Control Flow and Vectors

PIC 10A, UCLA

Control Flow and Vectors

Being able to make a computer do a task repeatedly, while certain conditions are upheld, or yielding different performances depending on certain conditions is known as control flow. With this understanding, it is possible to unlock the power of vectors (and arrays), that can store vast amounts of data in an easily accessible format.

Basic Logic

In C++, many statements can be viewed as conditions: **true** or **false**.

There are a number of **boolean operators** to compare two values and force a result of **true** or **false**.

- **==** (equal to): 3 == 4 is false, 9.1 == 9.1 is true, etc.
- **!=** (not equal to): 3 != 4 is true, 9.1 != 9.1 is false, etc.
- **<** (less than): std::string("crocs") < std::string("shoes") is true, 3 < 3 is false, etc.
- **<=** (less than or equal to): 3 <= 3 is true, 0 <= 7 is true, etc.
- **>** (greater than): 5 > 7 is false, 11 > 7 is true, etc.
- **>=** (greater than or equal to): 0 >= 0 is true, 'A' >= 'B' is false, etc.

In addition, there is a logical negation operator **!** that negates the truth of a statement: !(3== 3) is false, ! (7 > 100) is true, etc.

Basic Logic

Sometimes we want the truth of a statement to be conditional upon two or more statements being true. We use the **and operator &&** for this. For a statements joined by **&&**, the overall result is true only if all the statements are true; otherwise it is false.

For readability and to avoid subtle compiler errors, it is beneficial to wrap a condition in parentheses.

(1 < 2) && (2 < 3) is true: both statements are true
(1 < 2) && (8 > 100) && (6 <= 6) is false: the middle statement is false.

Basic Logic

In parsing **and**s, the compiler performs the evaluation “lazily”: it evaluates each statement joined by **&&** from left to right. As long as they are **true**, it continues to check the next statement. If it finds any statement to be **false**, it gives an output of **false** and **all remaining statements are ignored and not evaluated** - it’s as though they are not even present.

Logical and is only true if all statements are true.

Basic Logic

Sometimes we want the truth of a statement to be conditional upon at least one statement being true. We use the **or operator** || for this. For a statements joined by ||, the overall result is false only if all the statements are false; otherwise it is true.

(1 < 2) || (8 > 100) || (1000 <= 6) is true: the first statement is true.
(1 > 2) || (5 > 7) is false: both statements are false.

Basic Logic

In parsing **or**s, the compiler performs the evaluation “lazily”: it evaluates each statement joined by || from left to right. As long as they are **false**, it continues to check the next statement. If it finds any statement to be **true**, it gives an output of **true** and **all remaining statements are ignored and not evaluated** - it’s as though they are not even present.

Logical or is only false if all statements are false.

Basic Logic

We can combine **and**s and **or**s, using parentheses for extra clarity:

```
int x = 13, y = 20;
( !(x < 0) ) && ( (x > y) || (y < 100) ); // true
```

Above the statement is true:

- (x<0) evaluates to **false**, so **!(x<0)** is **true**. This **true** must be **and**ed with another statement.
- (x > y) || (y < 100) evaluates to **true** because although (x > y) is **false**, (y < 100) is **true**.

Basic Logic

What we write mathematically may not have the same meaning in C++. Consider the statement:

```
-5 <= 125 <= 2;
```

In mathematics, $-5 \leq 125 \leq 2$ is false because although $-5 \leq 125$, 125 is not ≤ 2.

A compiler processes things differently... It processes statements left to right, and may make conversions between data types if necessary.

- **-5 <= 125** is evaluated first: the result is **true** (of type **bool**).
- **true <= 2** doesn't make sense because **true** is **bool** and **2** is **int**, so the **bool** gets promoted to an **int**.
- Recall that **true** corresponds to **1** and **false** to **0**, so the compiler considers the truthfulness of **1 <= 2**. This is **true**.
- The overall result is **true**!!!

Basic Logic

DeMorgan's Laws: are useful in converting between logically equivalent statements joined by **and**s and/or **or**s. Suppose that S_1, S_2, **...,** S_n are each statements with some truth value. Then

$$\overbrace{!(S_1 \&\& S_2 \&\& S_3 \&\& \ldots \&\& S_n)}^{\textbf{not all of them are true}} \overbrace{==}^{\textbf{same as}} \overbrace{(!S_1)||(!S_2)||(!S_3)||\ldots||(!S_n)}^{\textbf{at least one is false}}$$

and

$$\overbrace{!(S_1||S_2||S_3||\ldots||S_n)}^{\textbf{not a single one is true}} \overbrace{==}^{\textbf{same as}} \overbrace{(!S_1)\&\&(!S_2)\&\&(!S_3)\&\&\ldots\&\&(!S_n)}^{\textbf{every one is false}}$$

Basic Logic

Some explanation on the colloquial meanings of the statements:

- “not all of them are true”: because we negated the fact that all are true.
- “at least one is false”: because we have **or**ed a collection of negations.
- “not a single one is true”: because we negated the fact that one or more statements are true.
- “every one is false”: because we **and**ed a collection of negations.

If

An **if statement** allows us to execute instructions conditional upon an input **bool** value being **true.**

```
const double speedLimit = 60; // mph
double speed = 0; // mph

cout << "Enter speed in mph: "; // ask for speed
cin >> speed; // read in the vehicle speed

// check if the vehicle speed exceeds the speed limit
if( speed > speedLimit ) { // and if so, print message
    cout << "The car is speeding!"<< endl;
    cout << "Give a hefty fine!";
}
```

If

In general, the syntax we use for an **if statement** is

```
if ( something ) {
   do these statements;
   there could be more than one;
}
```

something must be a **bool** value or an expression that can be converted to a **bool**.

The statements within the **{ ... }** are only executed provided **something** is **true**.

We call the content within the braces, the **body** of the if statement.

If

It is very important to note confuse **=** and **==**.

```
int x = 14;

if ( (x = 3) || (x==6) ) {
    cout <<x << "is 3 or 6!";
}
```

This outputs:

```
3 is 3 or 6!
```

If

The assignment operator returns a value (actually a reference).

- First, the expression **x=3** is performed: this sets **x** to the value of **3**;
- second, a **reference to x**, a value of type **int&** is returned;
- third, the **if** requires a **bool** value and the reference to x must be converted to a **bool**...
- **3** gets converted to **true**, so the next statement is ignored and the body is executed.

In general **0** will be converted to **false** and all nonzero values will be converted to **true**.

if... Else

We can also structure code so that we check a condition with **if** and then if the condition is false, all remaining possibilities are managed by an **else** block.

```
int x = 0; cout << "Enter a value for x: ";
cin >> x; // read in a new value

int absX = 0; // the absolute value of x

if ( x >= 0) { // if x is greater than or equal to 0
    absX = x; // then its absolute value is itself
}
else { // otherwise if x is negative
    absX = -x; // then its absolute value is negative x to make it positive
}
```

if... Else

Recall that the absolute value of a number x, mathematically is written as $|x|$ is its magnitude, a number always bigger than or equal to 0:
$|4| = 4, |0| = 0, |-11| = 11$, etc.

If the value of x is greater than or equal to 0, the absolute value is x and this is what the **if** condition checks.

If the value of x is not greater than or equal to 0 (hence negative), the absolute value must be the negated value of x, managed by the **else** part.

if... Else

Some remarks on **if... else**:

An **else** cannot exist in isolation; it must always be paired with a preceding **if**.

```
else { // ERROR: else what?
    cout << 5;
}
```

if... Else

When only a single statement follows an **if** or **else**, a set of braces is not required, but only that one single statement is part of the preceding **if** or **else**. Any subsequent statements are not part of the control structure.

The code below:

```
if ( true )
    cout <<"true!";
else
    cout <<"false!";
```

is equivalent to:

```
if ( true ) {
    cout << "true!";
}
else {
    cout <<"false!";
}
```

if... Else

The braces help to group the code and can make it more robust should the code be modified. Consider the snippet

```
// if the employee's productivity is excellent
if( productivity >= excellenceThreshold )
    wage *= raise; // give them a raise
```

and suppose someone else tried to add a display message to the process:

```
// if the employee's productivity is excellent
if( productivity >= excellenceThreshold )
    cout << "Giving raise...";
wage *= raise; /* PROBLEM: gives all employees a raise: this line is
    always read; it does not belong to an if */
```

Now the code compiles and runs, but whether the message is printed or not, **every employee** gets a raise, even without a high productivity.

if... Else

This could be fixed by adding braces for more robust coding.

```
// if the employee's productivity is excellent
if( productivity >= excellenceThreshold ) {
    wage *= raise; // give them a raise
}
```

The code below can also have the correct operation:

```
// if the employee's productivity is excellent
if( productivity >= excellenceThreshold ) {
    cout << "Giving raise...";
    wage *= raise; // give them a raise
}
```

