

CS 103 Unit 1b – C++ Program/Control Flow

Java and C++

- C++ uses the same control structures and syntax as Java
 - if, while, for, (switch)*
- We expect you know each of the above structures **AND when and how to employ them to implement computational approaches**
- You should also be familiar with:
 - break, continue
 - The operation of nested loops (the **inner** loop performs **ALL** of its iterations for each **one** iteration of the **outer** loop)

When Do I Use a While Loop (1)

- When you **DON'T** know in advance how many times something should repeat?
 - How many guesses will the user need before they get it right?

```
#include <iostream>
using namespace std;
int main()
{
    int guess;

    int secretNum = /* some code */
    cin >> guess;
    while(guess != secretNum)
    {
        cout << "Enter guess: " << endl;
        cin >> guess;
    }

    cout << "You got it!" << endl;
    return 0;
}
```

When Do I Use a While Loop (2)

- Whenever you see, hear, or use the word 'until' in a description
- Important Tip:
 - "until x" = "while not x"
 - $\text{until}(x) \Leftrightarrow \text{while}(!x)$
 - Ex: "Keep guessing until you are correct" is the same as "keep guessing while you are NOT correct"

```
#include <iostream>
using namespace std;
int main()
{
    int guess;
    int secretNum = /* some code */
    cin >> guess;
    while(guess != secretNum)
    {
        cout << "Enter guess: " << endl;
        cin >> guess;
    }

    cout << "You got it!" << endl;
    return 0;
}
```

When Do I Use a For Loop (1)

- When you **DO KNOW** in advance (before the loop starts) how many times to iterate
 - Usually, a constant or variable that has been calculated or input from the user

```
// Program to output numbers  
// 1 through n
```

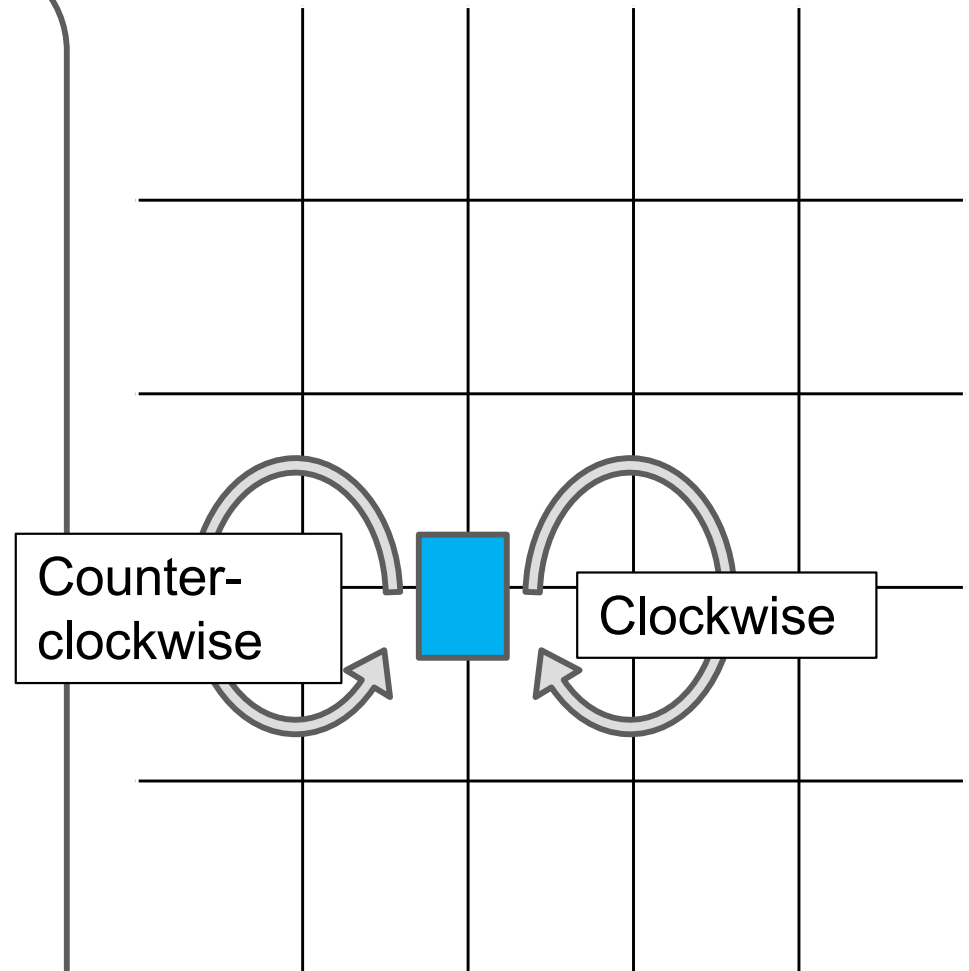
```
#include <iostream>  
using namespace std;  
int main()  
{  
    int n;  
  
    cin >> n;  
    for(int i=1; i < n; i++)  
    {  
        cout << i << endl;  
    }  
  
    return 0;  
}
```

Turn 360

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

int main() {
    // Write your code here!

    return 0;
}
```



Exercise

- Which of the following is NOT a condition to check if the integer x is in the range $[-1$ to $5]$
 - a. $x \geq -1 \ \&\& \ x \leq 5$
 - b. $-1 \leq x \leq 5$
 - c. $!(x < -1 \ || \ x > 5)$
 - d. $x > -2 \ \&\& \ x < 6$

See solutions at end of slides

Conditions and DeMorgan's

- DeMorgan's theorem says there are always two ways to express a logic condition
- Write a condition that eats a sandwich if it has neither tomato nor lettuce
 - `if ( !tomato && !lettuce) { eat_sandwich(); }`
 - `if ( !(tomato || lettuce) ) { eat_sandwich(); }`
- DeMorgan's theorem:
 - `!a && !b` $\Leftrightarrow$ `!(a || b)`
 - `!a || !b` $\Leftrightarrow$ `!(a && b)`
- More details in EE 109 and CS 170

Recall: Scope

- Scope refers to the **lifetime** and **visibility** of a variable
 - Recall variables are just memory slots in the computer...eventually the program will reclaim those slots and the variables will "die".
 - How long are those slots allocated and reserved for your use (i.e. what is their lifetime)?
 - What parts of your program can access the variables
- In C/C++, a variable's scope is the curly braces { } it is declared within
- **Main Point:** A variable dies at the end of the { ... } it was declared in

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    cin >> i;

    if(i > 0){
        int temp = 2*i;
        cout << temp << endl;
    } // temp died here

    cout << temp << endl; // ERROR!
    f1();
    return 0;
} // i dies here

void f1()
{
    // is i visible here?
    cout << i << endl;
}
```

