
  - The class, concept, concept map, or the namespace
- Functions in the same scope must have unique signatures
- The compiler uses to perform overload resolution

https://stackoverflow.com/questions/2322736/what-is-the-difference-between-function-declaration-and-signature

Function Overloading

- Several functions are defined with same name
  - Have different signatures
    → Have different number, types, and order of the arguments
- Example

```
#include <iostream>
#include <string>
using namespace std;

void PrintDate(int currDay, int currMonth, int currYear) {
   cout << currMonth << "/" << currDay << "/" << currYear;
}

void PrintDate(int currDay, string currMonth, int currYear) {
   cout << currMonth << " " << currDay << ", " << currYear;
}

int main() {

   PrintDate(30, 7, 2012);
   cout << endl;

   PrintDate(30, "July", 2012);
   cout << endl;

   return 0;
}
```

```
7/30/2012
July 30, 2012
```

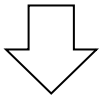
Function Overloading

- Function Overloading
  - A program has two functions with the same name but differing in the number or types of parameters
- How the compiler differentiates overloaded function
  - By their signatures, combining with a function's name and its parameter types (in order)
  - Name mangling or name decoration
    Encodes each function identifier with the number and types of its parameters → enable type-safe linkage
  - Type-safe linkage: ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters

Function Overloading

- Example of name mangling

```
int square (int x) {
          return x * x;
}
double square (double y) {
          return y * y;
}
void nothing1 (int a, float b, char c, int& d) {
}
void nothing2 (char a, int b, float& c, double& d) {
          return 0;
}
int main() {
}
```

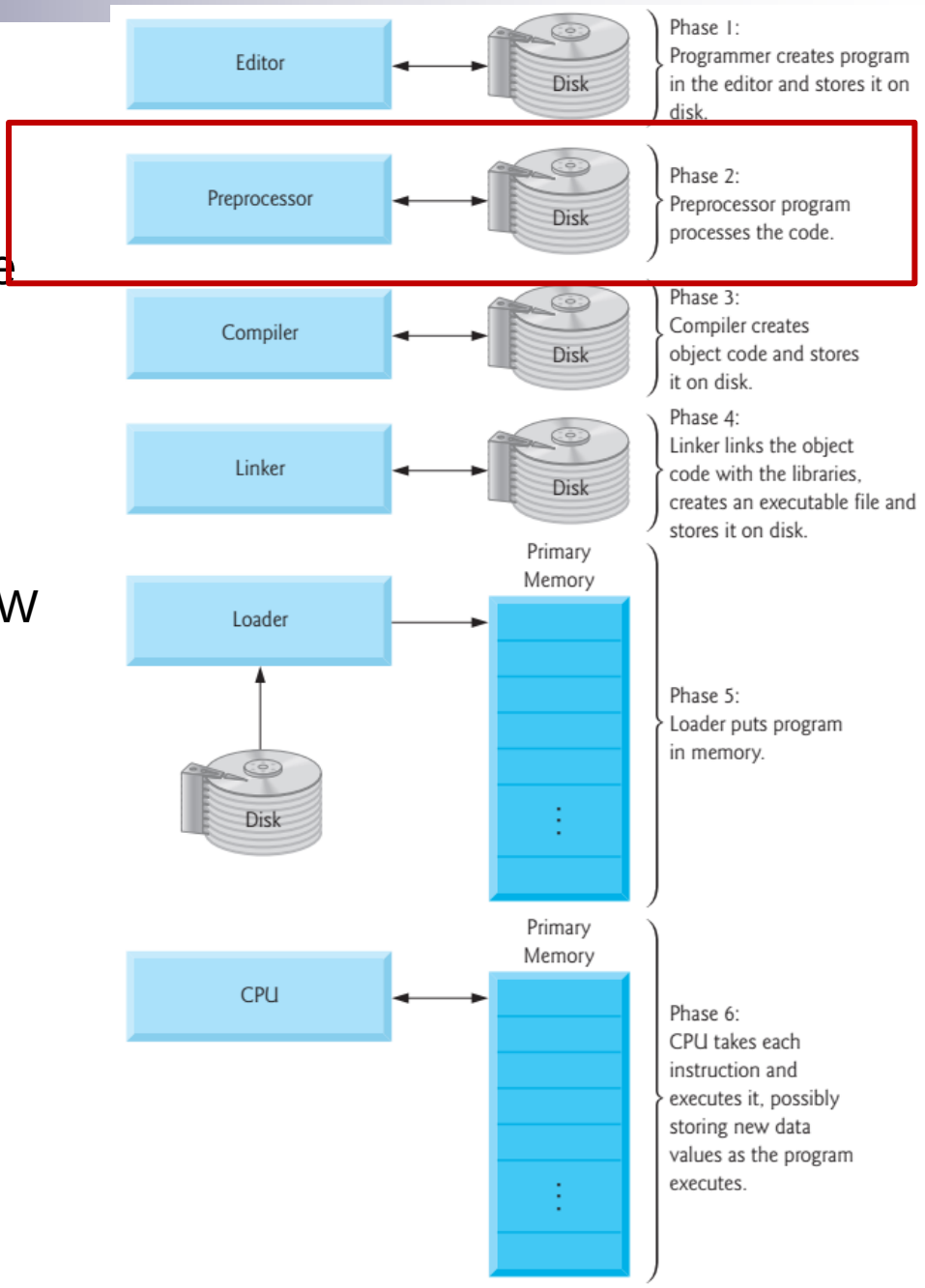


```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Header Files

Review - C++ Working Environment

- A text-editor - write source code
  - VSCode, Atom, etc.
- A compiler - translate source code into machine language
  - Linux: GNU C++ Compiler
  - Windows: Visual Studio, MinGW
  - Mac: Xcode
- A shell - a way to interact with the kernel; a means to execute the program (Unix)



Review - The First Program

- A simple C++ program form