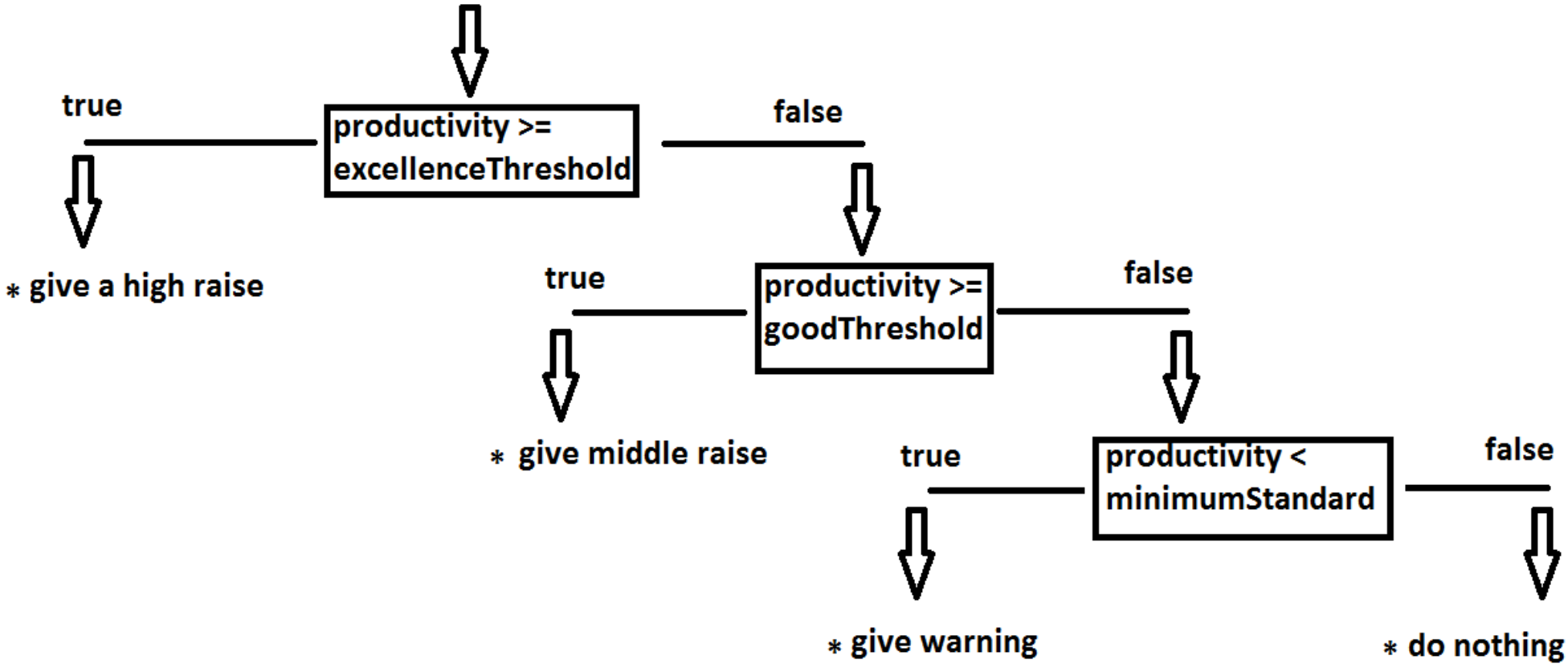
If... Else

Sometimes, we want to consider multiple cases and not just two branches:

```
if ( productivity >= excellenceTheshold ) { // if excellent
    wage *= highRaise; // give the highest raise
}
else if ( productivity >= goodThreshold ) { // if not excellent but good
    wage *= middleRaise; // give mid-range raise
}
// otherwise, if they below minimum standard
else if ( productivity < minimumStandard ) {
    cout <<"Warning: employee is under-performing"; // print warning
}
```

If... Else

Drawing a diagram can be helpful, too.



If... Else

The above branches should be understood as each **else if** being an **if** nested inside of an **else** block.

```
if ( productivity >= excellenceTheshold ) { // if excellent
    wage *= highRaise; // give the highest raise
}
else { // if not excellent
    if ( productivity >= goodThreshold ) { // but good
        wage *= middleRaise; // give mid-range raise
    }
    else { // if not excellent or good
        if ( productivity < minimumStandard ) { // if under-performing
            cout <<"Warning: employee is under-performing"; // print warning
        }
}
```

If... Else

Note that **only one** branch of an **if, else if, else if, ...** structure will be executed.

- The first condition is checked. If the condition is true, its body is executed and the remaining cases are not considered, even if they are true.
- If the condition is false, the second condition is checked.
  - If the second condition is true, its body is executed and the remaining cases are not considered, even if they are true.
  - If the second condition is false, the third condition is checked.
    - If ...
    - ...

If... Else

We are allowed to have **empty bodies {}** or **empty statements ;** meaning that nothing is to be done in a given case. The two snippets below are equivalent.

```
// check if x is greater than 7
if ( x > 7 ) { } // and if so, do nothing
else {
    cout << "x is not greater than 7.";
}

// check if x is greater than 7
if ( x > 7 )
    ; // do nothing: this is the empty statement
else {
    cout << "x is not greater than 7.";
}
```

If... Else

Warning: one should be cautious about using comparisons such as equality between floating points in control flow.

Due to roundoff error, it's hard to truly judge when two floating point numbers are equal.

Scopes

A set of braces { ... } defines a **scope**.

Within that scope, variables can be created, even masking the names of other variables, and after the closing braces, those variables no longer exist!

```
{
    int x = 42; // defined a variable x inside of this scope...
} // and now x no longer exists!

int x = 42;
if ( x > 0 ) {
    int y = x+1;
}

cout << y; // ERROR: y no longer exists!
```

Scopes

The following is an example of **masking** and such practice should be avoided due to the confusion it causes!

```
int x = 42;

if ( true ) {
    int x = 20; // this impostor x masks the other x
    cout << x << end;
} // the impostor x dies here

cout <<x; // and we can print the "real" x...
```

```
20
42
```

We should have just picked a different variable name for the inner x to avoid confusion.

While Loops

The **if... else** structure only allows us to decide how to proceed for a given condition; it does not directly tell us how many times to do something. To execute statements many times, we need to use **loops**.

Let's say we wanted to accept numbers from a user until they run out of numbers to supply, and then tell them the total of the numbers.

We can use a **while** loop: a **while** loop is a control flow structure that checks if a given condition is upheld (**true**) and as long as the condition is upheld, it repeatedly executes the statements within its body.

While Loops I

```
double sum = 0; // to store the sum of all of the numbers

bool more = true; // indicates if the user has more numbers to supply

while ( more ) { // while the user still has numbers to supply
    cout << "Enter a number: ";

    double input = 0; // define input variable and collect it
    cin >> input;

    sum += input; // add the input to the cumulative sum

    // check if the user has more numbers
    cout << "More numbers? y/n: ";
```

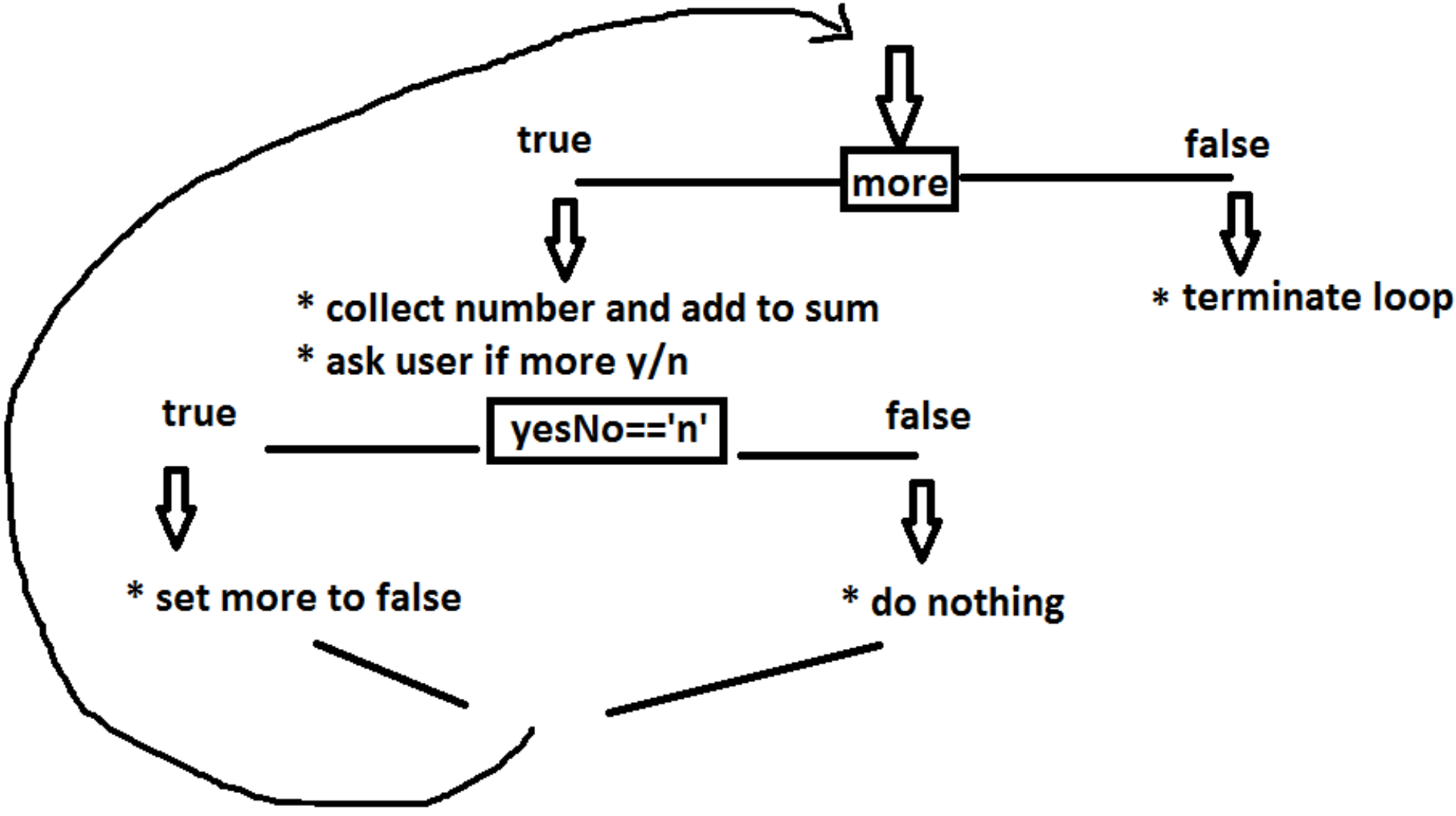
While Loops II

```
    char yesNo = ' '; // define user choice and collect it
    cin >> yesNo;

    if ( yesNo == 'n') { // if they chose no
        more = false; // no more inputs need collecting
    }
}

// announce the grand total
cout << "The grand total is: "<< sum;
```

While Loops


true
more
false
* collect number and add to sum
* ask user if more y/n
* terminate loop
true
yesNo=='n'
false
* set more to false
* do nothing

While Loops

Within the example, **input** and **yesNo** are local variables to the scope of the loop. They are created anew each time the loop body executes and they do not exist after the loop is over. Think of it like:

```
{
    // ...
    double input = 0;
    // ...
    char yesNo = ' ';
}

{
    // ...
    double input = 0;
    // ...
    char yesNo = ' ';
}
// ...
```

While Loops

The general syntax for a **while loop** is:

while (condition) {
*   statement(s);*
}

where, as for the **if** and **else**, if the **while** body consists of only a single statement the braces are not required, but do make for more robust coding.

While Loops

It is important to choose an appropriate test condition so that the loop does terminate at some point. Otherwise we wind up with an **infinite loop**:

```
int i = 0;

while ( (2 * i - 1 ) % 2 != 0 ) {
    cout << "This is the loop that never ends. "<<
        "Yes, it goes on and on, my friends."<<
        "Someone started running it, not knowing what it was, "<<
        "and it will continue running forever just because..."<<endl;

    i+=2;
}
```

While Loops

Tracing through the infinite loop:

- Initially **i** is **0**.
- **(2*i-1) % 2** is **-1 % 2**, which is **-1**. Since **-1** is nonzero, this gets converted to **true** and the loop is entered.
- The message is displayed and **i** is then incremented by **2** to make it **2**.
- **(2*i-1) % 2** is **3** and **3 % 2** is **1**, which is converted to **true** and the loop is entered again.
- The message is displayed and **i** is then incremented by **2** to make it **4**...

