CONST

CONST

C++ HAS AN INTERESTING NEW KEYWORD CALLED `const`.

CONST

C++ HAS AN INTERESTING NEW KEYWORD CALLED `const`.

A VARIABLE CAN BE MARKED `const`, AND THEN ANY ATTEMPT TO CHANGE ITS VALUE WILL THROW AN ERROR

CONST

C++ HAS AN INTERESTING NEW KEYWORD CALLED `const`.

A VARIABLE CAN BE MARKED `const`, AND THEN ANY ATTEMPT TO CHANGE ITS VALUE WILL THROW AN ERROR

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

CONST

C++ HAS AN INTERESTING NEW KEYWORD CALLED `const`.

A VARIABLE CAN BE MARKED `const`, AND THEN ANY ATTEMPT TO CHANGE ITS VALUE WILL THROW AN ERROR

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

`const` CAN BE USED IN ALL KINDS OF INTERESTING WAYS!

EXAMPLE 30

DEFINE AND USE `const` VARIABLES

EXAMPLE 30 DEFINE AND USE const VARIABLES

```
// create a const int variable - the declaration and initialisation must be in the same sentence
const int x = 5;
// Why must the initialisation occur in the definition itself?
// Because re-assignment to a const will not work
// x = 10;

// create a const string
const string firstName("Vitthal");
// Any attempt to modify this string will throw some ferocious errors!
//firstName.insert(0,"Mr. ");

return 0;
```

EXAMPLE 30 DEFINE AND USE const VARIABLES

```
// create a const int variable - the declaration and initialisation must be in the same sentence
const int x = 5;
// Why must the initialisation occur in the definition itself?
// Because re-assignment to a const will not work
    x = 10;

// create a const string
const string firstName("Vitthal");
// Any attempt to modify this string will throw some ferocious errors!
//firstName.insert(0,"Mr. ");

return 0;
```

EXAMPLE 30 DEFINE AND USE const VARIABLES

```
// create a const int variable - the declaration and initialisation must be in the same sentence
const int x = 5;
// Why must the initialisation occur in the definition itself?
// Because re-assignment to a const will not work
// x = 10;

// create a const string
const string firstName("Vitthal");
// Any attempt to modify this string will throw some ferocious errors!
//firstName.insert(0,"Mr. ");

return 0;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example30.cpp
Example30.cpp:10:5: error: read-only variable is not assignable
  x = 10;
  ~ ^
1 error generated.
```

EXAMPLE 30 DEFINE AND USE const VARIABLES

```
// create a const int variable - the declaration and initialisation must be in the same sentence
const int x = 5;
// Why must the initialisation occur in the definition itself?
// Because re-assignment to a const will not work
// x = 10;

// create a const string
const string firstName("Vitthal");
// Any attempt to modify this string will throw some ferocious errors!
//firstName.insert(0,"Mr. ");

return 0;
```

EXAMPLE 30 DEFINE AND USE const VARIABLES

```
// create a const int variable - the declaration and initialisation must be in the same sentence
const int x = 5;
// Why must the initialisation occur in the definition itself?
// Because re-assignment to a const will not work
// x = 10;

// create a const string
const string firstName("Vitthal");
// Any attempt to modify this string will throw some ferocious errors!
  firstName.insert(0,"Mr. ");

return 0;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example30.cpp
Example30.cpp:15:13: error: no matching member function for call to 'insert'
  firstName.insert(0,"Mr. ");
  ~~~~~~~~~~^~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1529:19: note:
      candidate function not viable: 'this' argument has type 'const string' (aka 'const
      basic_string<char, char_traits<char>, allocator<char> >'), but method is not marked
      const
    basic_string& insert(size_type __pos, const value_type* __s);
                  ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1526:19: note:
      candidate function not viable: 'this' argument has type 'const string' (aka 'const
      basic_string<char, char_traits<char>, allocator<char> >'), but method is not marked
      const
    basic_string& insert(size_type __pos1, const basic_string& __str);
                  ^
```

EXAMPLE 31

UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

EXAMPLE 31 UNDERSTAND THAT const ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

EXAMPLE 31 UNDERSTAND THAT const ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

EXAMPLE 31 UNDERSTAND THAT const ONLY GUARANTEES BITWISE CONSTNESS