Declaring the Inductive Variable

- The initialization statement can be used to declare a control/inductive variable, but its scope is ONLY the for loop (even though it is not technically declared in the {..} of the for loop)
 - Just realize that variable will die at the end of the loop
- However, because it dies after the first loop you can use that same variable name in a subsequent loop

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    for(int i=0; i < n; i++){
        cout << 3*i << endl;
    } // i dies here

    // won't compile
    cout << i << endl;

    // okay to reuse i
    for(int i=0; i < n; i++){
        cout << 4*i << endl;
    } // reincarnated i dies again

    return 0;
} // n dies here
```

Nested Loops Example 1

- **Key idea:** Perform all iterations of the inner loop before starting the next iteration of the outer loop
 - Said another way: The inner loop executes completely for each single iteration of the outer loop
- Trace through the execution of this code and show what will be printed

```
int main()
{
    for(int i=0; i < 2; i++){
        for(int j=0; j < 3; j++){
            cout << i << " " << j << endl;
        }
    }
}
```

i
0
0
0
0
1
1
1
1
1

j
0
1
2
3
0
1
2
3

Understand Your Bodies

- When you write loops write a comment as to what the body of each loop means in an abstract sense
 - The body of the outer loop represents 1 game (and we repeat that over and over)
 - The body of the inner loop represents 1 turn (and we repeat turn after turn)

```
int main()
{
    int secret, guess;
    char again = 'y';

    while(again == 'y') {
        // A single game

        // Choose secret num. 0-19
        secret = rand() % 20;
        guess = -1;
        // inner loop
        while(guess != secret) {
            // A turn of the game
            cout << "Enter guess: ";
            cin >> guess;
        }
        cout << "Win!" << endl;
        cout << "Play again (y/n): ";
        cin >> again;
    }
    return 0;
}
```

1 game {

1 turn {

Computing e^x

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

int main()
{
    // Starter code: modify the lines below
    double x;
    cin >> x;
    double x_to_i = 1;
    int i_fact = 1;
    double e_x = 1;
    for(int i=1; i < 10; i++){
        x_to_i *= x;
        i_fact *= i;
        e_x += x_to_i / i_fact;
    }
    cout << e_x << endl;
    return 0;
}
```

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Comparison, Logical Operators, if statements, switch statements

MODULE 4: CONDITIONAL STRUCTURES

Comparison Operators

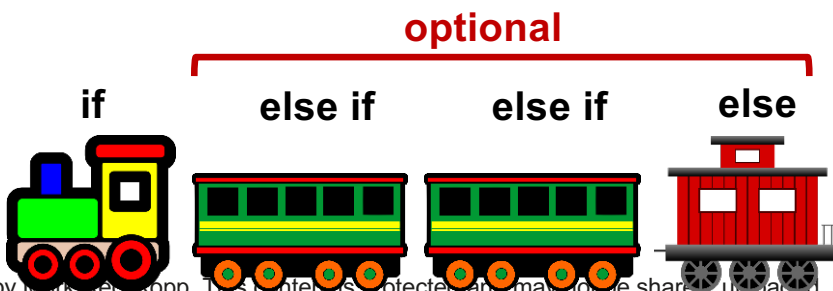
- To perform comparison of variables, constants, or expressions in C/C++ we can use the basic 6 comparison operators

Operator(s)	Meaning	Example
==	Equality	<code>if(x == y)</code>
!=	Inequality	<code>if(x != 7)</code>
<	Less-than	<code>if(x < 0)</code>
>	Greater-than	<code>if(y > x)</code>
<=	Less-than OR equal to	<code>if(x <= -3)</code>
>=	Greater-than OR equal to	<code>if(y >= 2)</code>

If...Else If...Else

- Use to execute only certain portions of code
- **else if** is *optional*
 - Can have any number of else if statements
- **else** is *optional*
- { ... } indicate code associated with the if, else if, else block

```
if (condition1)
{
    // executed if condition1 is true
}
else if (condition2)
{
    // executed if condition2 is true
    // but condition1 was false
}
else if (condition3)
{
    // executed if condition3 is true
    // but condition1 and condition2
    // were false
}
else
{
    // executed if neither condition
    // above is true
}
```



Mutually Exclusive Conditions

- What will each implementation print if 'grade' is 95?

```
if (grade >= 90)
{
    cout << "A range" << endl;
}
else if (grade >= 80)
{
    cout << "B range" << endl;
}
else if (grade >= 70)
{
    cout << "C range" << endl;
}
else if (grade >= 60)
{
    cout << "D range" << endl;
}
else
{
    cout << "Not gonna happen!" << endl;
}
```

```
if (grade >= 90)
{
    cout << "A range" << endl;
}
if (grade >= 80)
{
    cout << "B range" << endl;
}
if (grade >= 70)
{
    cout << "C range" << endl;
}
if (grade >= 60)
{
    cout << "D range" << endl;
}
else
{
    cout << "Not gonna happen!" << endl;
}
```

If...Else If...Else

- Guideline:
 - If various blocks of code are **mutually exclusive** then put them in an **if..else if..else** structure and not many individual **if..if..if..** statements

```
// BAD!
if (x < 0) {
    cout << "negative" << endl;
}
if (x >= 0) {
    cout << "positive" << endl;
}

// GOOD!
if (x < 0) {
    cout << "negative" << endl;
}
else {
    cout << "positive" << endl;
}
```

Logical Operators

- We can create compound conditions by using the logical AND, OR, and NOT operator

Operator(s)	Meaning	Example
&&	AND	<code>if((x==0) && (y==0))</code>
 	OR	<code>if((x < 0) (y < 0))</code>
!	NOT	<code>if(!x)</code>

Logical AND, OR, NOT

- The following tables show how the logical operations are evaluated under any set of values
- AND:
 - All inputs must be true for resulting expression to be true
 - If even one is false, the condition fails (false)
- OR:
 - If any input is true the condition evaluates to true

A	B	AND
False	False	False
False	True	False
True	False	False
True	True	True

A	B	OR
False	False	False
False	True	True
True	False	True
True	True	True

A	NOT
False	True
True	False

Exercise

- Which of the following is NOT a condition to check if the integer x is in the range $[-1$ to $5]$
 - a. $x \geq -1 \ \&\& \ x \leq 5$
 - b. $-1 \leq x \leq 5$
 - c. $!(x < -1 \ || \ x > 5)$
 - d. $x > -2 \ \&\& \ x < 6$