At no point will **(2*i - 1) % 2** ever be **0** because **2*i - 1** will always be odd. The loop will repeat forever (or until the program crashes).

Commenting Control Flow Structures

In addition to commenting the main ideas and actions taken in “normal code”, to make work with control flow structures easier to follow, there should generally be a comment for each branch of an if/else and each condition/range for a loop.

Someone reading the code should be able to read what cases are being considered in the comments and then see how they are managed, in the comments!

Coding and Debugging Strategies

It's important to be able to fix code when it is not working: due to compiler/linker errors or because it is not generating the correct output.

We'll cover some strategies here to fix broken code.

- **Unit testing:** check individual parts of code to ensure they have the correct behaviour. This involves knowing some correct outputs for given inputs.

  For example, if a line of code is supposed to collect a user's full name, we should try a few different names, maybe names that include spaces between the first and last names, etc. Provided these work, it's likely that component of code is good. Or, if we know the way a given loop is supposed to behave given a set of user inputs, we should ensure it does exactly that.

  Often, we should try to "break" our code to ensure even the edge cases are handled well.

Coding and Debugging Strategies

- **Pseudo-code**: writing the essential logic down, without being precise about syntax can be immensely beneficial and can easily be converted into real code. For example:

  while the account balance is < target balance
  - add the interest
  - increase the count of the number of years

- **Diagrams:** drawing a picture of what should be in a container, or what variable references another, etc. can help with unit testing and debugging.

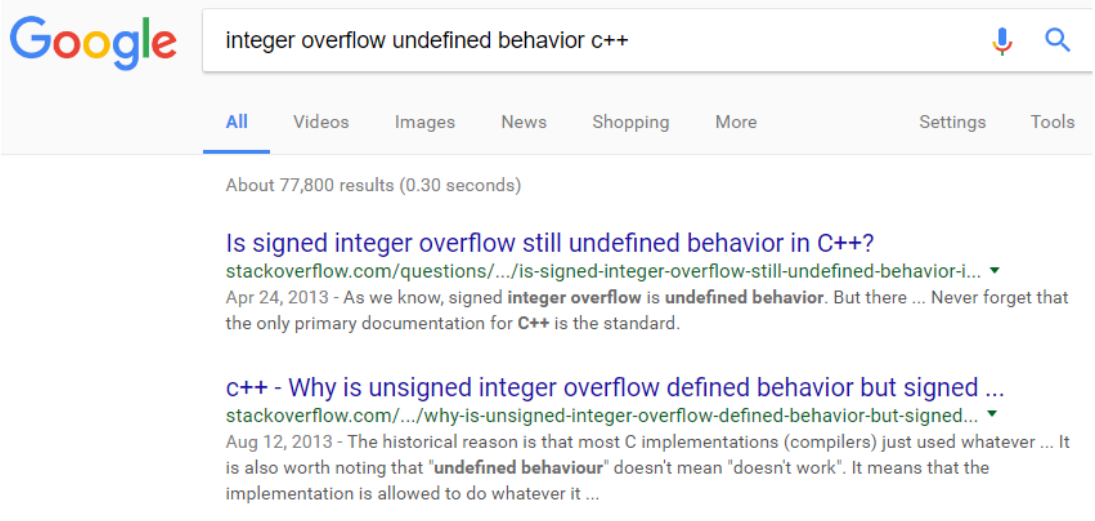
  For example, if there are indexing issues with a **std::string** and we know what should be in the variable, a picture may show us that our index is off by one, etc.

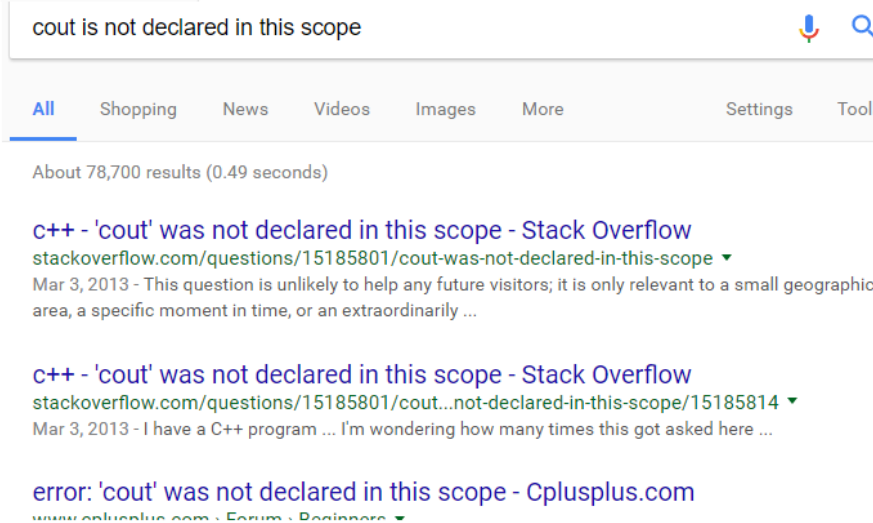
Coding and Debugging Strategies

- **Online resources:** there are many online references for C++ (and programming in general), and helpful forums such as **Stack Overflow** that store a wide database of questions/answers and allow users to ask questions.

  Living in the 21st century, this should be one of the first things to try.

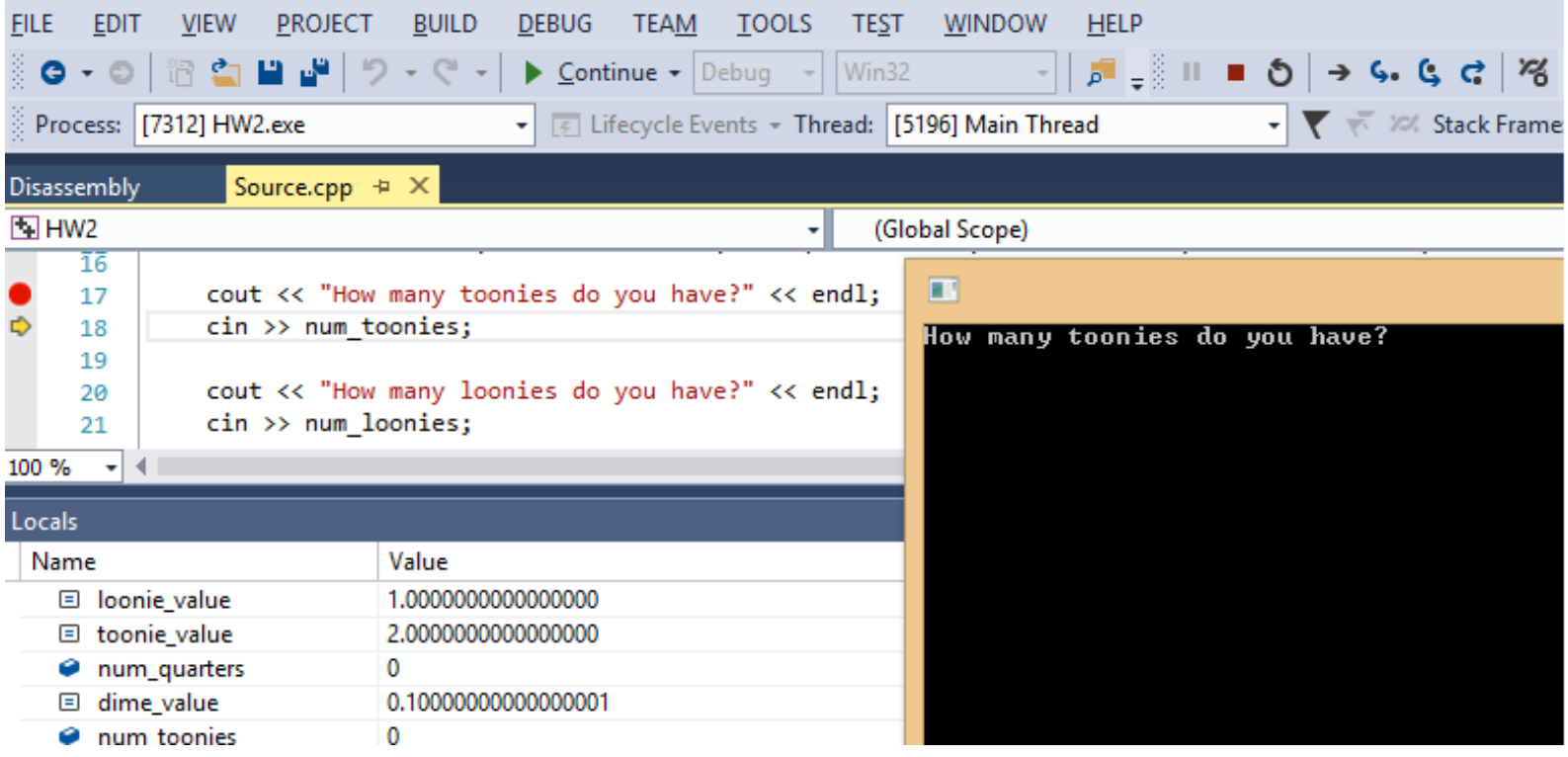
  Compiler error? **Google the error message!**
  Conceptual problem? **Google the question and add "C++"**





Coding and Debugging Strategies

- **Tracing through the code** by hand or with a debugger: it can be a useful exercise to trace the values of variables as we proceed through control flow structures to check beginning and ending conditions and to determine if our logic is correct. Many IDEs also include an ability to use a **debugger** to step through the code line-by-line.



For Loops

Sometimes we know exactly how many times a sequence of statements should execute and a **for loop** is a convenient structure for this. A for loop can be used in other contexts, too, but one of the most natural uses is in doing steps a fixed number of times.

Let's say we wanted to print all the integers from 1 to 100 separated by new lines. We could write:

```
// create index value i and take it from 1 to 100
for ( int i = 1; i <= 100; ++i ) {
    cout << i <<endl; // and print i each time with a new line
}
```

For Loops

The general syntax for a **for loop** is:

for (initialization statement; condition to check for; statement to perform after body of statements) {
body of statements;
}

- First, we can define/initialize a local variable: this variable does not exist outside of the loop.
- Second, we check that the condition is **true**; if true, we enter the body; if false, we skip over the loop.
- Third, we execute all the statements in the loop.
- Fourth, we perform the statement to be done after the body.
- And we go back to the second step...