```
// create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
```

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example31.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Initial value of y = 5 and that of x = 5
Final value of y = 15 and that of x = 15
```

EXAMPLE 31

UNDERSTAND THAT const ONLY GUARANTEES BITWISE CONSTNESS

```
// create a
int x = 5;
// create a
const int&
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
// has the
cout << "Fi                                    x << endl;
```

YES IT HAS! DESPITE THE const NATURE OF THE REFERENCE!

THE REFERENCE Y IS CONST, NOT THE VARIABLE X, SO THE BITS OF X CAN BE CHANGED

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example31.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Initial value of y = 5 and that of x = 5
Final value of y = 15 and that of x = 15
```

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

EVEN A `const int&` CAN BE CHANGED - SIMPLY BY MODIFYING THE VARIABLE IT POINTS TO!

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

EVEN A `const int&` CAN BE CHANGED - SIMPLY BY MODIFYING THE VARIABLE IT POINTS TO!

THIS WAS BECAUSE OUR `const int&` DID NOT POINT TO A `const int`

THE C++ COMPILER WILL ONLY PROTECT THE BITWISE CONST-NESS OF THE ADDRESS OF THE `const int&`

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

EVEN A `const int&` CAN BE CHANGED - SIMPLY BY MODIFYING THE VARIABLE IT POINTS TO!

THIS WAS BECAUSE OUR `const int&` DID NOT POINT TO A `const int`

THE C++ COMPILER WILL ONLY PROTECT THE BITWISE CONST-NESS OF THE ADDRESS OF THE `const int&`

THE C++ COMPILER ONLY UNDERSTANDS BITWISE CONST-NESS - SEMANTICS ARE OWNED BY YOU!!

EXAMPLE 31 UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
  // create a non-const int variable
  int x = 5;
  // create a const reference to this (non-const) int
  const int& y = x;
  cout << "Initial value of y = " << y << " and that of x = " << x << endl;
  // modify x
  x += 10;
  y = 10;
  // has the value of y changed?
  cout << "Final value of y = " << y << " and that of x = " << x << endl;
  // Yes it has! The const reference was modified simply by changing the
original variable it pointed to!
  return 0;
```

EXAMPLE 31

UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
  // create a non-const int variable
int x = 5;
// create a const reference to this (non-const) int
const int& y = x;
cout << "Initial value of y = " << y << " and that of x = " << x << endl;
// modify x
x += 10;
y = 10;
// has the value of y changed?
cout << "Final value of y = " << y << " and that of x = " << x << endl;
// Yes it has! The const reference was modified simply by changing the
original variable it pointed to
return 0;
```

HERE WE TRY TO CHANGE THE VALUE OF X REFERRED TO BY Y USING Y

Y IS A CONST REFERENCE TO INTEGER

EXAMPLE 31

UNDERSTAND THAT `const` ONLY GUARANTEES BITWISE CONSTNESS

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example31.cpp
Example31.cpp:14:5: error: read-only variable is not assignable
  y = 10;
  ~
1 error generated.
```