See solutions at end of slides

Conditions and DeMorgan's

- DeMorgan's theorem says there are always two ways to express a logic condition
- Write a condition that eats a sandwich if it has neither tomato nor lettuce
 - `if ( !tomato && !lettuce) { eat_sandwich(); }`
 - `if ( !(tomato || lettuce) ) { eat_sandwich(); }`
- DeMorgan's theorem:
 - `!a && !b` $\Leftrightarrow$ `!(a || b)`
 - `!a || !b` $\Leftrightarrow$ `!(a && b)`
- More details in EE 109 and CS 170

Timeout: In-Class Exercises

- nth

Common Mistakes 1

- Using assignment operator (=) rather than equality check operator (==)
 - If you accidentally use '=', it will convert the assigned value to a Boolean
 - Recall: The computer uses
 - 0 to mean **false**
 - Non-zero to mean **true**
- Using multiple if statements rather than if..else or if..else if statements
 - Two 'if' statements imply both could be true while 'if..else' implies only one

```
int main()
{
    int x, y;
    cin >> x >> y;

    // Wrong!
    if( x = 0 ) { /* some code */ }
    // Right!
    if( x == 0 ) { /* some code */ }

    // Wrong!
    if(x != y) { x = 5; }
    if(x == y) { y = 7; }

    // Right
    if(x != y) { x = 5; }
    else      { y = 7; }
    return 0;
}
```

When comparing with a constant, many companies and style guides recommend you flip the order to:

```
if( 0 == x ) { /* some code */ }
```

This, way the code won't compile if you accidentally write:

```
if( 0 = x ) // won't compile!
```



Common Mistakes 2

- All conditions must be formulated as a combination of comparisons of **two** values at a time
- Recall: The computer uses
 - 0 to mean **false**
 - Non-zero to mean **true**

```
int main()
{
    int x, y;
    cin >> x >> y;

    // Wrong!
    if( 0 <= x <= 9 )
        { /* some code */ }
    // Right!
    if( (0 <= x) && (x <= 9) )
        { /* some code */ }

    // Wrong!
    if( x == 0 || 1 )
        { /* some code */ }
    // Right!
    if( (x == 0) || (x == 1) )
        { /* some code */ }
    return 0;
}
```

Other Selection Structures

- C/C++ (and some other languages) provide alternative structures to if..else
 - `switch` (case) statement
 - Ternary operator (`cond ? x : y`)
- We will not require knowledge of these but simply recommend you briefly look over this material
 - Slides covering these structures are available at the end of the packet

while, do..while, and for Loops

MODULE 5: ITERATIVE STRUCTURES

Motivation for Loops

- Take a simple task such as outputting the first 1000 positive integers
 - We could write 1000 cout statements
 - Yikes! We could do it but it would be painful!
- Or we could use a loop

```
#include <iostream>
using namespace std;
int main()
{
    cout << 1 << endl;
    cout << 2 << endl;
    cout << 3 << endl;
    // hundreds more cout statements

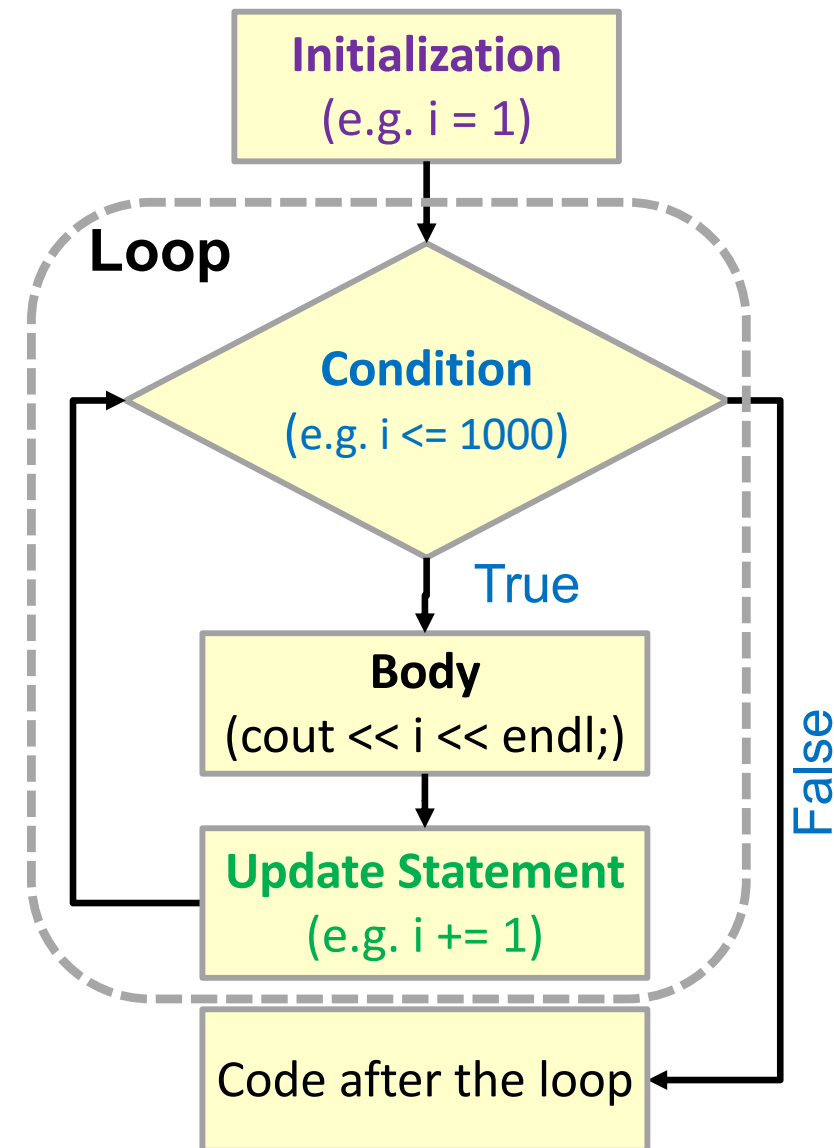
    cout << 999 << endl;
    cout << 1000 << endl;

    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1; i <= 1000; i+=1 )
    {
        cout << i << endl;
    }
    return 0;
}
```

4 Necessary Parts of a Loop

- Loops involve writing a task to be repeated
- Regardless of that task, there must be **4 parts** to make a loop work
- **Initialization**
 - Initialization of the variable(s) that will control how many iterations (repetitions) the loop will execute
- **Condition**
 - Condition to decide whether to repeat the task or stop the loop
- **Body**
 - Code to repeat for each iteration
- **Update**
 - Modify the variable(s) related to the condition



Types of Loops

- There are 2 (and a half) kinds of loops
- **while** (**do..while**) loops and **for** loops
 - Let's look at the syntax of each

```
int i = 1;
while (i <= 1000)
{
    // repetitive task
    cout << i << endl;
    i++; // update
}
// following statements
```

4 parts:

- Initialization
- Condition
- Body
- Update

```
          T   T   F
          1   2   5   8   4   7
for (int i = 1; i <= 1000; i++)
{
    3 6
    cout << i << endl;
}
9 // following statements
```

There is a variant of the while loop which is the **do..while** loop which we'll cover later.