Note: no semi-colon appears after the statement preceding the closing parenthesis).

For Loops

The **factorial function**, denoted $n!$ for an integer $n \geq 0$, is defined by

$$n! = n \times (n-1) \times (n-2) \times .... \times 3 \times 2 \times 1$$

if $n > 1$, with $0! = 1! = 1$ by definition.

For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$, etc.

We can compute the factorial of an **unsigned long long int** value **n** by:

```
unsigned long long int n_factoral = 1; // start product at 1

// for each integer from 1 to n
for ( unsigned long long int i = 1; i <= n; ++ i) {
    n_factorial *= i; // multiply them
}

cout << n << "! ="<< n_factorial;
```

For Loops

We can count how many vowels there are in a **std::string** of lowercase letters called **message**:

```
size_t numberOfVowels = 0;

for ( size_t i = 0; i < message.size(); ++i ) { for each index i of the string
    char c = message[i]; // look at the ith char

    // check if the character is a, e, i, o, or u
    if ( (c == 'a') || ( c == 'e') || ( c == 'i') || (c == 'o') || ( c = 'u') ) {
        ++numberOfVowels; // increase count
    }
}
```

For Loops

Notice that our index began at **0**, the first index of the **string** and can never reach **message.size()**, which would be an invalid index. This is guaranteed by requiring that **i < message.size()**.

We used **size_t** as a variable type. Just a friendly reminder that **size_t** is to be used for indexing and container sizes.

For Loops

The variables defined in the parentheses **(...)** of a for loop stay alive for the duration of the loop: they are not re-created each time the loop starts again.

The variables defined inside the body of the loop **{ ... }** are destroyed after each run of the loop before the next run:

```
for (int i=1; i <= 7; ++i) {
    int j = 42; // okay to do this: won't be "redefined" because
} // j is destroyed before i gets incremented
```

For Loops

Each statement within a **for** loop is optional: it can be replaced by an **empty** statement. The initialization statement can also be a reassignment:

```
double x = 2.2;
int k = 1;
for (k=2; x < 17; ) { // while x < 17
    x *= k; // multiply it by k, which is 2
}
```

Nesting

Loops and if/else structures can be nested inside of each other: loops can appear within loops, if/else within a loop, loops within if/else, if else within if/else, etc.

Suppose we wanted to print a pattern such as:

```
*
**
***
****
*****
```

where there are n rows, and for a row number i, there are i number of *'s that appear: 1, 2, 3, 4, 5, ...

Nesting

We could write code such as:

```
// the number of rows to print: can be changed
int numberOfRows = 2;

// for each row from 1 to numberOfRows
for ( int rowNumber = 1; rowNumber <= numberOfRows; ++rowNumber )
{

   // go through a count of rowNumber
   for ( int count = 1; count <= rowNumber; ++ count ) {
      cout << '*'; // and print an asterisk each time
   }

   cout << endl; // add a new line after each row is printed
}
```

Nesting

We have **numberOfRows == 2** and we begin the loops.

- Initially **rowNumber == 1**, which is **<= numberOfRows** and we then
  - Start **count** at **1**, which is **<= rowNumber**, 1, so we print * and increase **count**.
  - **count==2**, which is not **<= rowNumber** so the inner loop terminates and we print **endl**.

  At this point, we increase **rowNumber**.
- Now **rowNumber == 2**, which is **<= numberOfRows**, 2, and we then
  - Start **count** at **1**, which is **<= rowNumber**, 2, so we print * and increase **count**.
  - **count==2**, which is **<= rowNumber**, 2, so we print * and increase **count**.
  - **count==3**, which is not **<= rowNumber** so the inner loop terminates and we print **endl**.

  At this point, we increase **rowNumber**.
- Now **rowNumber == 3**, which is not **<= numberOfRows**, 2, so the outer loop terminates.

Nesting

Consider determining a shipping cost:

```
if( country == "USA") { // if within USA
    if ( cost < USA_free_ship ) { // but not qualify for free shipping
        cost += USA_shipping_fee; // charge fee
    }
}
// if either Canada or Mexico
else if ( (country == "Canada") || (country == "Mexico") ) {
// but not quality free for shipping
    if ( cost < NorthAmerica_free_ship ) {
        cost += NorthAmerica_shipping_fee; // charge fee
    }
}
// not in North America so world shipping
else if ( cost < world_free_ship ) { // not quality for free world shipping
    cost += world_shipping_fee; // charge fee
}
```

Index Variable Names

An **index variable** is one we use in counting through a list of numbers or indexing a container.

Generally, these variable names can be generic like **i**, **j**, **k**, **ell**, and need not be descriptive unless it helps the readability of the loop itself.

break and continue

In managing control flow for *any type of loop* (more types to be discussed) and **switch statements** (to be discussed), we can use a **break** statement.

A **break** statement breaks us out of the enclosing control structure: a **break** would terminate the enclosing loop.

If there are nested loops, it only breaks out of the loop that immediately encloses it!

break and continue

```
int i = 1;

for (/* empty statement*/; i < 10; i += 2) {
    if ( (i == 3) || (i == 7 ) ) // if i is 3 or 7
        break; // terminate the loop

    cout << i << " ";
}

cout << "loop done!";
```

The output is:

```
1 loop done!
```

Once **(i==3)**, which happens first, or **(i==7)**, the loop is terminated.

break and continue

There is also a **continue** statement that skips to the end of an enclosing loop, before continuing to run that loop.

The **continue** just skips the remaining statements in the body during that round of execution.

break and continue

```
int i = 1;

for (/* empty statement*/; i < 10; i += 2) {
    if ( (i == 3) || (i == 7 ) ) // if i is 3 or 7
        continue; // go to end of loop and continue

    cout << i << " ";
}

cout << "loop done!";
```

The output is:

```
1 5 9 loop done!
```

If **(i==3)** or if **(i==7)** the rest of that loop cycle is skipped over and we go straight to incrementing **i** and checking if we should enter the loop again.

do loops

A slight variant of a **while loop** is a **do loop**. It has the general syntax of

```
do {
    this stuff;
} while ( condition is true );
```

There is no checking of the condition at first: the condition is checked at the end of each loop. A do-loop is guaranteed to run at least once.

Note the semi-colon at the end!

do loops

```
int i = 11;

while (i < 4 ) { // this will never be entered because i < 4 is false
    cout << "WHILE";
}

do { // this body will be executed
    cout << "DO";
} while ( i < 4 ); // and the loop will terminate here because this is false
```

```
DO
```

Note that the condition checked must use variables that are still within scope.

```
do {
    bool b = false;
} while ( b ) ; // ERROR: b no longer in scope!
```

Vectors

Consider trying to keep track of 9 homework scores (out of 100%). One particularly lengthy means would be to do something like:

```
// the homework scores
double HW1 = 0, HW2 = 0, HW3 = 0, HW4 = 0, HW5 = 0, HW6 = 0,
    HW7 = 0, HW8 = 0, HW9 = 0;

// update those scores, take an average, etc.
```

This is quite a lot of work: we have to explicitly write down 9 different variables. If we wanted to find the largest or smallest value, we'd have to do this manually, etc.

And what about for 10 homeworks? Or 6 instead? We may have to rewrite a lot...

Vectors

We can do a lot better by working with a container known as a **std::vector**.

Like a **std::string**, a **std::vector** allows us to store a large collection of data, to append items at the end of the collection, to access data at a given position with a subscript, etc.

Unlike a **std::string** that stores **char**s, a **std::vector** can store any sort of data type we want: **int**s, **long double**s, **std::string**s, even other vectors! To use **std::vector**, we need to **#include<vector>**.

Vectors

An **std::vector** is a **templated class**. This means that alone, without referencing a specific data type the **std::vector** is supposed to store, it is not a class. By specifying the data type, it becomes a class.

To create a **std::vector** storing a given data type, we need to specify that data type inside of angled brackets.

```
vector<int> vectorOfInts; // this is a vector that can store int values
```

A **vector<int>**, **vector<long double>**, **vector<string>**, etc., are all classes. They are all distinct classes: one stores **int**s, one stores **long double**s, one stores **std::string**s, etc.

Vectors

std::vectors store their data **contiguously** in memory: the data are stored next to each other in adjacent blocks of memory.

A **std::vector<double>** might have data arranged as below:

Vectors

Here are some basic functions of the class:

- Creating an empty vector:

```
// v1 stores nothing right now, and can store int values v1 == {}
vector<int> v1;
```

- Inserting a value into a vector with **push_back**:

```
v1.push_back(7); // now v1 == {7}
v1.push_back(8); // now v1 == {7, 8}
v1.push_back(9); // now v1 == { 7, 8, 9 }
```

Vectors

- Finding the size of a vector with **size**:

```
cout << v1.size(); // will print 3: the vector stores 3 values
```

```
3
```

- **Subscripting** a vector:

```
v1[0] = 6; // now v1 == { 6, 8, 9 }
cout << v1[0] + v1[2]; // will add 6 and 9: 0th and 2nd index values
```

```
15
```

Vectors

- Removing the last element with **pop_back**:

```
v1.pop_back(); // now v1 == { 6, 8 }
```

- Check if a vector is empty using **empty** (true if there are no elements, false if at least one element)

```
// will be true because empty is false, and ! false is true
if ( ! v1.empty() ) {
    cout << "The vector is not empty.";
}
```

```
The vector is not empty.
```

Vectors

Like **std::string**s, a **std::vector** indexes its values from 0.

It is an error to try to subscript a vector beyond its size and this will result in **undefined behaviour**.

A safer version of the **subscript[]** operator is the **at** function that will throw an exception (which can terminate the program) if the index values are invalid rather than proceeding with unknown behaviour.

```
cout << v1.at(4); // will terminate the program because 4 >= size of v1
```

It is an error to try to **pop_back** on an empty vector and this will result in **undefined behaviour**.

Vectors

The way **std::vector**s work internally in the language is that a certain amount of memory is allocated to store the elements of the vector - generally more space than there are elements in the vector. This gives room for the vector to grow.

If the vector needs to grow beyond the amount of space allocated for its elements (the **capacity**), then a new and larger block of memory is created for the vector and all of its contents are moved to the new block of memory before adding more elements.

Vectors

The process of adding an element is in general a very fast process, $O(1)$, we say, meaning that regardless of how many elements are in the vector, it takes a constant amount of time to append a new element.