THIS IS AN ERROR, THE RUNTIME RECOGNIZES THAT YOU ARE TRYING TO CHANGE THE VALUE OF A READ-ONLY VARIABLE Y

EXAMPLE 32

UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

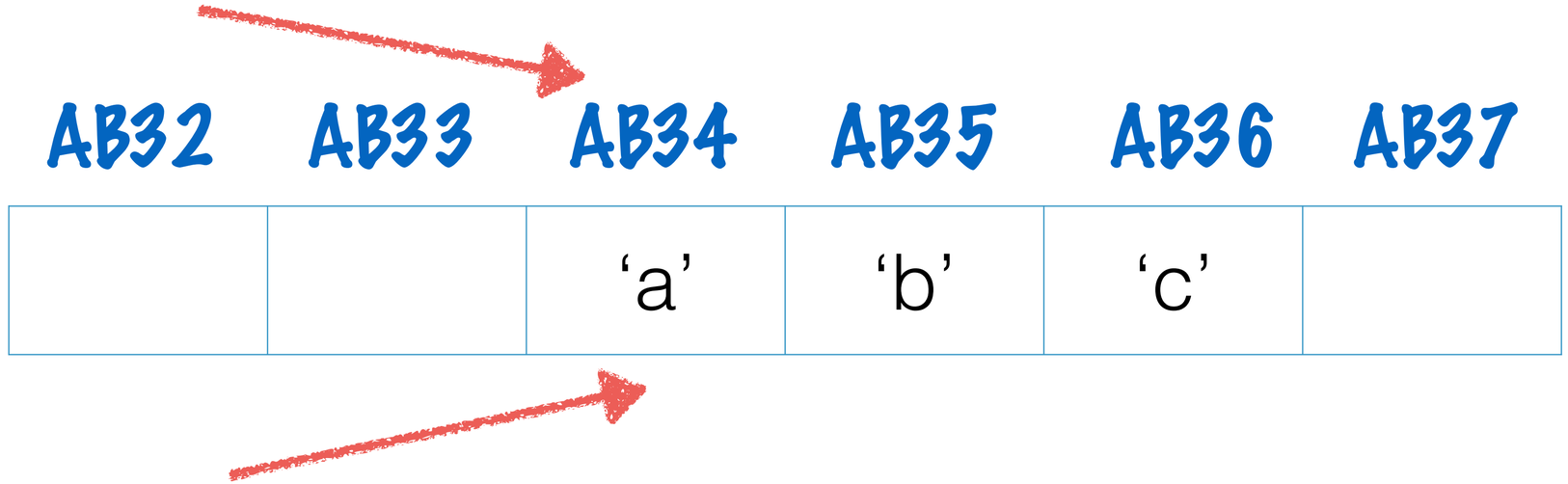
- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE **ADDRESS** THIS POINTER POINTS TO, I.E. THE `*`



- THE **CHARACTERS** IN THAT ADDRESS LOCATION, I.E. THE `char`

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

```
char *        const         char *        const char
              char *        const           * const
```

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

```
char *
```

```
const
char *
```

```
char *
const
```

```
const char
* const
```

char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO NEITHER

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

char *p1 = firstName;                    // non-const pointer, non-const data
cout << "Initial Value of p1 : " << p1 << endl;
p1[0] = 'Z';                              // can change the data
cout <<  "Value of p1 after changing data : " << p1 << endl;
p1 = lastName; // can change what is pointed to
cout <<  "Value of p1 after changing what it points to : " << p1 << endl;
```

char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO NEITHER

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

char *p1 = firstName;                    // non-const pointer, non-const data
cout << "Initial Value of p1 : " << p1 << endl;
p1[0] = 'Z';                                // can change the data
cout <<  "Value of p1 after changing data : " << p1 << endl;
p1 = lastName; // can change what is pointed to
cout <<  "Value of p1 after changing what it points to : " << p1 << endl;
```

char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO NEITHER

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

char *p1 = firstName;                    // non-const pointer, non-const data
cout << "Initial Value of p1 : " << p1 << endl;
p1[0] = 'Z';                                    // can change the data
cout <<  "Value of p1 after changing data : " << p1 << endl;
p1 = lastName; // can change what is pointed to
cout <<  "Value of p1 after changing what it points to : " << p1 << endl;
```

`char *`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` APPLIES TO NEITHER

```
Initial Value of p1 : Vitthal
Value of p1 after changing data : Zitthal
Value of p1 after changing what it points to : Srinivasan
```

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