```
directives

int main()
{
     statements
}
```

- Example

```
#include <iostream>
using namespace std;

int main() {
  int wage;

  wage = 20;

  cout << "Salary is ";
  cout << wage * 40 * 52;
  cout << endl;

  return 0;
}
```

Directives

- A language construct that specifies how a compiler should process its input
- In a C/C++ program, directives usually begin with a # character, which distinguishes them from other items.

#include <iostream>

- The information in **<iostream>** libraries are “included” into the program before it is compiled

<iostream>

- Contains information about C++’s console I/O library

Preprocessor Directives

- Lines included in the code of programs preceded by a hash sign (#)
- Are directives for the preprocessor
  - Examine the code before actual compilation of code begins
  - Resolve all these directives before any code is actually generated by regular statements
- Only across a single line of code. No semicolon is expected
- Types
  - Source file inclusion (`#include`)
  - Macro definitions (`#define`, `#undef`)
  - Conditional inclusions (`#ifdef`, `#ifndef`, `#if`, `#endif`, `#else` and `#elif`)
  - Line control (`#line`)
  - Error directive (`#error`)
  - Pragma directive (`#pragma`)

https://cplusplus.com/doc/tutorial/preprocessor/

Writing Large Programs

- Programs may consist of more than one file
  - Source files (`.cpp`)
    - Contain definitions of functions and external variables
    - Group related functions and variables into the same file
    - E.g., `stack.cpp` supports stack-related functions such as `push`, `pop`, etc.
    - Functions are more easily reused in other programs.
    - How can a function in one file call a function that's defined in another file
- Header files (`.h`)
  - Contain information to be shared among source files
  - Use `#include` directive to include
    - C/C++'s own library: `#include <filename>`
      Compiler will search the directories in which system header files reside (e.g., `/usr/include` in Linux system)
    - All other header `#include "filename.h"`
      Compiler will search the current directory, then search the directories in which system header file reside.

Writing Large Programs

- Header files (`.h`)
  - Contain information to be shared among source files
    - Macro definition. E.g., define boolean values in `boolean.h`:

```
#define TRUE 1
#define FALSE 0
```

    - Type definition. E.g., create Bool type:

