Methods

Named Blocks of Code

Methods are just **named** blocks of code

variables

if-else

switch

for

while

do-while

Code

Methods have 2 parts: a **declaration** and a **definition**

- Method declaration: introduces the method's name and type. It describes **how to communicate** with the block of code. For classes, these are found in the **.h file**.

- Method definition: the **code** that is associated with the method's name. For classes, these are usually found in the **.cpp file**.

A Method declaration

```
class Calculator
{
public:
   //the declaration of the sum method
   int sum(int number1, int number2);
};
```

This method declares how to communicate with it.

- It is non-static.
- It requires 2 ints to be passed to it.
- It returns an int.

A Method declaration

```
int sum(int number1, int number2);
```

How do we call the method?

- Since it is **non-static**, we need an instance of the Calculator class.

```
Calculator t1000; //a stack instance
```

- Since sum requires 2 ints to be passed to it…
- Since sum returns an int…

```
int n1 = 5, n2 = 2;
int result = t1000.sum(n1, n2);
```

A Method definition

```
int Calculator::sum(int number1, int number2)
{
   return number1 + number2;
}
```

Methods

Passing Parameters by Value

Method Parameters: pass by value

Pass by value = COPY

- When you pass parameters by value to a method, the value is COPIED from the code that calls the method to a new variable in the method.
- Changes made to the variable inside the method DO NOT affect the code that calls the method.

Pass by Value (COPY)

```
int n1 = 5, n2 = 2;
int result = t1000.sum(n1, n2);
```

Copied to

Copied to

```
int Calculator::sum(int number1, int number2)
{
    return number1 + number2;
}
```

Arrays

C++ Arrays (tutorialspoint.com)

Arrays: the basics

Arrays store a **fixed-size** sequential collection of elements of the **same type**.

Arrays are often thought of as a collection of variables. For example, if you needed 10 integers, you could create 10 separate int variables.
Or you could create **1 array** variable to **hold 10 integers**.

Arrays: the basics

The data for the array are **stored contiguously** in memory. Each item is in memory right next to each other.

Arrays: the basics

Declaring arrays:

type variableName [size];

```
int highScores[5];
```

Initializing arrays:

```
int highScores[5] { 100, 90, 80, 70, 60 };
```

Accessing arrays:

```
int myScore = highScores[3]; //stores 70 in the myScore variable
```

Arrays: the pros

Good for <u>fixed-sized</u> collections.

Indexing is FAST!

The time it takes to access the first element is the SAME as it takes to access the last element. The worst-case performance is **O(1)** or constant time.

Why? The indexing is a calculation:
memory address + (index * size of type)

Arrays: the cons

Must allocate all the space needed.

When creating an array, you must specify how much memory to set aside even if all the memory is not used.

Bad when resizing.

Resizing requires creating a new array and custom code to copy items from 1 array to the other.

std::vector<type>

Vector in C++ STL - GeeksforGeeks

std::vector<type>

- Vectors are dynamic arrays – they **resize themselves** automatically.
- They use an **array internally** to store the elements contiguously.
- Vectors are defined in the **<vector>** header file.

std::vector<type>

Creating a vector…

```
#include <vector>

std::vector<int> highScores;
```

std::vector<type>

Adding items to a vector…

```
std::vector<int> scores { 1,2,3,4 }; //add items on the initializer

scores.push_back(rand()); //adds a random int to the end of the vector
```

std::vector<type>

Accessing items in a vector…

```
int score = scores[0]; //indexes are zero-based like arrays
```

std::vector<type>

Looping over a vector using for loop…

```
for (int i=0; i < scores.size(); ++i)
   std::cout << scores[i] << " ";
```

std::vector<type>

Looping over a vector using iterators…

```
for (auto i = scores.begin(); i != scores.end(); ++i)
   std::cout << *i << " ";
// *i dereferences the iterator to give you the value at that location
```

begin() – returns an iterator pointing to the first element

end() – returns an iterator pointing to the element ***AFTER*** the last element

std::vector<type>

Looping over a vector using range based for loop…

```
for (auto highScore : scores)
   std::cout << highScore << " ";
```

std::vector<type>.clear()

clear() removes all the items in a vector

```
std::vector<int> scores { 1,2,3,4 };
scores.clear();
```

std::vector<type>.erase(position)

erase(position) removes the item at the position

```
std::vector<int> scores = { 1,2,3,4 };
```

- Position is an iterator.
- Use begin() to start from the beginning of the vector.
  - scores.erase(scores.begin()) will remove the first item in the vector
  - Result: { 2,3,4 }
- Add a number to begin() to remove an item at a different index
  - scores.erase(scores.begin()+2) will remove the 3rd item in the vector
  - Result: { 1,2,4 }

std::vector<type>.erase(start, end)

erase(start, end) removes the items in the range

```
std::vector<int> scores = { 1,2,3,4,5,6 };
```

- start and end are iterators.
- The item at the end position is not removed.
  - `scores.erase(scores.begin()+2, scores.begin()+4)`
  - Result: { 1,2,5,6 }