The process of needing to move elements from one segment of memory to another can be longer and scales with the length of the vector. We write this as $O(n)$.

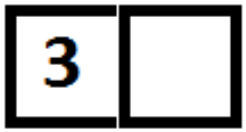
size=0

capacity=2

Vectors

The process of adding an element is in general a very fast process, $O(1)$, we say, meaning that regardless of how many elements are in the vector, it takes a constant amount of time to append a new element.

The process of needing to move elements from one segment of memory to another can be longer and scales with the length of the vector. We write this as $O(n)$.



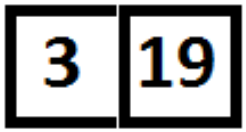
size=1

capacity=2

Vectors

The process of adding an element is in general a very fast process, $O(1)$, we say, meaning that regardless of how many elements are in the vector, it takes a constant amount of time to append a new element.

The process of needing to move elements from one segment of memory to another can be longer and scales with the length of the vector. We write this as $O(n)$.



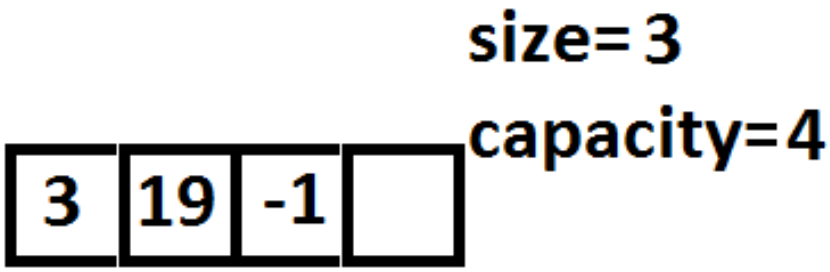
size=2

capacity=2

Vectors

The process of adding an element is in general a very fast process, $O(1)$, we say, meaning that regardless of how many elements are in the vector, it takes a constant amount of time to append a new element.

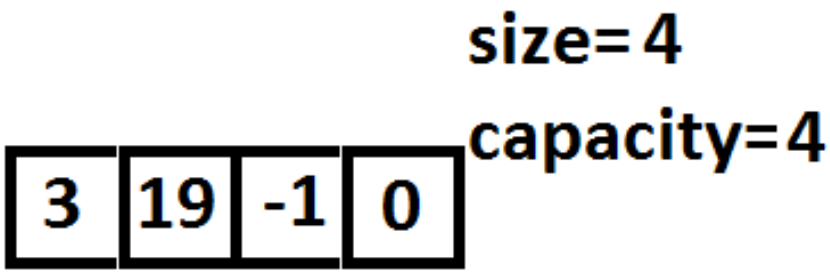
The process of needing to move elements from one segment of memory to another can be longer and scales with the length of the vector. We write this as $O(n)$.



Vectors

The process of adding an element is in general a very fast process, $O(1)$, we say, meaning that regardless of how many elements are in the vector, it takes a constant amount of time to append a new element.

The process of needing to move elements from one segment of memory to another can be longer and scales with the length of the vector. We write this as $O(n)$.



Vectors

- The **capacity** function tells us the capacity of a vector: the largest size it can reach before more memory needs to be allocated if we wish to add more elements. The **capacity** of a vector is always at least as large as its **size**. For example,

```
cout << "v1 size: "<< v1.size() << endl;
cout << "v1 capacity: "<< v1.capacity();
```

```
v1 size:  2
v1 capacity:  8
```

- We can **reserve** space by requesting that the capacity be at least as large as our input parameter.

```
v1.reserve(20); // now v1.capacity() >= 20
```

Vectors

- We can **resize** a vector: if the new size we specify is less than the vector, elements are popped off the back until the size is its new value; if the new size we specify is larger than the current size, default/value initialized values are inserted into the vector until its size is the new value.

  Value initialization for a numeric type means a value of **0**.

```
// currently v1 == {6, 8}

v1.resize(4); // now v1 == { 6, 8, 0, 0 }

v1.resize(1); // now v1 == {6}
```

Vectors

- We can construct a **std::vector** with an initial size. All initial elements of this vector will be default initialized (or value initialized for fundamental types).

```
vector<double> v2(4); // v2 == {0, 0, 0, 0}
vector<string> v3(5); // v3 == { "", "", "", "", ""}
```

Warning: if we have a vector storing some sort of a class type (like string), we cannot use the above construction unless that class type can be default constructed! Not every class type can be constructed without giving any input arguments!

Vectors

- We can construct a **std::vector** with an initial size and value to repeat.

```
vector<double> v4(4, 3.14); // v4 == {3.14, 3.14, 3.14, 3.14}
vector<string> v5(2, "AB"); // v5 == { "AB", "AB"}
```

- We can also construct a **std::vector** by giving it an initial list of values:

```
vector<char> v6 = { 'a', 'b', 'c'}; // can do this...
vector<char> v7 { 'a', 'b', 'c'}; // or this
```

Both vectors above are equal to **{ 'a', 'b', 'c'}** .

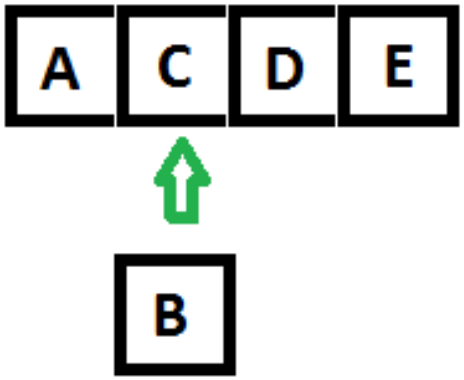
Vectors

Vectors are particularly nice when we want to insert or remove an item at the end, but inserting elements at an arbitrary position or removing an element at an arbitrary position is more cumbersome.

The basic logic is that to **insert** an element, all elements from the final element through to the element at the insertion place need to be shifted right, and then we can set the value at the insertion point.

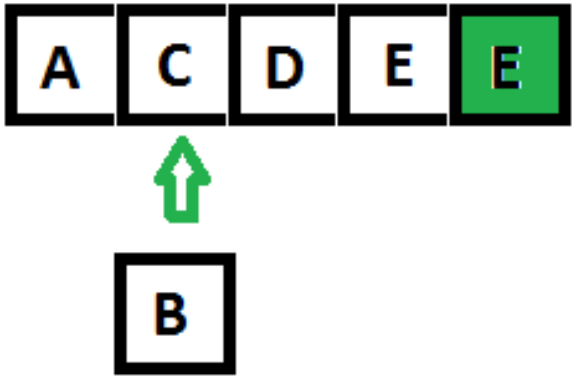
Vectors

Insertion



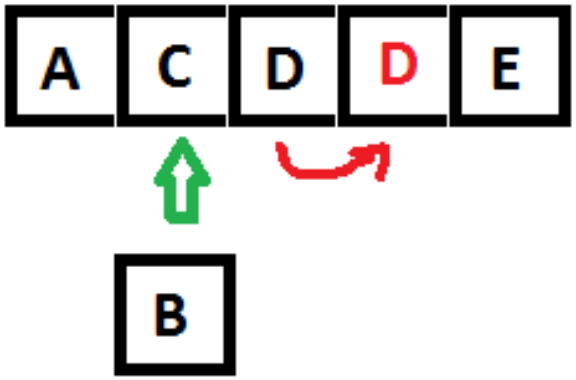
Vectors

Insertion



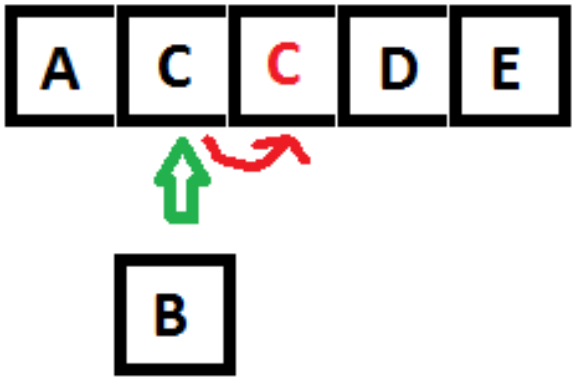
Vectors

Insertion



Vectors

Insertion



Vectors

Insertion

Vectors

Let's consider this in code. Suppose we wish to insert the value **insertVal** at position **insertPos** into a **std::vector<double>**, **foo**, where the **insertPos < foo.size()**.

```
double lastValue = foo[ foo.size() - 1 ]; // the last value in the vector
foo.push_back ( lastValue ); // copy it with a push_back

/* from 2nd last element down to element just right of insertion point,
   we overwrite values */
for ( size_t i = foo.size()-2; i > insertPos; - - i) {
   foo[i] = foo[i-1]; // replace right value with left value
}

// overwrite the element at the insertion point
foo[ insertPos ] = insertVal;
```

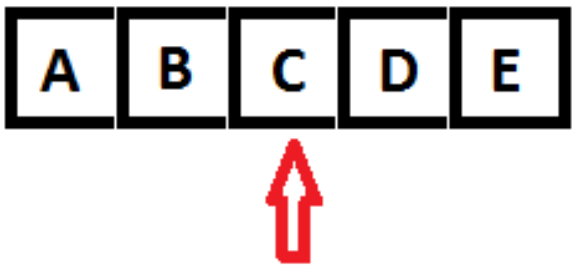
Vectors

To **remove** an element and preserve the order, we need to shift elements to overwrite the element we wish to remove, and to delete/overwrite all duplicates that arise.

To **remove** an element without preserving the order, we can overwrite the to-be-removed element by the last element and remove the last element at the end.