```
char *
```

```
const
char *
```

```
char *
const
```

```
const char
* const
```

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

`const char *`

`char * const`

`const char * const`

const char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO THE CHARACTERS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

const char *p2 = firstName;          // non-const pointer,const data
cout << "Initial Value of p2 : " << p2 << endl;
//p2[0] = 'Z';                              // can NOT change the data
cout <<  "Value of p2 after changing data : " << p2 << endl;
p2 = lastName; // can change what is pointed to
cout <<  "Value of p2 after changing what it points to : " << p2 << endl;
```

const char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO THE CHARACTERS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

const char *p2 = firstName;          // non-const pointer,const data
cout << "Initial Value of p2 : " << p2 << endl;
//p2[0] = 'Z';                                // can NOT change the data
cout <<  "Value of p2 after changing data : " << p2 << endl;
p2 = lastName; // can change what is pointed to
cout <<  "Value of p2 after changing what it points to : " << p2 << endl;
```

const char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO THE CHARACTERS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

const char *p2 = firstName;          // non-const pointer,const data
cout << "Initial Value of p2 : " << p2 << endl;
//p2[0] = 'Z';                                  // can NOT change the data
Example32.cpp:19:9: error: read-only variable is not assignable   ;
  p2[0] = 'Z';                                 // can NOT change the data
  ~~~~~ ^
1 error generated.
                                                        << p2 << endl;
```

const char *

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

const APPLIES TO THE CHARACTERS ALONE

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Initial Value of p2 : Vitthal
Value of p2 after changing data : Vitthal
Value of p2 after changing what it points to : Srinivasan
```

```
cout <<  "Initial value of p2 :   " << p2 << endl;
//p2[0] = 'Z';                              // can NOT change the data
cout <<  "Value of p2 after changing data : " << p2 << endl;
p2 = lastName; // can change what is pointed to
cout <<  "Value of p2 after changing what it points to : " << p2 << endl;
```

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

`const char *`

`char * const`

`const char * const`

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS LOCATION, I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

✓ `const char *`

`char * const`

`const char * const`

char * const

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

const APPLIES TO THE ADDRESS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

char * const p3 = firstName;          // const pointer, non-const data
cout << "Initial Value of p3 : " << p3 << endl;
p3[0] = 'Z';                                  // can change the data
cout <<  "Value of p3 after changing data : " << p3 << endl;
//p3 = lastName; // can NOT change what is pointed to
cout <<  "Value of p3 after changing what it points to : " << p3 << endl;
```

char * const

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

const APPLIES TO THE ADDRESS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";

char * const p3 = firstName;        // const pointer, non-const data
cout << "Initial Value of p3 : " << p3 << endl;
p3[0] = 'Z';                                // can change the data
cout <<  "Value of p3 after changing data : " << p3 << endl;
//p3 = lastName; // can NOT change what is pointed to
cout <<  "Value of p3 after changing what it points to : " << p3 << endl;
```

char * const

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

const APPLIES TO THE ADDRESS ALONE

```
Initial Value of p3 : Vitthal
Value of p3 after changing data : Zitthal
Value of p3 after changing what it points to : Zitthal
cout << "Initial Value of p3 : " << p3 << endl;
p3[0] = 'Z';                                    // can change the data
cout <<  "Value of p3 after changing data : " << p3 << endl;
//p3 = lastName; // can NOT change what is pointed to
cout <<  "Value of p3 after changing what it points to : " << p3 << endl;
```

char * const

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

const APPLIES TO THE ADDRESS ALONE

```
char firstName[] = "Vitthal";
char lastName[] = "Srinivasan";
```

```
Example32.cpp:31:6: error: read-only variable is not assignable
  p3 = lastName; // can NOT change what is pointed to
  ~~ ^
1 error generated.
```

```
cout <<  "Value of p3 after changing data : " << p3 << endl;
//p3 = lastName; // can NOT change what is pointed to
cout <<  "Value of p3 after changing what it points to : " << p3 << endl;
```

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

✓ `const char *`

`char * const`

`const char * const`

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

✓ `const char *`

✓ `char * const`

`const char * const`

```
const char
  * const
```

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

`const` APPLIES TO BOTH THE CHARACTERS AND THE ADDRESS

YOU CAN'T CHANGE ANYTHING ABOUT SUCH A VARIABLE!