```
typedef int Bool
```

    - Function prototypes. E.g., the stack.c files contain definitions of `make_empty`, `is_empty`, `is_full`, `push`, and `pop` function. The prototypes for these functions should go in the `stack.h` header file:

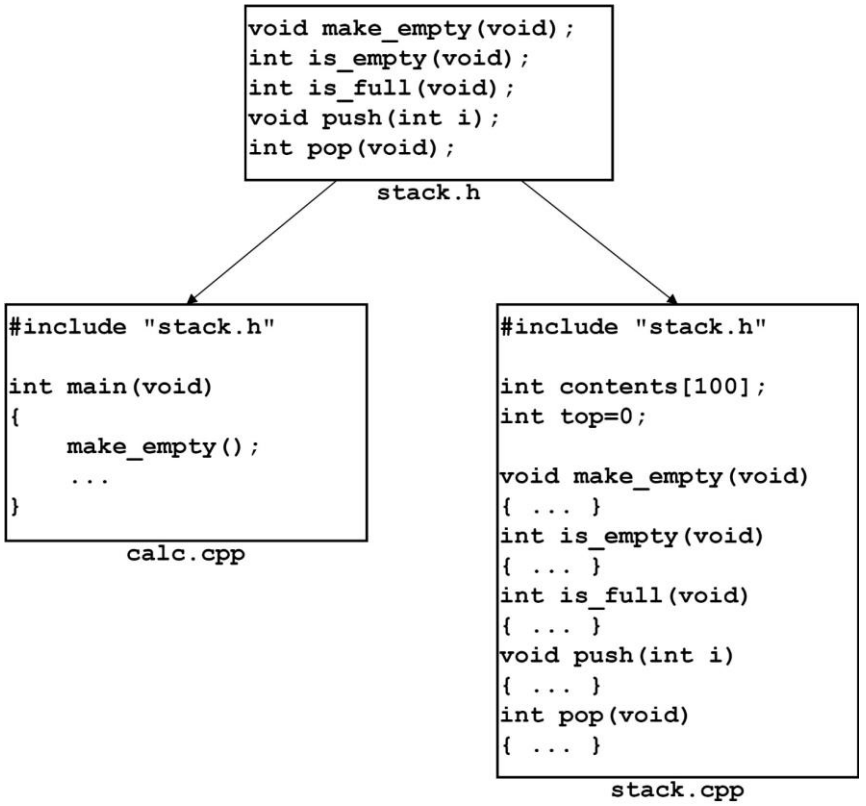
```
void makd_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

Writing Large Programs

- Header files (.h)
  - Contain information to be shared among source files
    - Function prototypes. E.g., the stack.c files contain definitions of `make_empty`, `is_empty`, `is_full`, `push`, and `pop` function. The prototypes for these functions should go in the `stack.h` header file:

```
void makd_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

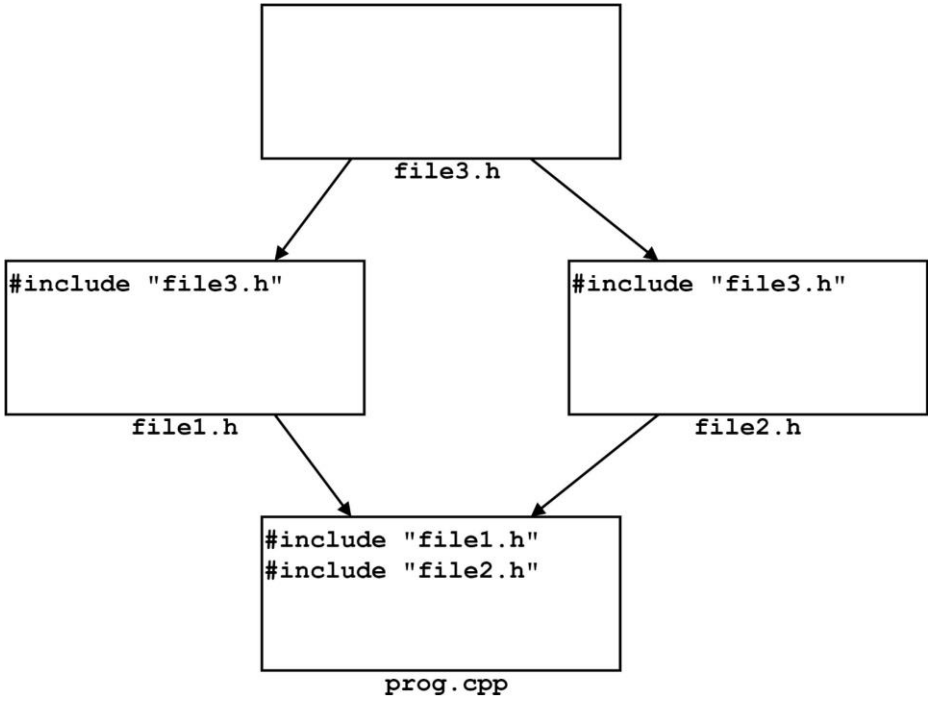
`stack.h` should be also included in `stack.cpp` so the compiler can verify that the prototypes in `stack.h` match the definition in `stack.cpp`



Protecting Header Files

- If a source file includes the same header file twice
  - Contains only macro definition, function prototypes, and/or variable declarations → no compilation error
  - Contains type definition → compilation error
- Include guard
  - To avoid the problem of double inclusion when dealing with the include directive
  - Use