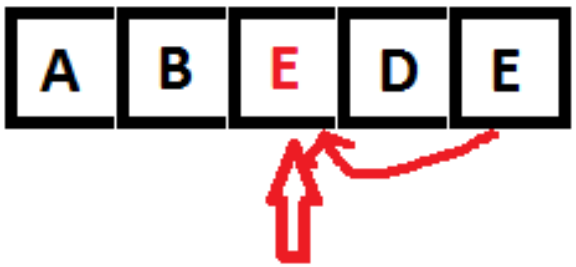
Vectors

Removal where order does not matter



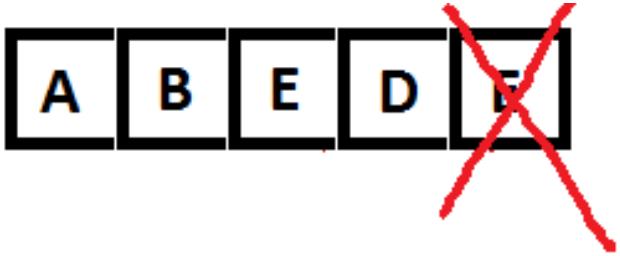
Vectors

Removal where order does not matter



Vectors

Removal where order does not matter



Vectors

Removal where order does not matter

Vectors

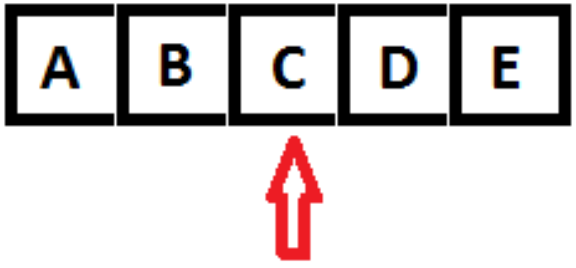
Let's consider this in code. Suppose we wish to remove the element at position **removePos** from a **std::vector<double>**, **foo**, where **removePos < foo.size()**.

```
foo[ removePos ] = foo[ foo.size()- 1]; // overwrite with last value

foo.pop_back(); // and then pop_back
```

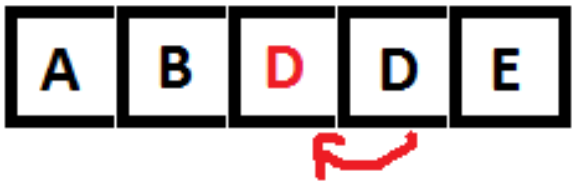
Vectors

Removal where order matters (burying the value alive...)



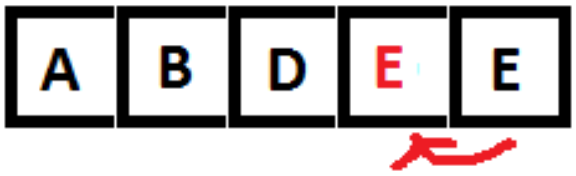
Vectors

Removal where order matters (burying the value alive...)



Vectors

Removal where order matters (burying the value alive...)

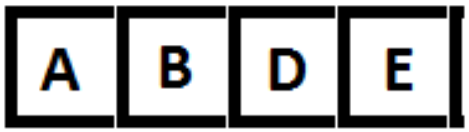


Vectors

Removal where order matters (burying the value alive...)

Vectors

Removal where order matters (burying the value alive...)



Random Numbers

It is possible to generate a random number using C++. Random numbers are useful in simulations: synthesizing the effect of rolling a die, flipping a coin, etc.

We will focus upon a very old means of generating random numbers, the **rand()** function. This function is defined in the **<cstdlib>** header. The **rand** function returns an **int** value between **0** and **RAND_MAX** (some large **int** value).

```
cout << rand() << endl << rand();
```

```
846930886
1804289383
```

Random Numbers

Before using the **rand** function, it should be **seeded**: it turns out the numbers aren't completely random and without seeding, subsequent runs of the program will yield the same random numbers.
We need to **#include<ctime>**, a library used in managing time, and then before using **rand()**, we must include the line of code

```
srand( time( nullptr ) ); // srand seeds the rand function
```

Based upon the number of seconds that have elapsed since January 1, 1970, at the instant the program runs, a new seeding is given to the **rand** function.

Random Numbers

The numbers are *close?* to being uniformly selected at random from **0** to **RAND_MAX** inclusive, with each number equally likely to be selected.

In many applications, we need only concern ourselves with a smaller list of numbers. Suppose we wanted to simulate rolling a die, with numbers 1-6:

```
int dieRoll = rand() % 6 + 1; // a number from 1-6
```

Random Numbers

By modding a random **int** by the **int**-value **v**, the possible values are **0, 1, 2, 3, ... v-1**.

For the die, **rand()%6** will be a number 0-5, and adding 1 gives us a number from 1-6.

In general to generate a random **int** value from **a** to **b** inclusive, we would use:

```
int random_from_a_to_b = rand() % (b - a + 1) + a;
```

The largest value of **rand() % (b - a + 1)** is **b-a**, which when added to **a** is **b**; the smallest value of **rand() % (b - a + 1)** is **0**, which when added to **a** is **a**.

Random Numbers

We can also generate random floating point numbers:

```
// this will be from 0 to 1
double random_0_to_1 = static_cast<double>( rand() ) / RAND_MAX;

// by rescaling and adding a, this will be from a to b
double random_a_to_b = a + (b - a) * random_0_to_1;
```

Random Numbers

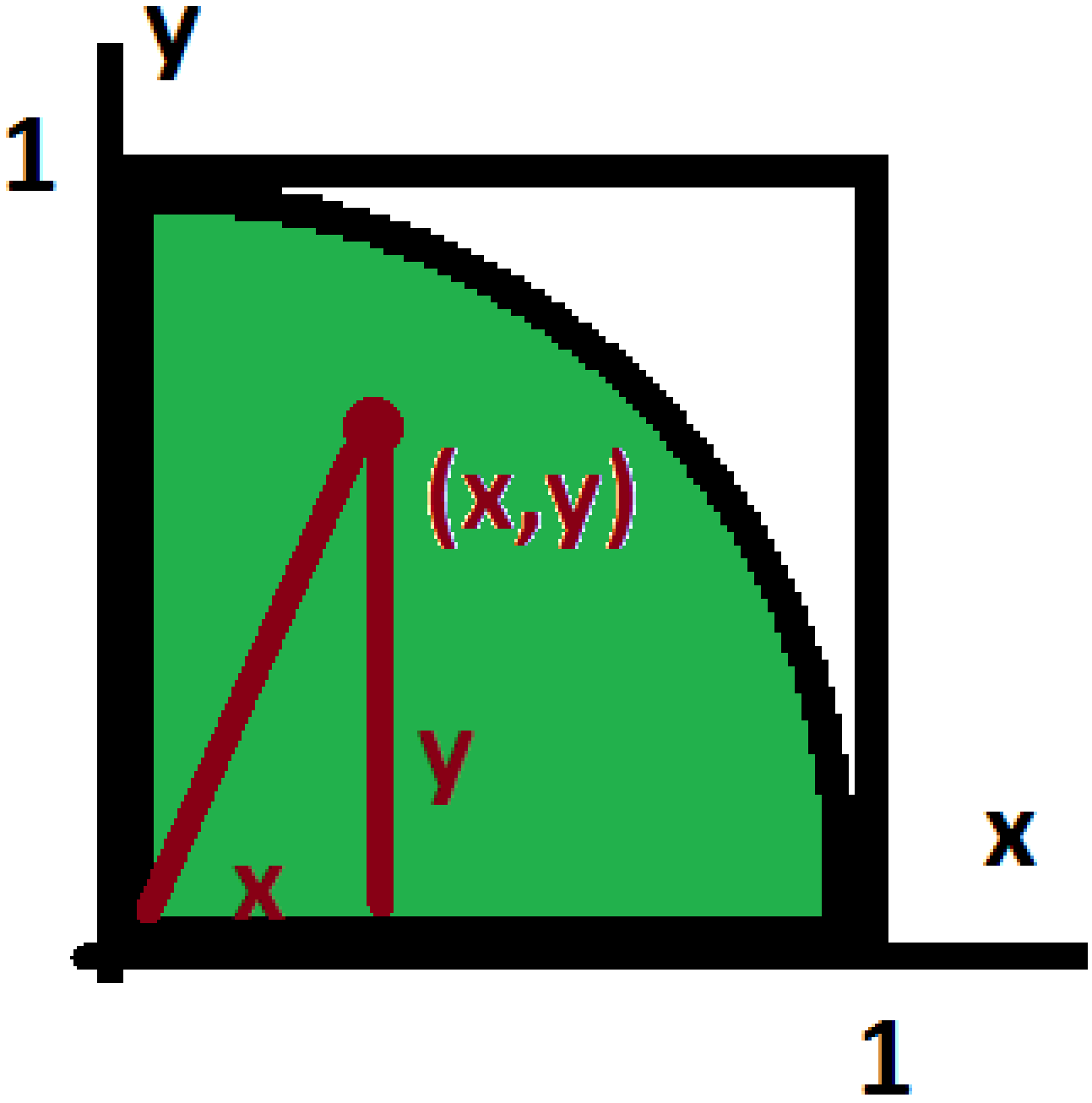
Consider a quarter circle inscribed in a square as shown on the next slide, in a Cartesian coordinate system (x, y).

The side length of the square is 1 and the square area is $1^2 = 1$.

The radius of the quarter circle is 1 and the circle area is $\pi 1^2/4 = \pi/4$

A point (x, y) inside/on the boundary of the quarter circle (x, y) is a pair of values $0 \leq x \leq 1$, $0 \leq y \leq 1$, such that $x^2 + y^2 \leq 1$. In other words: the distance from the origin (bottom left corner of square) to the point is ≤ 1.

Random Numbers



Random Numbers

If we randomly choice a point at random in the square, the probability it is within the quarter circle is the ratio of the quarter circle area to the square area, $(\pi/4)/1 = \pi/4$. Over many random choices of points, the fraction of times a point lies within the quarter circle among simulations should be $\pi/4$.

We can use this to simulate a value for π using the pseudo-code:

- Set numberOfRuns = lage number
- Set circleHits = 0
- For each i from 1 to numberOfRuns
  - Pick a point x from 0 to 1
  - Pick a point y from 0 to 1
  - If (x,y) in circle, increase circleHits
- Set hitRatio = circleHits / numberOfRuns
- Set pi = 4*hitRatio

Multidimensional Vectors

We can also have **std::vector**s storing **std::vectors**!

Sometimes it makes sense to visualize data as two- (or more) dimensional. Consider storing a matrix/table of numbers as below:

1 2 3
4 5 6
7 8 9
10 11 12

This could be viewed as a **vector**, **M**, of rows, where each row such as **{ 4, 5, 6 }** is itself a **vector<int>**.

In this case, **M[0][2]** would be **3**: the element with index **2** in the first row, **M[0]**. Likewise, **M[3][1]** would be **11**, etc.

Multidimensional Vectors

Here's how we could create **M**:

```
// M is a vector that stores vector<int>, begins empty
vector< vector<int> > M;

for(size_t i = 0; i < 4; ++i) { // for row indices 0 to 3

   /* given i, the row values are 3*i+1, 3*i+2, 3*i+3
      but we need to explicitly convert the size_t to int */

   vector<int> row = { 3*static_cast<int>(i) + 1, 3*static_cast<int>(i) + 2,
      3*static_cast<int>(i) + 3 };

   M.push_back( row ); // add the row to the vector
}
```

Multidimensional Vectors

Remarks: the code would compile without **static_cast<int>(i)**, but the compiler would likely generate a warning in converting **size_t**, an unsigned integer type, to an **int**, a signed integer type. To stop the compiler from complaining, we cast.