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

✓ `const char *`

✓ `char * const`

`const char * const`

EXAMPLE 32 UNDERSTAND THE DIFFERENCE BETWEEN `const char*` AND `char* const`

THERE ARE TWO PORTIONS IN A `char*`

- THE ADDRESS THIS POINTER POINTS TO,I.E. THE `*`
- THE CHARACTERS IN THAT ADDRESS,I.E. THE `char`

`const` COULD APPLY TO EITHER, BOTH OR NONE OF THESE

✓ `char *`

✓ `const char *`

✓ `char * const`

✓ `const char * const`

EXAMPLE 33

MARKING A MEMBER FUNCTION OF A CLASS `const`

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS const

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED const, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED const, CAN NOT CALL ANY OTHER MEMBER FUNCTION THAT IS NOT MARKED const

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS const

```
class Student
{
private:
  string studentName;
public:
  Student(const char* name) : studentName(name)
  {
    cout << "Initialized string to: " << studentName << endl;
  }

  void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can not modify an member data!
    //studentName = string("Srinivasan");
  }
};
```

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS const

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED const, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

```
void print() const
{
  cout << "StudentName:" << studentName << endl;
  // This member function is marked const, so it can not modify an member data!
  //studentName = string("Srinivasan");
}
```

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

```
void print() const
{
  cout << "StudentName:" << studentName << endl;
  // This member function is marked const, so it can not modify an member data!
  //studentName = string("Srinivasan");
}
```

EXAMPLE 33 MARKING A MEMBER FUNCTION OF A CLASS `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

```
void print() const
{
  cout << "StudentName:" << studentName << endl;
  // This member function is marked const, so it can not modify an member data!
    studentName = string("Srinivasan");
}
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example33.cpp
Example33.cpp:20:17: error: no viable overloaded '='
    studentName = string("Srinivasan");
    ~~~~~~~~~~~ ^ ~~~~~~~~~~~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1363:19: note: candidate funct
      viable: 'this' argument has type 'const string' (aka 'const basic_string<char, char_traits<char>, allocator<char> >'), but method is not
      const
    basic_string& operator=(const basic_string& __str);
                  ^
```

WHAT THE ?@#$ IS THIS ERROR MESSAGE TRYING TO SAY?

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example33.cpp
Example33.cpp:20:17: error: no viable overloaded '='
    studentName = string("Srinivasan");
    ~~~~~~~~~~~ ^ ~~~~~~~~~~~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1363:19: note: candidate funct
      viable: 'this' argument has type 'const string' (aka 'const basic_string<char, char_traits<char>, allocator<char> >'), but method is not
      const
    basic_string& operator=(const basic_string& __str);
                  ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1370:45: note: candidate funct
      viable: 'this' argument has type 'const string' (aka 'const basic_string<char, char_traits<char>, allocator<char> >'), but method is not
      const
    _LIBCPP_INLINE_VISIBILITY basic_string& operator=(const value_type* __s) {return assign(__s);}
                                            ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:1371:19: note: candidate funct
      viable: 'this' argument has type 'const string' (aka 'const basic_string<char, char_traits<char>, allocator<char> >'), but method is not
      const
    basic_string& operator=(value_type __c);
                  ^
1 error generated.
```

C++ ERROR MESSAGES ARE OFTEN IMPENETRABLE, UNFORTUNATELY.

EXAMPLE 34
MARKING A MEMBER VARIABLE OF A CLASS `mutable`

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS `mutable`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

BUT A MEMBER FUNCTION MARKED `const` CAN STILL MODIFY ANY MEMBER VARIABLE MARKED `mutable`

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS mutable

A MEMBER FUNCTION MARKED const CAN STILL MODIFY ANY MEMBER VARIABLE MARKED mutable

```
class Student
{
private:
  mutable string studentName;
public:
  Student(const char* name) : studentName(name)
  {
    cout << "Initialized string to: " << studentName << endl;
  }

  void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can not modify an member data -
    // UNLESS that member data is marked mutable.
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
  }
};
```

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS mutable

A MEMBER FUNCTION MARKED const CAN STILL MODIFY ANY MEMBER VARIABLE MARKED mutable