Which Kind of Loop

- Use a while loop:
 - When you **DON'T** know how many times to iterate before the loop starts.
 - How many guesses will the user need before they get it right?
 - When you use "**until**" (see next slide)
- Use a for loop:
 - When you **DO** know the number of times to iterate in **BEFORE** you start the loop.

```
#include <iostream>
using namespace std;
int main()
{
    int guess;

    int secretNum = /* some code */
    cin >> guess;
    while(guess != secretNum)
    {
        cout << "Enter guess: " << endl;
        cin >> guess;
    }

    cout << "You got it!" << endl;
    return 0;
}
```

"Until" and "While not"

- Whenever you see or use the word '**until**' in a description
- Important Tip:
 - "**until** *x*" = "**while not** *x*"
 - Saying "keep guessing **until** you are correct" is the same as "keep guessing **while** you are **not** correct"

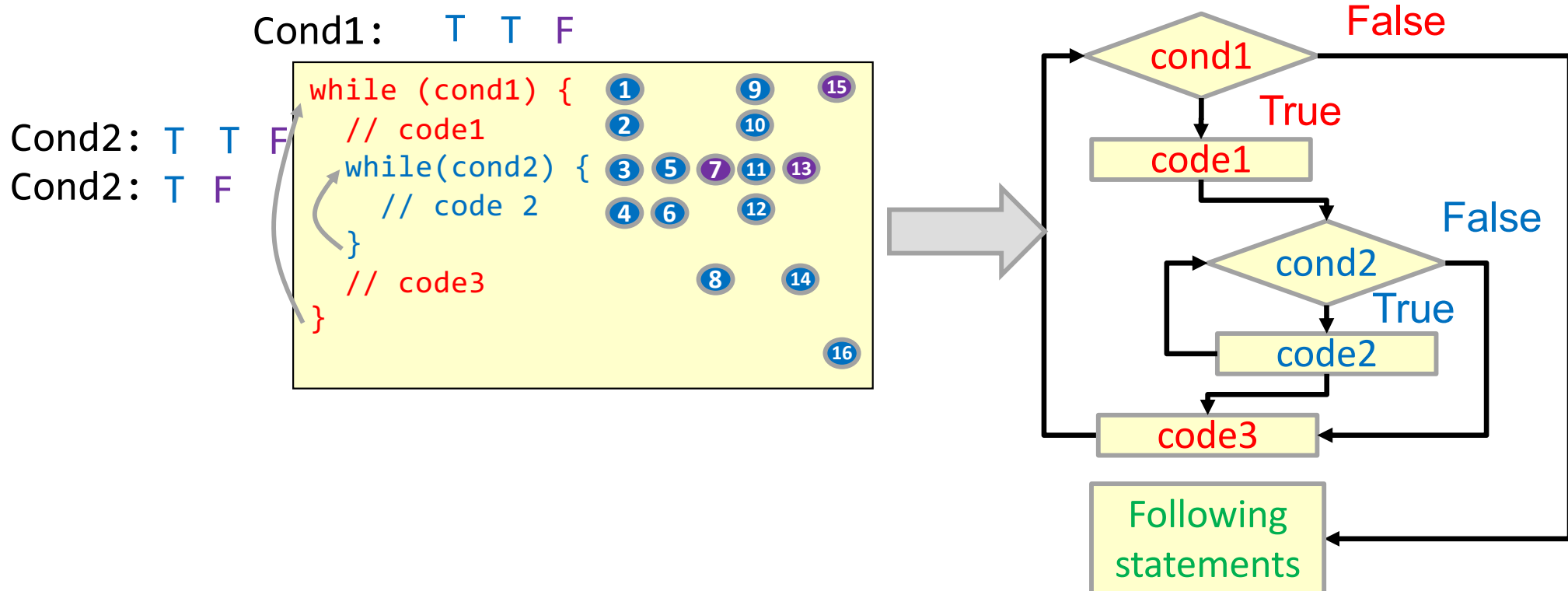
```
#include <iostream>
using namespace std;
int main()
{
    int guess;
    int secretNum = /* some code */
    cin >> guess;
    while(guess != secretNum)
    {
        cout << "Enter guess: " << endl;
        cin >> guess;
    }

    cout << "You got it!" << endl;
    return 0;
}
```

NESTED LOOPS

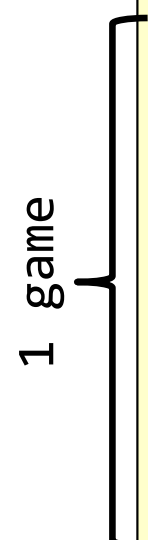
Nested Loop Sequencing

- Key Idea:** The inner loop runs in its entirety for each iteration of the outer loop



Nested Loops Example 1

- When you write loops consider what the body of each loop means in an abstract sense
 - The body of the outer loop represents 1 game (and we repeat that over and over)
 - The body of the inner loop represents 1 turn (and we repeat turn after turn)



```
int main()
{
    int secret, guess;
    char again = 'y';
    // outer loop
    while(again == 'y')
    { // Choose secret num. 0-19
        secret = rand() % 20;
        guess = -1;
        // inner loop
        while(guess != secret)
        {
            cout << "Enter guess: ";
            cin >> guess;
        }
        cout << "Win!" << endl;
        cout << "Play again (y/n): ";
        cin >> again;
    }
    return 0;
}
```

Nested Loops Example 2

- **Key idea:** Perform all iterations of the inner loop before starting the next iteration of the outer loop
 - Said another way: The inner loop executes completely for each single iteration of the outer loop
- Trace through the execution of this code and show what will be printed

```
int main()
{
    for(int i=0; i < 2; i++){
        for(int j=0; j < 3; j++){
            cout << i << " " << j << endl;
        }
    }
}
```