We chose to construct one row at a time, and add a fully constructed row to the vector **M**.

Multidimensional Vectors

We could have similarly constructed **M** via:

```
// Make M a vector of size 4, with all elements a vector of int of size 3
vector< vector<int> > M( 4, vector<int>(3) );

for( size_t i = 0; i < 4; ++i ) { // for each row
    for (size_t j = 0; j < 3; ++j) { // and each column

        // set the value at row i column j
        M[i][j] = 3*static_cast<int>(i) + static_cast<int>(j) + 1;
    }
}
```

Multidimensional Vectors

Remarks: In the initialization of **M**, **M** will have a size of 4, and we set the vector of ints that it stores to be the value **vector<int>(3)**: a **vector<int>(3)** is a **vector<int>** of size **3**, with all values initially **0.** Thus, **M** is initially equal to:

0 0 0
0 0 0
0 0 0
0 0 0

Multidimensional Vectors

This can look a little confusing at first. It's important to recall we are thinking of **M[i]** as the row with index **i**, and **M[i][j]** as the jth component of that row.

We had to recognize the pattern of those counting numbers that when indexed from 0 for both rows and columns, the general value is 3*i + j + 1.

Arrays

An older **C-style** means for managing data in a like manner to **std::vector**s is an **array**. A **C-style array** is a built in data type that manages a contiguous block of memory.

An array has a **fixed capacity**, which can never be changed, and we have to manually keep track of the **size** ourselves. The size of an array must be a **const** value or a literal.

Arrays

An array is defined with the **[]** notation.

Values in an array are generally uninitialized until we set them. Arrays can only store primitive types and classes that can be default constructed (or else we have to explicitly list every value to initially store to avoid a compiler error).

Arrays I

```
const size_t capacityA = 10; // the capacity set ahead of time

int arrayA[ capacityA ]; // define an array of ints with size capacityA

size_t sizeA = 0; // a variable to track the size of the array

for(size_t i=0; i < 4; ++i ) { // initialize first 4 values to 1
    arrayA[i] = 1; // set value to 1
    ++sizeA; // increase the size
}

arrayA[1] = 7;

// now arrayA == { 1, 7, 1, 1, ?, ?, ?, ?, ?, ? }
```

Arrays II

```
// remove the last element
if ( sizeA > 0 ) { // ensure size is big enough to remove an element
    - - sizeA; // decreasing size effectively removes the last element
}

// display all elements in the array
for (size_t i=0; i < sizeA; ++i ) {
    cout << arrayA[i] << endl;
}
```

```
1
7
1
```

Arrays III

```
// output values that were never set or out of range
cout << arrayA[8] << endl << arrayA[ capacityA ];
```

```
-858993460
-858993460
```

If a value is not set, then it can be anything. Accessing an array beyond its capacity can lead to **undefined behaviour**.

Arrays

Arrays lack member functions. Consider removing the element at position **1** while **preserving** the order of the elements.

```
size_t removePos = 1; // position of removal

// from removal position to 2nd last position
for (size_t i=removePos; i < sizeA - 1; ++ i) {
    // overwrite value with its right neighbour value
    arrayA[i] = arrayA[i+1];
}

- - sizeA; // decrease the size
```

We did not use **size()** or **pop_back()** as we could have for **std::vector**.

Arrays

There are alternative ways to initialize arrays:

```
double arrayB[5]; // arrayB will store 5 double values

// arrayC will have size 6, with the given values
bool arrayC[] = { true, true, false, false, true, true };

/* arrayD will have size 8 and unspecified values are value
   initialized: arrayD == { 1, 2, 3, 4, 5, 0, 0, 0 } */
unsigned arrayD[8] = { 1, 2, 3, 4, 5 };
```

Arrays

There is a way to initialize multidimensional arrays, too.

```
// arrayE is an array of size 6, storing arrays of ints of size 8
int arrayE [6][8];

// we set the value in the 6th row (index 5) and 8th column (index 7) to 90
arrayE[5][7] = 90;
```

Using **std::vector**s is just better in general as C-style arrays are outdated.

Switch Statements

A **switch statement** offers a different control flow structure when we are conditioning upon the value of **integer types**.

The syntax is:

```
switch( value ) {
case possibleValue1:
    do this stuff;
    break;
case possibleValue2:
    do this stuff;
    break;
// ...
default:
    stuff to do by default;
    break;
}
```

Switch Statements

```
int AP_Exam_Grade = 4;

switch( AP_Exam_Grade ) { // depending on AP exam score
case 5: // if the score is 5, worth credit
    cout << "Worth university credit";     break;
case 4: // if the score is 4, worth credit
    cout << "Worth university credit";     break;
case 3: // if the score is 3, may be worth credit
    cout << "Possibly worth credit.";     break;
case 2: // if the score is 2, not worth credit
    cout << "Not worth credit.";     break;
case 1: // if the score is 1, not worth credit
    cout << "Not worth credit.";     break;
default: // if the score is not 1-5, it is invalid
    cout << "Invalid score.";     break;
}
```

Switch Statements

We could have accomplished the same task with **if** and **else**:

```
int AP_Exam_Grade = 4;

if ( (AP_Exam_Grade == 5) || ( AP_Exam_Grade == 4) ) { // worth credit
    cout << "Worth university credit";
}
else if ( AP_Exam_Grade == 3 ) { // may be worth credit
    cout << "Possibly worth credit.";
}
else if ( (AP_Exam_Grade == 2) || (AP_Exam_Grade == 1) ) { // no credit
    cout << "Not worth credit.";
}
else { // of not 1-5, score is invalid
    cout << "Invalid score.";
}
```

Switch Statements

Remarks: based on the value being switched, a matching value is sought in the cases.

The **default** can appear anywhere in the switches: if another switch case has not already been found for the switch value, the default behaviour will be executed.

Some switch statements can be done without a **break;** statement following the case, but that is more error prone and not a robust coding practice. It could very easily lead to serious bugs and glitches later on. If a case does not end with a **break** then all subsequent cases are executed, whether or not they are the correct case.

Switch Statements

```
switch( true ) {
case true:
   cout << "true";
case false:
   cout << "false";
default:
   cout << "neither true nor false";
}
```

The code above will output:

```
truefalseneither true nor false
```

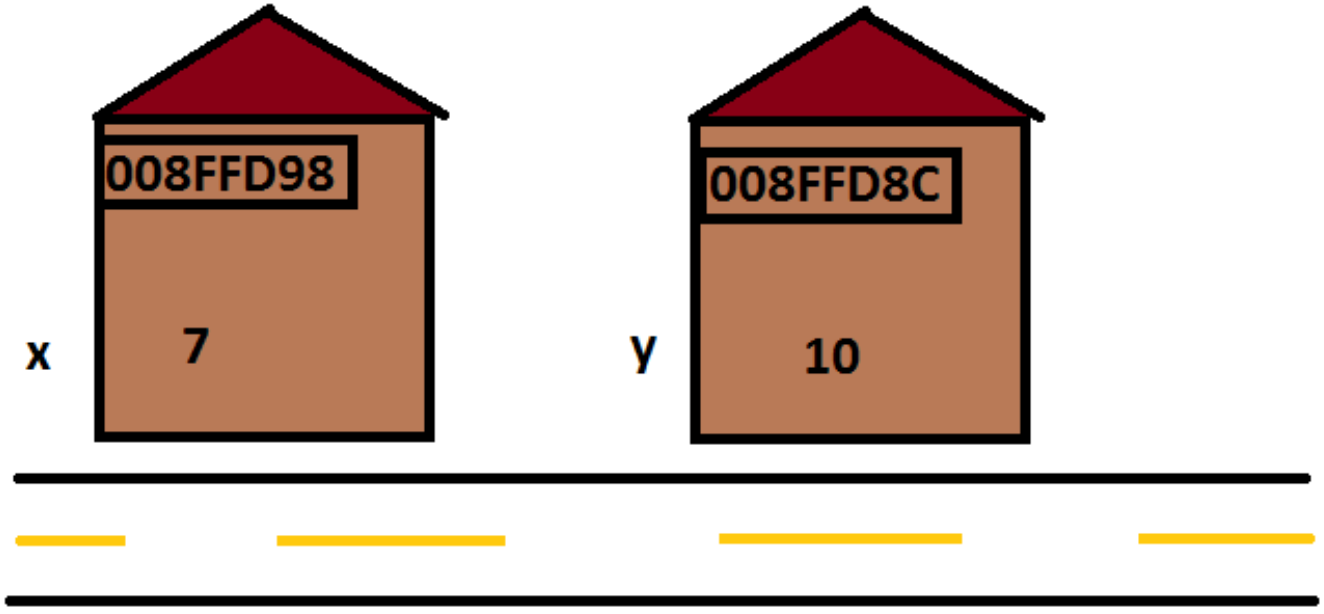
All switch statements are executed after a matching case until there is a break.

Pointers

Recall that when a variable is defined, it is given a place in memory. That place has an **address**.

The address of a variable is stored in **hexadecimal**, base 16: uses digits 0-F (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

```
int x = 7, y = 10;
```



Pointers

The address of a variable is accessed by the **address of operator &**.

```
cout << "x address: "<< &x << endl;
cout << "y address: "<< &y << endl;
```

```
x address:  008FFD98
y address:  008FFD8C
```

Pointers

We can store the memory location of a variable in something called a **pointer**.

A **pointer** is a data type that stores the address of a specific type of variable.

```
int *xpointer = &x; // xpointer is a pointer to int, pointing to x
int *ypointer = &y; // ypointer is a pointer to int, pointing to y
```

Both **xpointer** and **ypointer** above are of type **int ***, a pointer to **int**.

Pointers

Pointers can be moved around:

```
xpointer = &y; // now xpointer points to y
```

Remark: the line of code below would have had the same effect as the line of code above:

```
xpointer = ypointer; // use ypointer to set xpointer
```

Pointers

The general syntax of defining and initializing a pointer to a variable of a type **valueType** is:

*valueType * pointerName = someAddress;*

```
const double pi = 3.14159;

// piPointer is of type const double* and points to pi
const double * piPointer = &pi;

string s;

string *sPtr = &s; // sPtr is of type string* and points to s
```

Pointers

These pointers give us another means to access a variable: if we know where a variable lives, we effectively have direct access to it.