```
class Student
{
private:
  mutable string studentName;
public:
  Student(const char* name) : studentName(name)
  {
    cout << "Initialized string to: " << studentName << endl;
  }

  void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can not modify an member data -
    // UNLESS that member data is marked mutable.
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
  }
};
```

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS `mutable`

A MEMBER FUNCTION MARKED `const` CAN STILL MODIFY ANY MEMBER VARIABLE MARKED `mutable`

```
class Student
{
private:
  mutable string studentName;
public:
  Student(const char* name) : studentName(name)
  {
    cout << "Initialized string to: " << studentName << endl;
  }

  void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can not modify an member data -
    // UNLESS that member data is marked mutable.
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
  }
};
```

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS mutable

A MEMBER FUNCTION MARKED const CAN STILL MODIFY ANY MEMBER VARIABLE MARKED mutable

```
class Student
{
private:
  mutable string studentName;
public:
  Student(const char* name) : studentName(name)
  {
    cout << "Initialized string to: " << studentName << endl;
  }

  void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can not modify an member data -
    // UNLESS that member data is marked mutable.
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
  }
};
```

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS `mutable`

A MEMBER FUNCTION MARKED `const` CAN STILL MODIFY ANY MEMBER VARIABLE MARKED `mutable`

```
class Student
{
private:
  mutable string studentName;
public:
```

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example34.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Initialized string to: Vitthal
 StudentName:Vitthal
 Oh, btw, now StudentName:Srinivasan
```

```
    // This member function is marked const, so it can not modify an member data -
    // UNLESS that member data is marked mutable.
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
  }
};
```

EXAMPLE 34 MARKING A MEMBER VARIABLE OF A CLASS `mutable`

A MEMBER FUNCTION MARKED `const` CAN STILL MODIFY ANY MEMBER VARIABLE MARKED `mutable`

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example34.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Initialized string to: Vitthal
StudentName:Vitthal
Oh, btw, now StudentName:Srinivasan
```

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

EXAMPLE 35 OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

EXAMPLE 35 OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, CAN NOT CALL ANY OTHER MEMBER FUNCTION THAT IS NOT MARKED `const`

EXAMPLE 35 OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, WHICH MEANS THAT IT WILL NOT CHANGE ANY MEMBER VARIABLE OF THAT OBJECT

A MEMBER FUNCTION OF AN OBJECT CAN BE MARKED `const`, CAN NOT CALL ANY OTHER MEMBER FUNCTION THAT IS NOT MARKED `const`

TO GET AROUND THIS, C++ ALLOWS MEMBER FUNCTIONS TO BE OVERLOADED PURELY ON `const`

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON const

```
class Student
{
 void print() const

 void print()

  void changeStudentName()
```

C++ ALLOWS MEMBER FUNCTIONS TO BE OVERLOADED PURELY ON const

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

C++ ALLOWS MEMBER FUNCTIONS TO BE OVERLOADED PURELY ON `const`

```
class Student
{
void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can NOT
    // call a non-const method
    // changeStudentName();
  }

 void print()
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is NOT marked const, so it can
    // call a non-const method
    changeStudentName();
  }
```

```
void changeStudentName()
{
  studentName = string("Srinivasan");
```

A NON-CONST FUNCTION THAT ALTERS A MEMBER VARIABLE

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

C++ ALLOWS MEMBER FUNCTIONS TO BE OVERLOADED PURELY ON `const`

```
class Student
{
void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can NOT
    // call a non-const method
    // changeStudentName();
  }

 void print()
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is NOT marked const, so it can
    // call a non-const method
    changeStudentName();
  }

  void changeStudentName()
  {
    studentName = string("Srinivasan");
```

A CONST FUNCTION CAN NOT CALL A NON-CONST FUNCTION

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

C++ ALLOWS MEMBER FUNCTIONS TO BE OVERLOADED PURELY ON `const`

SO CREATE A NON-CONST VERSION INSTEAD!