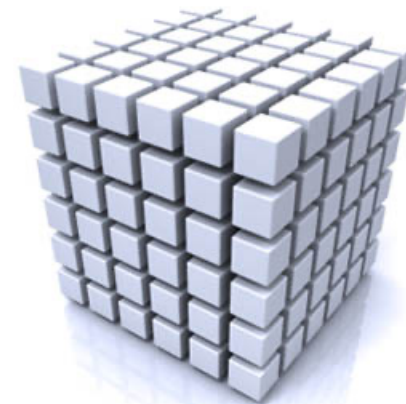
i
0
0
0
0
1
1
1
1
1

j
0
1
2
3
0
1
2
3

Tips

- Nested loops often help us represent and process multi-dimensional data
 - 2 loops allow us to process data that corresponds to 2 dimension (i.e. rows/columns)
 - 3 loops allow us to process data that corresponds to 3 dimensions (i.e. rows/columns/planes)

	0	1	2	3	4
0					
1					
2				2,3	
3					
4					



I/O Manipulators

- Manipulators control HOW `cout` handles certain output options and how `cin` interprets the input data (but print nothing themselves)
 - Must `#include <iomanip>`
- Common examples
 - `setw(n)`: Separate consecutive outputs by `n` spaces
 - `setprecision(n)`: Use `n` digits to display doubles (both the integral + decimal parts)
 - `fixed`: Uses the precision for only the digits after the decimal point
 - `boolalpha`: Show Booleans as `true` and `false` rather than 1 and 0, respectively
- Separated by `<<` or `>>` and used inline with actual data
- Other than `setw`, manipulators continue to apply to other output until changed

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

int main()
{
    double pi = 3.14159;

    cout << pi << endl;
    // Prints: 3.14159

    cout << setprecision(2) << fixed << pi << endl;
    // Prints: 3.14

    return 0;
}
```

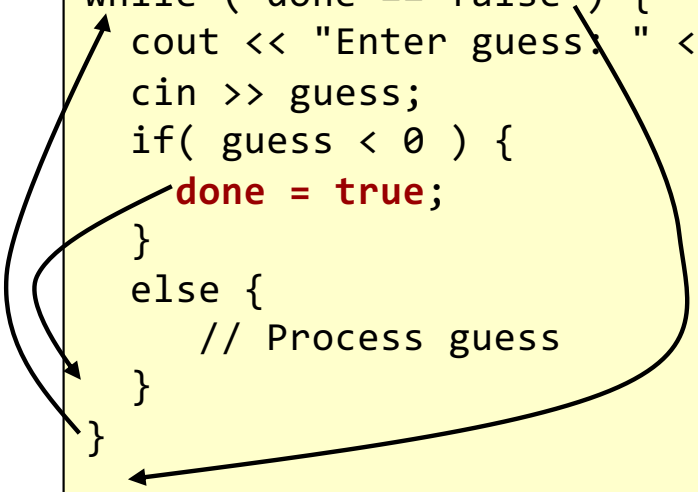
<http://en.cppreference.com/w/cpp/io/manip>

See "iomanip" in-class exercise to explore various options

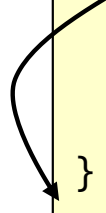
break statement

- **break**
 - Ends the current **loop** immediately and continues execution after its last statement
 - Only stops the INNER-MOST containing loop, not ALL nested loops.
- Consider two alternatives for stopping a loop if an invalid (negative) guess is entered

```
bool done = false;
while ( done == false ) {
    cout << "Enter guess: " << endl;
    cin >> guess;
    if( guess < 0 ) {
        done = true;
    }
    else {
        // Process guess
    }
}
```



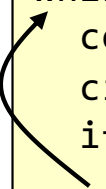
```
bool done = false;
while ( done == false ) {
    cout << "Enter guess: " << endl;
    cin >> guess;
    if( guess < 0 ) {
        break;
    }
    // Process guess
    // If guess < 0 we would skip this
}
```



continue statement

- `continue`
 - Ends the current **loop** [not **if** statement] immediately and continues execution after its last statement
- Consider two alternatives for repeating a loop to get a new guess if an invalid (negative) guess is entered
 - Often **continue** can be eliminated by changing the if condition

```
bool done = false;
while( done == false) {
    cout << "Enter guess: " << endl;
    cin >> guess;
    if(guess < 0){
        continue;
    }
    // Process guess (only here if guess>=0)
}
```



```
bool done = false;
while ( done == false ) {
    cout << "Enter guess: " << endl;
    cin >> guess;
    if( guess >= 0 ) {
        // Process Guess
    }
}
```



ODDS AND ENDS REGARDING C/C++ LOOPS

Recall: Scope

- Scope refers to the **lifetime** and **visibility** of a variable
 - Recall variables are just memory slots in the computer...eventually the program will reclaim those slots and the variables will "die".
 - How long are those slots allocated and reserved for your use (i.e. what is their lifetime)?
 - What parts of your program can access the variables
- In C/C++, a variable's scope is the curly braces { } it is declared within
- **Main Point:** A variable dies at the end of the { ... } it was declared in

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    cin >> i;

    if(i > 0){
        int temp = 2*i;
        cout << temp << endl;
    } // temp died here

    cout << temp << endl; // ERROR!
    f1();
    return 0;
} // i dies here

void f1()
{
    // is i visible here?
    cout << i << endl;
}
```

Declaring the Inductive Variable

- The initialization statement can be used to declare a control/inductive variable, but its scope is ONLY the for loop (even though it is not technically declared in the {..} of the for loop)
 - Just realize that variable will die at the end of the loop
- However, because it dies after the first loop you can use that same variable name in a subsequent loop

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    for(int i=0; i < n; i++){
        cout << 3*i << endl;
    } // i dies here

    // won't compile
    cout << i << endl;

    // okay to reuse i
    for(int i=0; i < n; i++){
        cout << 4*i << endl;
    } // reincarnated i dies again

    return 0;
} // n dies here
```

The Loops That Keep On Giving

- There's a problem with the loops below
- We all write "**infinite**" loops at one time or another
- **Infinite** loops never quit
- When you do write such a program, just type "**Ctrl-C**" at the terminal to halt the program

```
#include <iostream>
using namespace std;
int main()
{ int val;
  bool again = true;
  while(again = true){
    cout << "Enter an int or -1 to quit";
    cin >> val;
    if( val == -1 ) {
      again = false;
    }
  }
  return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
  int i=0;
  while( i < 10 ) {
    cout << i << endl;
    i + 1;
  }
  return 0;
}
```

The Loops That Keep On Giving

- There's a problem with the loop below
- We all write "infinite" loops at one time or another
- Infinite loops never quit
- When you do write such a program, just type "Ctrl-C" at the terminal to halt the program