```
#ifndef FILE_H
…
#endif
```



Protecting Header Files

- Example: the boolean.h file could be protected in the following way

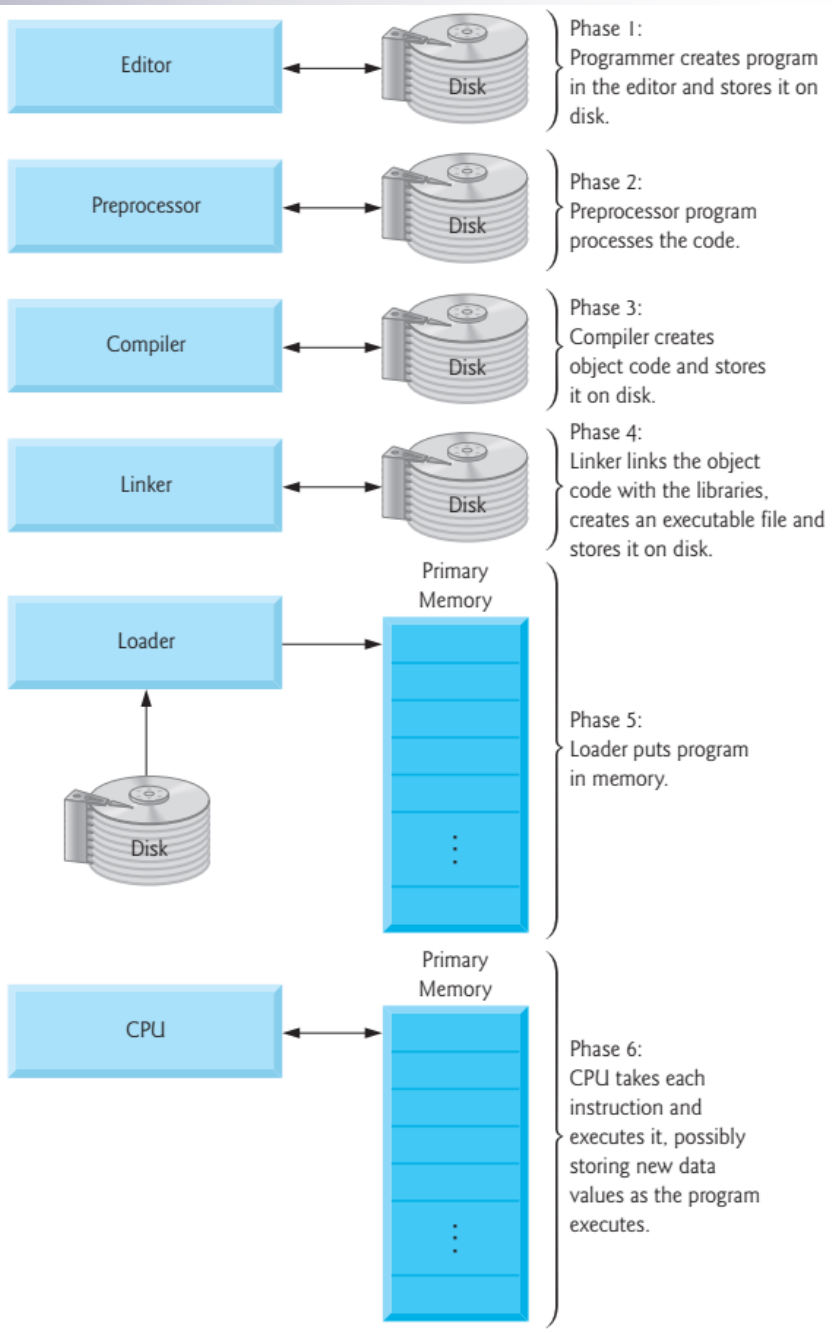
```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

  - 1st time: the `BOOLEAN_H` macro won't be defined, so the preprocessor will allow the lines between `#ifndef` and `#endif` to stay.
  - 2nd time: the preprocessor will remove the lines between `#ifndef` and `#endif`
  - Macro name: making it resemble the name of the header file is a good way to avoid conflicts with other macros. We can't name `BOOLEAN.H` because the identifier can't contain periods → `BOOLEAN_H`

Makefile

Review - C++ Working Environment

- A text-editor - write source code
  - VSCode, Atom, etc.
- A compiler - translate source code into machine language
  - Linux: GNU C++ Compiler
  - Windows: Visual Studio, MinGW
  - Mac: Xcode
- A shell - a way to interact with the kernel; a means to execute the program (Unix)