Through a pointer and a **dereferencing operator** *, we gain direct access to whatever variable a pointer is pointing to.

```
// below is same as: cout << pi << endl;
cout << *piPointer << endl;
```

Pointers

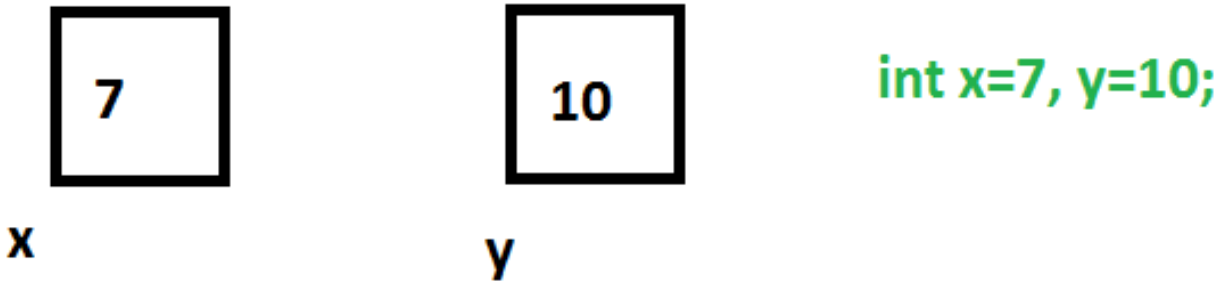
```
// below is same as: cout << s.size() << endl;
cout << (*sPtr).size() << endl;

// below is same as: s = "new message";
*sPtr = "new message";

/* recall that xpointer points to y, thus
below is same as: ++y; */
++(*xpointer);
```

Pointers

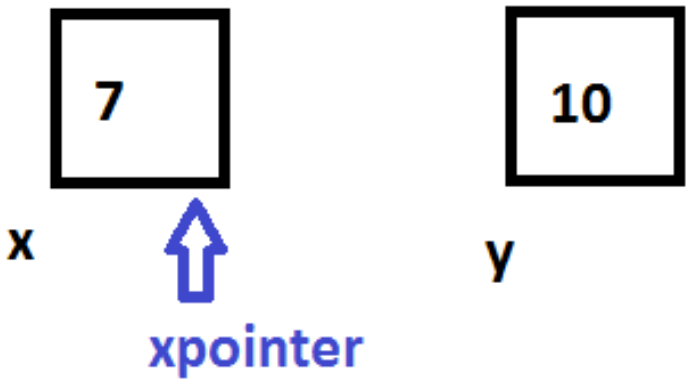
Some work with pointers: a **pointer** keeps track of a variable's location in memory. **xpointer** and **ypointer** are pointers.




```
int x=7, y=10;
```

Pointers

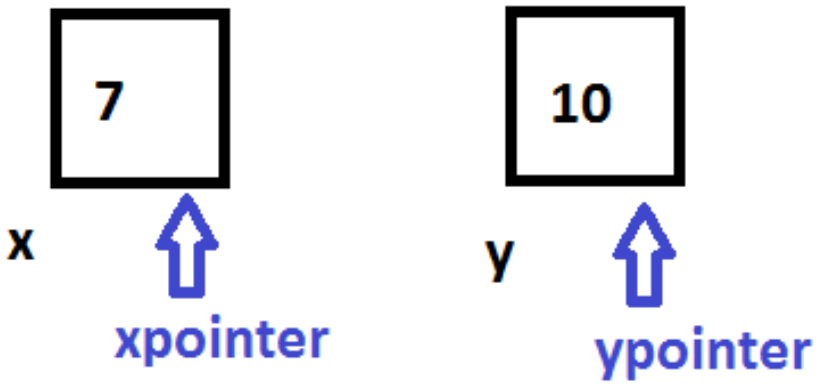
Some work with pointers: a **pointer** keeps track of a variable's location in memory. **xpointer** and **ypointer** are pointers.




```
int *xpointer = &x;
```

Pointers

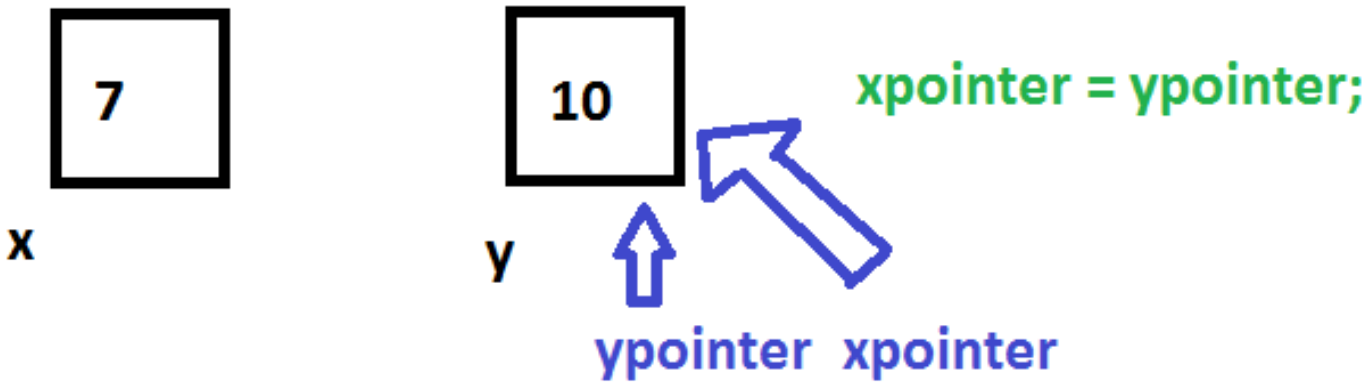
Some work with pointers: a **pointer** keeps track of a variable's location in memory. **xpointer** and **ypointer** are pointers.




```
int *ypointer =&y;
```

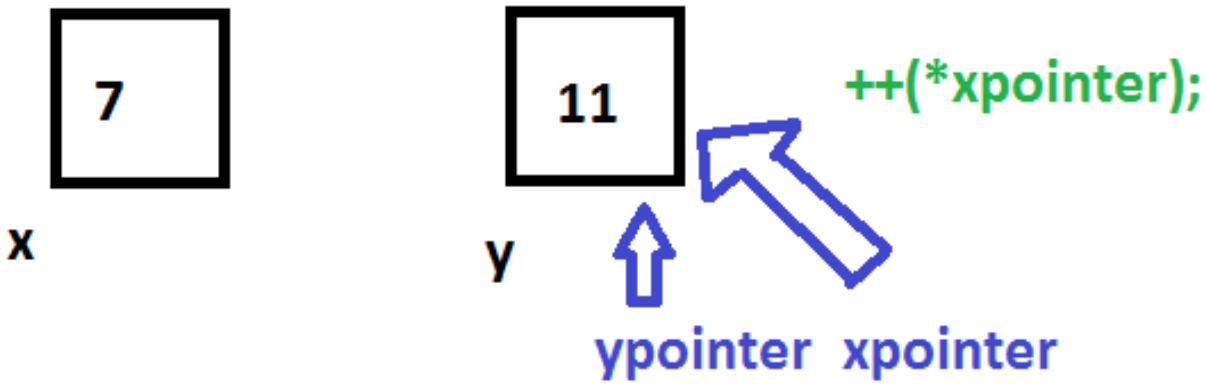
Pointers

Some work with pointers: a **pointer** keeps track of a variable's location in memory. **xpointer** and **ypointer** are pointers.



Pointers

Some work with pointers: a **pointer** keeps track of a variable's location in memory. **xpointer** and **ypointer** are pointers.



Pointers

Remark: the preceding cartoon pictures portray a nice visual for how pointers can move around but one subtle detail is missing...

Each pointer variable itself has a place in memory. So **xpointer** and **ypointer** really should have their own little box like **x** and **y** do. Instead of storing an **int** like **x** and **y**, they each store the **address of an int**.

Pointers

When dealing with class objects and accessing members of dereferenced pointers, there is an alternative syntax with the **arrow ->** notation:

```
cout << sPtr->size(); // this is the same as
cout <<(*sPtr).size(); // this
```

If **ptr** is a pointer to a class object then

ptr->something is the same as **(*ptr).something**

Pointers

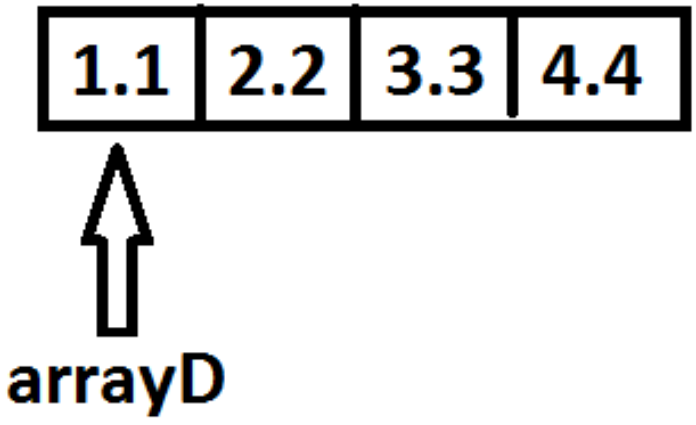
When we create a C-style array, we are working with pointers!

```
double arrayD[4] = { 1.1, 2.2, 3.3, 4.4 };
```

The array variable name is actually a pointer! Above **arrayD** is a **double***. We can then use the array name as a pointer.

```
// will output the address of the first element in the array
cout << arrayD << endl;

cout <<*arrayD; // will output arrayD[0], the value 1.1, the first element
```



Summary

- Statements that can be **true** or **false** are useful in managing control flow.
- An **if... else** structure allows for a given body to be executed based on a condition being true.
- **while**, **for**, and **do** loops allow for a sequence of statements to be executed multiple times. A **do** loop only checks its condition at the end.
- A **break** statement exists an enclosing loop or switch statement; a **continue** statement skips the remaining steps in the enclosing loop's iteration.
- **std::vector** is a templated class that stores **contiguous** blocks of data, capable of quickly accessing elements, appending new elements, and removing end elements.

Summary

- A **std::vector** has many member functions and constructors that resemble that of **std::string**.
- A **C-style** array has an unchangeable capacity and lacks member functions that a vector has.
- Random **int**s can be generated through **rand**, which must first be seeded.
- **Switch statements** can sometimes be used in lieu of **if... else** structures.
- A **pointer** tracks a variable type and its address; **dereferencing** a pointer gives direct access to the element it references.