```
#include <iostream>
using namespace std;
int main()
{ int val;
  bool again = true;
  while(again == true){
    cout << "Enter an int or -1 to quit";
    cin >> val;
    if( val == -1 ) {
      again = false;
    }
  }
  return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
  int i=0;
  while( i < 10 ) {
    cout << i << endl;
    i = i + 1;
  }
  return 0;
}
```

SOLUTIONS

Exercise

- Which of the following is NOT a condition to check if the integer x is in the range $[-1$ to $5]$
 - a. $x \geq -1 \ \&\& \ x \leq 5$
 - b. $-1 \leq x \leq 5$
 - c. $!(x < -1 \ || \ x > 5)$
 - d. $x > -2 \ \&\& \ x < 6$

See solutions at end of slides

OTHER SELECTION STRUCTURES

Switch (Study on own)

- Again used to execute only certain blocks of code
- *Cases must be a constant*
- Best used to select an action when an expression could be 1 of a set of constant values
- { ... } around entire set of cases and not individual case
- Computer will execute code until a break statement is encountered
 - Allows multiple cases to be combined
- Default statement is like an else statement

```
switch(expr) // expr must eval to an int
{
    case 0:
        // code executed when expr == 0
        break;
    case 1:
        // code executed when expr == 1
        break;
    case 2:
    case 3:
    case 4:
        // code executed when expr is
        // 2, 3, or 4
        break;
    default:
        // code executed when no other
        // case is executed
        break;
}
```

Switch (Study on own)

- What if a break is forgotten?
 - All code underneath will be executed until another break is encountered

```
switch(expr) // expr must eval to an int
{
    case 0:
        // code executed when expr == 0
        break;
    case 1:
        // code executed when expr == 1
        // what if break was commented
        // break;
    case 2:
    case 3:
    case 4:
        // code executed when expr is
        // 3, 4 or 5
        break;
    default:
        // code executed when no other
        // case is executed
        break;
}
```

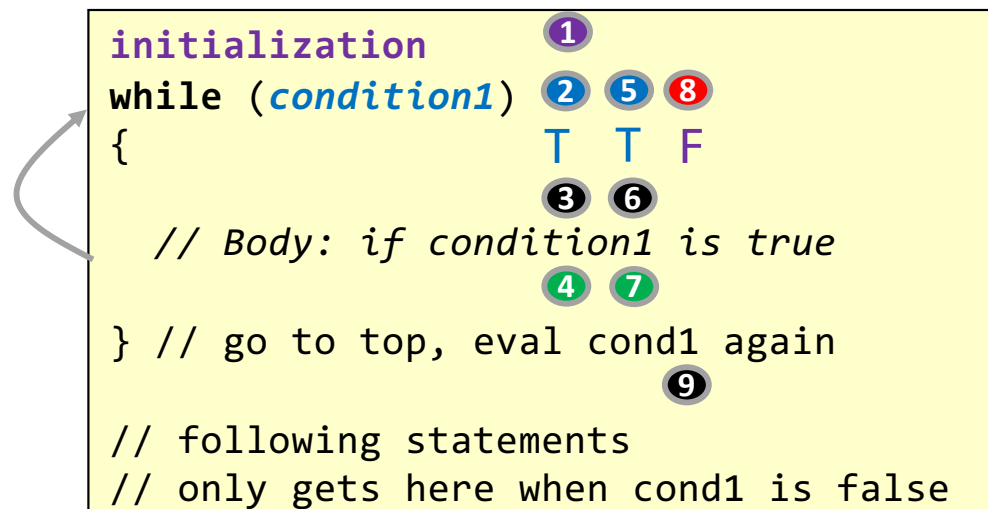
? Operator (Study on own)

- A simple if..else statement can be expressed with the ? operator
 - `int x = (y > z) ? 2 : 1;`
 - Same as:
`if(y > z) x = 2;`
`else x = 1;`
- Syntax: *(condition) ? expr_if_true : expr_if_false;*
- Meaning: the expression will result/return *expr_if_true* if *condition* evaluates to true or *expr_if_false* if *condition* evaluates to false

LOOP STRUCTURES

Type 1: while Loops

- A while loop is essentially a repeating 'if' statement

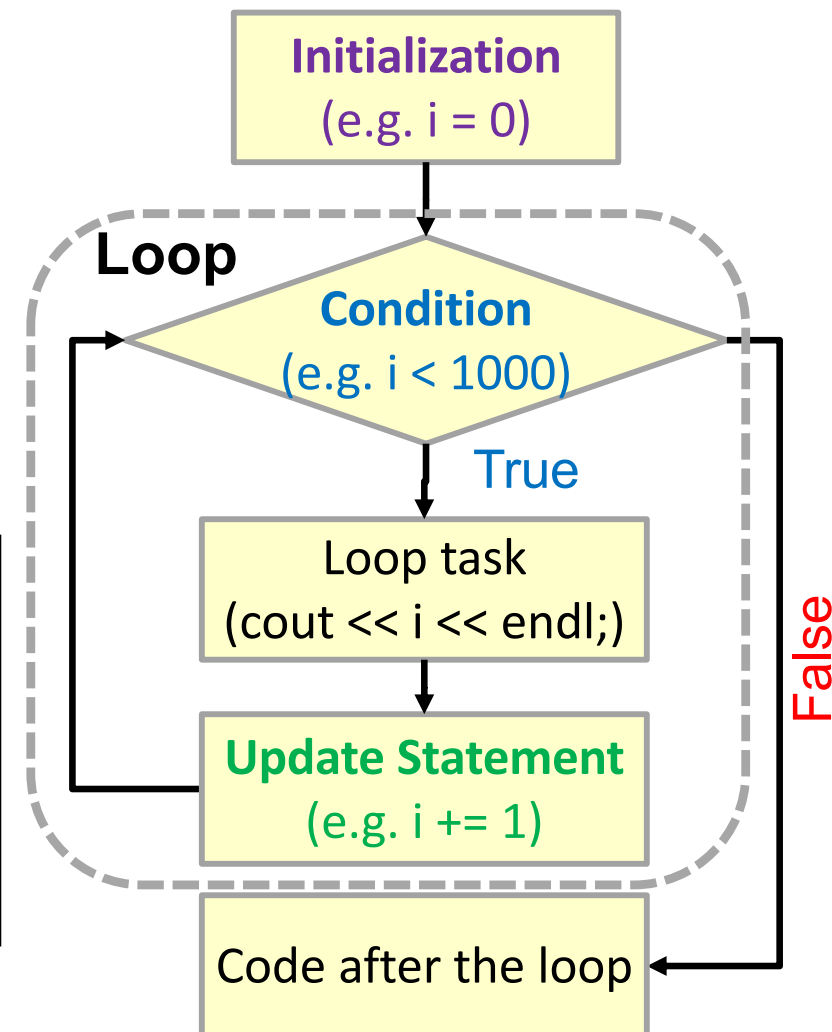