Compiling

- Take high level language and converts it to machine language
- 4 Steps to Compilation
  - Preprocessing
    `g++ -E hello.cpp`
  - Compilation (convert higher level language to assembly)
    `g++ -S hello.cpp` → output is stored in `hello.s`
  - Assembly (convert assembly to machine language)
    `g++ -c hello.cpp` → output is stored in `hello.o`
  - Linking (Add precompiled standard C library function calls or system calls to executable)
    `g++ hello.cpp` → output is stored in `a.out`
- The result is an executable file

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ ls -l
total 36
-rwxr-xr-x 1 rita rita 10736 Aug 19 18:21 a.out
-rwxr-xr-x 1 rita rita 10736 Aug 19 18:28 hello
-rw-r--r-- 1 rita rita    77 Aug 19 17:41 hello.c
-rw-r--r-- 1 rita rita  1544 Aug 19 19:55 hello.o
-rw-r--r-- 1 rita rita   474 Aug 19 19:55 hello.s
rita@CSE113:~/CSE113/exercise$
```

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
  - The three source files are first compiled into object code
  - The object files are then automatically passed to the linker.
  - The linker combines them into a single file
- Makefiles
  - Contain the information necessary to build a program
    - List the files that are part of the program
    - Describe the dependencies among the files
  - E.g.,
    - If the file `foo.cpp` includes the file `bar.h`
    - `foo.cpp` depends on `bar.h`
      → a change to `bar.h` will require us to recompile `foo.cpp`

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp

word.o: word.cpp word.h
        g++ -c word.cpp

line.o: line.cpp line.h
        g++ -c line.cpp
```

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp

word.o: word.cpp word.h
        g++ -c word.cpp

line.o: line.cpp line.h
        g++ -c line.cpp
```

There are four groups of line; each group is known as a **rule**

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp

word.o: word.cpp word.h
        g++ -c word.cpp

line.o: line.cpp line.h
        g++ -c line.cpp
```

The first line in each rule gives a target file, followed by the file on which it depends

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp

word.o: word.cpp word.h
        g++ -c word.cpp

line.o: line.cpp line.h
        g++ -c line.cpp
```

The second line is a command to be executed if the target should need to be rebuilt because of a change to one of its dependent files

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp
```

  - The first line
    - `justify` depends on the files `justify.o`, `word.o`, and `line.o`.
    - If any one of these three files has changed since the program was last built, the `justify` needs to be rebuilt
  - The second line
    - How to rebuild the files

Building a Multiple-File Program

- Compilers allow building a program in a single step
  - E.g., `g++ -o justify justify.cpp line.cpp word.cpp`
- Makefile
  - E.g.,

```
justify: justify.o word.o line.o
         g++ -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -c justify.cpp
```

  - The first line
    - `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h` (the two `.h` files are included in `justify.c`)
  - The second line
    - How to update `justify.o`
    - `-c`: compile `justify.c` into an object file but not attempt to link it

Building a Multiple-File Program

- `make` utility
  - To build (or rebuild) the program
- Details in `make` utility
  - Each command in a makefile must be preceded by a tab character, not a series of space
  - A make file is normally stored in a file named `Makefile` (or `makefile`). When the `make` utility is used, it automatically checks the current directory for a file with one of these names
  - To invoke make, use the command

    `make` ***target***

    where target is one of the targets listed in the make file

    Example:

    `make justify`

Building a Multiple-File Program

- Details in `make` utility
  - If no target is specified when make is invoked, it will be build the target of the first rule. E.g., the command

    ```
    make
    ```

    Will build the `justify` executable, since `justify` is the first target in the make file

Building a Multiple-File Program

- Variables
  - A name that represents a string of text, much like a macro in C
  - Can be used to represent file names, compiler options, executable, filepaths, or just about any other string of text that make or the shell can interpret
  - May consist of any string of characters except “`:`”, “`#`”, “`=`”, or whitespace
- Automatic variables examples
  - `$@`: The file name of the target of the rule
  - `$<`: The name of the first prerequisite
  - `$^`: The names of all the prerequisites
- Usage
  - Reduce the chance of errors caused by spelling mistakes or changes made in one section that are not propagated to the rest of the makefile
  - Make more efficient and portable makefile

Building a Multiple-File Program

- Example
  - original:

```
justify: justify.o word.o line.o
         g++ -g -Wall -o justify justify.o word.o line.o

justify.o: justify.cpp word.h line.h
           g++ -g -Wall -c justify.cpp
```

  - modified:

```
CC=g++
CFLAGS=-g -Wall
OFLAGS=-g -Wall -c

justify: justify.o word.o line.o
                $(CC) $(CFLAGS) –o $@ justify.o word.o line.o

justify.o: justify.cpp word.h line.h
                $(CC) $(OFLAGS) justify.cpp
```