```

int i=0;
while (i < 1000)
{
    cout << i << endl;
    i++;
}

// following statements
    
```

While loop printing 0 to 999



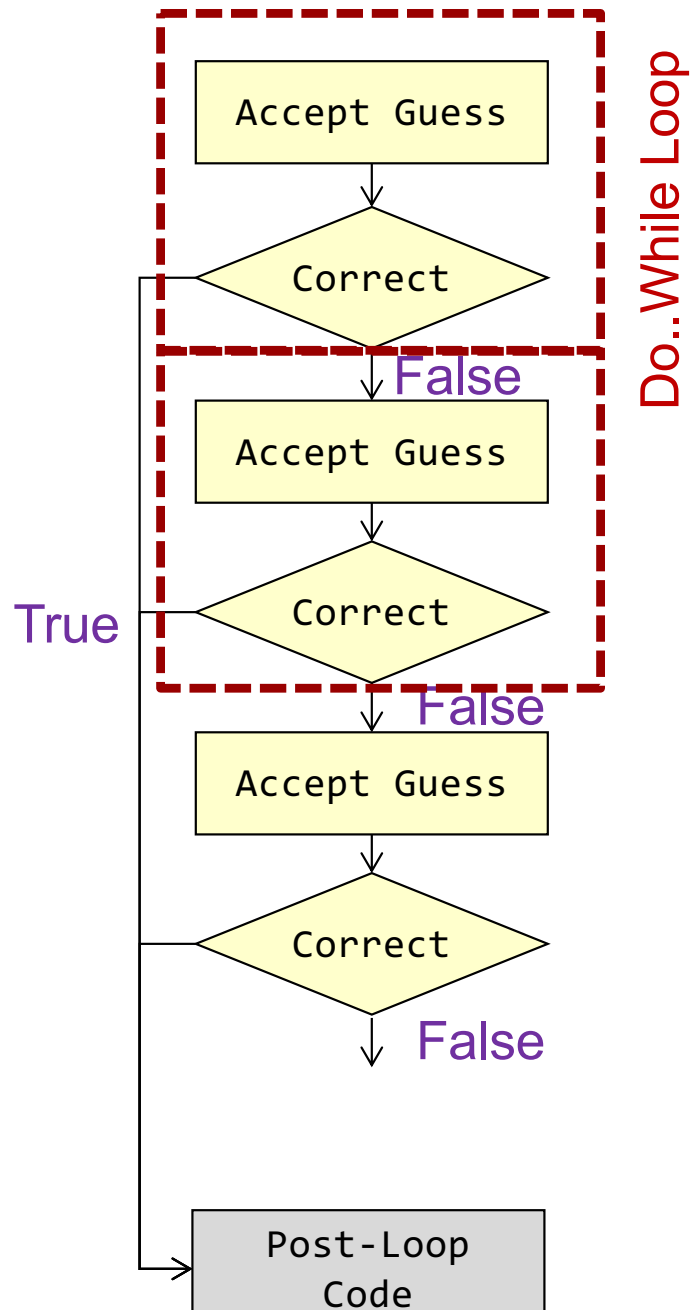
while vs. do..while Loops

- while loops have two variations: while and do..while
- while
 - Cond is evaluated first
 - Body only executed if condition is true (**maybe 0 times**)
- do..while
 - Body is executed **at least once**
 - Cond is evaluated
 - Body is repeated if cond is true

```
// While:
while(condition)
{
    // code to be repeated
    // (should update condition)
}

// Do while:
do {
    // code to be repeated
    // (should update condition)
} while(condition);
```

Using Flow Charts to Find Loops

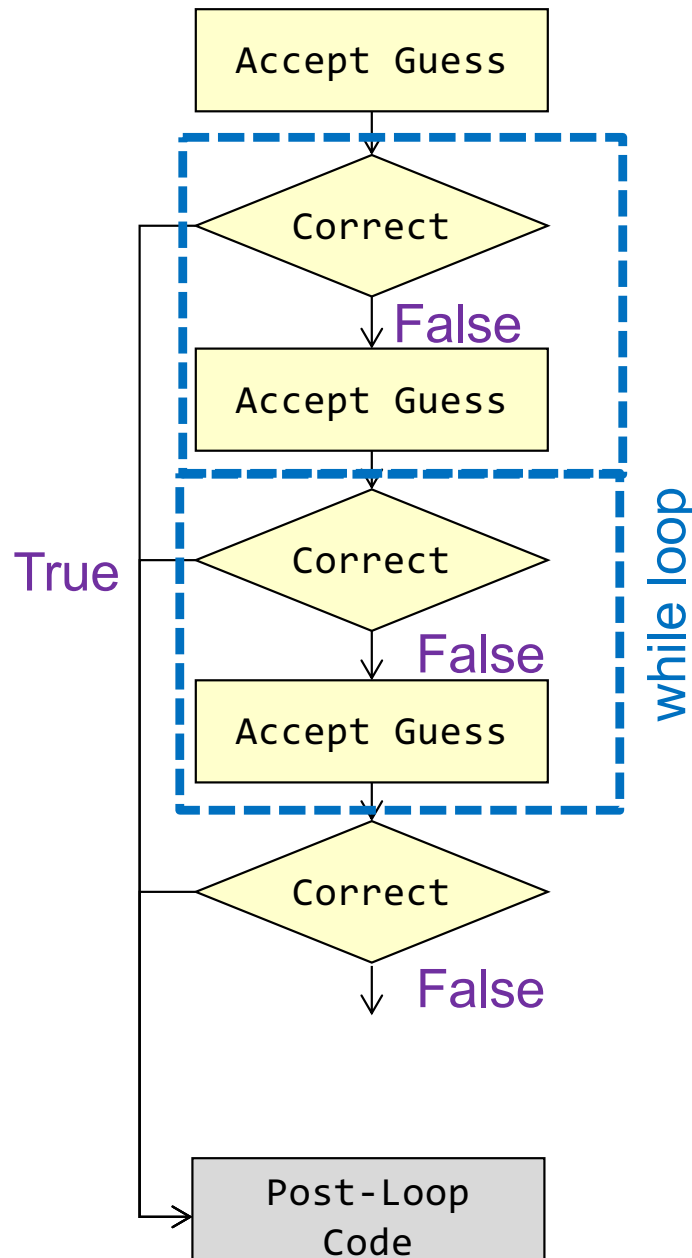


Draw out a flow chart of the desired sequence and look for the repetitive sequence

Here we check at the end to see if we should repeat...perfect for a **do..while** loop

```
do
  { accept_guess }
while ( ! correct )
```

Finding the 'while' Structure



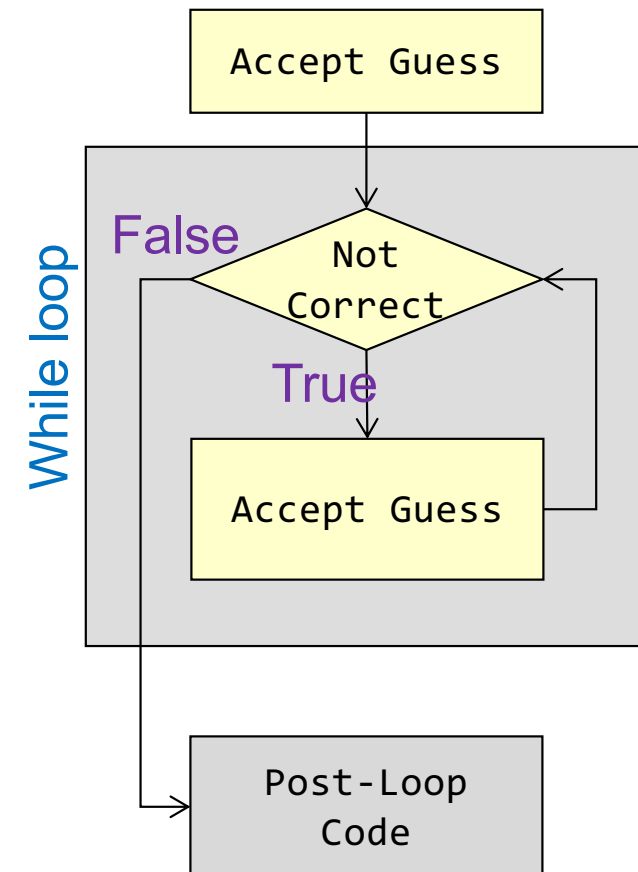
Draw out a flow chart of the desired sequence and look for the repetitive sequence

Here we check at the end to see if we should repeat...perfect for a **do..while** loop

```
do
{ accept_guess }
while ( ! correct )
```

But a while loop checks at the beginning of the loop, so we must accept one guess before starting:

```
accept_guess
while( ! correct )
{ accept_guess }
```



Hand Tracing (1)

- For the first program, trace through the code and show all changes to i for:
 - $n = 2$;
- For the second program, trace through the code and show the output for:
 - $t = \text{PI}/2, T = 2*\text{PI}$

```
int main()
{
    int n;
    cin >> n;
    for(int i = -n; i <= n; i++)
    {
        cout << i << endl;
    }
    return 0;
}
```

```
int main()
{
    double t, T;
    cin >> t >> T;
    for( double th = 0 ; th < T; th += t)
    {
        cout << sin(th) << endl;
    }
    return 0;
}
```

Hand Tracing (2)

- For the first program, trace through the code and show all changes to *i* and *y* for:
 - *x* = 10
 - *y* = 2
- For the second program, trace through the code and show all changes to *i* and *y* for:
 - *x* = 4
 - *y* = 11

```
int main()
{
    int x, y;
    cin >> x >> y;
    for(int i=1; i <= x; i=i+y)
    {
        cout << i << endl;
        y++;
    }
    return 0;
}
```

```
int main()
{
    int x, y;
    cin >> x >> y;
    for( ; x < y; x++)
    {
        cout << x << " " << y << endl;
        y--;
    }
    return 0;
}
```

bools, ints, and Conditions

- Loops & conditional statements require a **condition** to be evaluated resulting in a **true** or **false** result.
- In C/C++...
 - 0 means **false** / Non-Zero means **true**
 - **bool** type available in C++ => '**true**' and '**false**' keywords can be used but internally
 - **true** = non-zero (usually 1) and
 - **false** = 0
- Any place a condition would be used, a bool or int type can be used and will be interpreted as bool

```
int x = 100;  
while(x)  
{ x--; }
```

```
bool done = false;  
while( ! done )  
{ cin >> done; }
```

```
int x=100, y=3, z=0;  
if( !x || (y && !z) )  
{ /* code */ }
```

Single Statement Bodies

- **The Rule:** Place code for an if, else if, or else construct in curly braces { ... }
- **The Exception:**
 - An if or else construct with a **single statement body** does not require { ... }
 - Another if counts as a single statement
- However, you should ALWAYS prefer { ... } even in single statement bodies so that editing later does not introduce bugs

```
if (x == 5)
    y += 2;
else
    y -= 3;
cout << "done1" << endl;

while (x != 0)
    x--;
cout << "done2" << endl;

for(int i=0; i < 10; i++)
    if( i % 2 == 0)
        cout << i << endl;
cout << "done3" << endl;
```

Solutions 1

```
int main()
{
    int n;
    cin >> n;
    for(int i = -n; i <= n; i++)
    {
        cout << i << endl;
    }
    return 0;
}
```

Program Output for input of 2:

```
-2
-1
0
1
2
```

```
int main()
{
    double t, T;
    cin >> t >> T;
    for( double th = 0 ; th < T; th += t)
    {
        cout << sin(th) << endl;
    }
    return 0;
}
```

Program Output for input $\pi/2$ and 2π :

```
0
1
0
-1
```

Solutions 2

```
int main()
{
    int x, y;
    cin >> x >> y;
    for(int i=1; i <= x; i=i+y)
    {
        cout << i << endl;
        y++;
    }
    return 0;
}
```

Program Output for input of **10 2**:

```
1
4
8
```

```
int main()
{
    int x, y;
    cin >> x >> y;
    for( ; x < y; x++)
    {
        cout << x << " " << y << endl;
        y--;
    }
    return 0;
}
```

Program Output for input **4 11**:

```
4 11
5 10
6 9
7 8
```