```
class Student
{
void print() const
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is marked const, so it can NOT
    // call a non-const method
    // changeStudentName();
  }

 void print()
  {
    cout << "StudentName:" << studentName << endl;
    // This member function is NOT marked const, so it can
    // call a non-const method
    changeStudentName();
  }

  void changeStudentName()
  {
    studentName = string("Srinivasan");
    cout << "Oh, btw, now StudentName:" << studentName << endl;
```

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

SO WHEN WILL THE CONST VERSION BE CALLED VS THE NON-CONST VERSION?

CALLS THE CONST VERSION

```
const Student const_s;
Student nonconst_s;

const_s.print();

nonconst_s.print();
```

EXAMPLE 35

OVERLOADING A MEMBER FUNCTION OF A CLASS PURELY ON `const`

SO WHEN WILL THE CONST VERSION BE CALLED VS THE NON-CONST VERSION?

```
const Student const_s;
Student nonconst_s;

const_s.print();

nonconst_s.print();
```

CALLS THE NON-CONST VERSION

EXAMPLE 36
GETTING RID OF CONSTNESS USING
const_cast

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;

// Create a non-const Student& object but
// and assign const Student object - but with a const_cast!
Student& StudentThree = const_cast<Student &>(studentOne);
```

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;

// Create a non-const Student& object but
// and assign const Student object - but with a const_cast!
Student& StudentThree = const_cast<Student &>(studentOne);
```

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;

// Create a non-const Student& object but
// and assign const Student object - but with a const_cast!
Student& StudentThree = const_cast<Student &>(studentOne);
```

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example36.cpp
Example36.cpp:47:12: error: binding of reference to type 'Student' to a value of type 'const Student' drops quali
  Student& StudentTwo = studentOne;
           ^            ~~~~~~~~~~
1 error generated.
```

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;

// Create a non-const Student& object but
// and assign const Student object - but with a const_cast!
Student& StudentThree = const_cast<Student &>(studentOne);
```

EXAMPLE 36 GETTING RID OF CONSTNESS USING const_cast

```
const char name[50] = “Vitthal";

// Create a const Student object
const Student studentOne(name);

// Create a non-const Student& object
// and assign const Student object - Compiler Error!
Student& StudentTwo = studentOne;

// Create a non-const Student& object but
// and assign const Student object - but with a const_cast!
Student& StudentThree = const_cast<Student &>(studentOne);
```

const_cast IS THE FIRST C++ CAST WE ARE SEEING - IT HELPS GET RID OF CONSTNESS IF FOR SOME REASON WE NEED TO

EXAMPLE 37

PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

STUDENT

```
void SomeFunction(Student s) {



}
```

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

STUDENT

```
void SomeFunction(Student s) {



}
```

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION AS A FUNCTION ARGUMENT, A TEMPORARY VARIABLE IS CREATED

THIS TEMPORARY VARIABLE IS CONSTRUCTED (USING THE COPY CONSTRUCTOR), AND THEN MUST BE DESTRUCTED TOO (VIA THE DESTRUCTOR)

ALL OF THESE CONSTRUCTOR AND DESTRUCTOR CALLS ADD UP (REMEMBER THEY WILL CASCADE INTO THE MEMBER VARIABLES OF THAT OBJECT!)

IN A PERFORMANCE-SENSITIVE APPLICATION THIS WILL BECOME UNACCEPTABLE

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

IN A PERFORMANCE-SENSITIVE APPLICATION
THIS WILL BECOME UNACCEPTABLE

SLICING

THERE IS ANOTHER REASON TO PREFER PASSING FUNCTION ARGUMENTS AS CONST REFERENCES RATHER THAN BY VALUE, CALLED **SLICING**

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

IN A PERFORMANCE-SENSITIVE APPLICATION
THIS WILL BECOME UNACCEPTABLE

SLICING

THERE IS ANOTHER REASON TO PREFER PASSING
FUNCTION ARGUMENTS AS CONST REFERENCES
RATHER THAN BY VALUE, CALLED SLICING

SLICING HAS TO DO WITH INHERITANCE,
SO WE WILL GET TO IT LATER

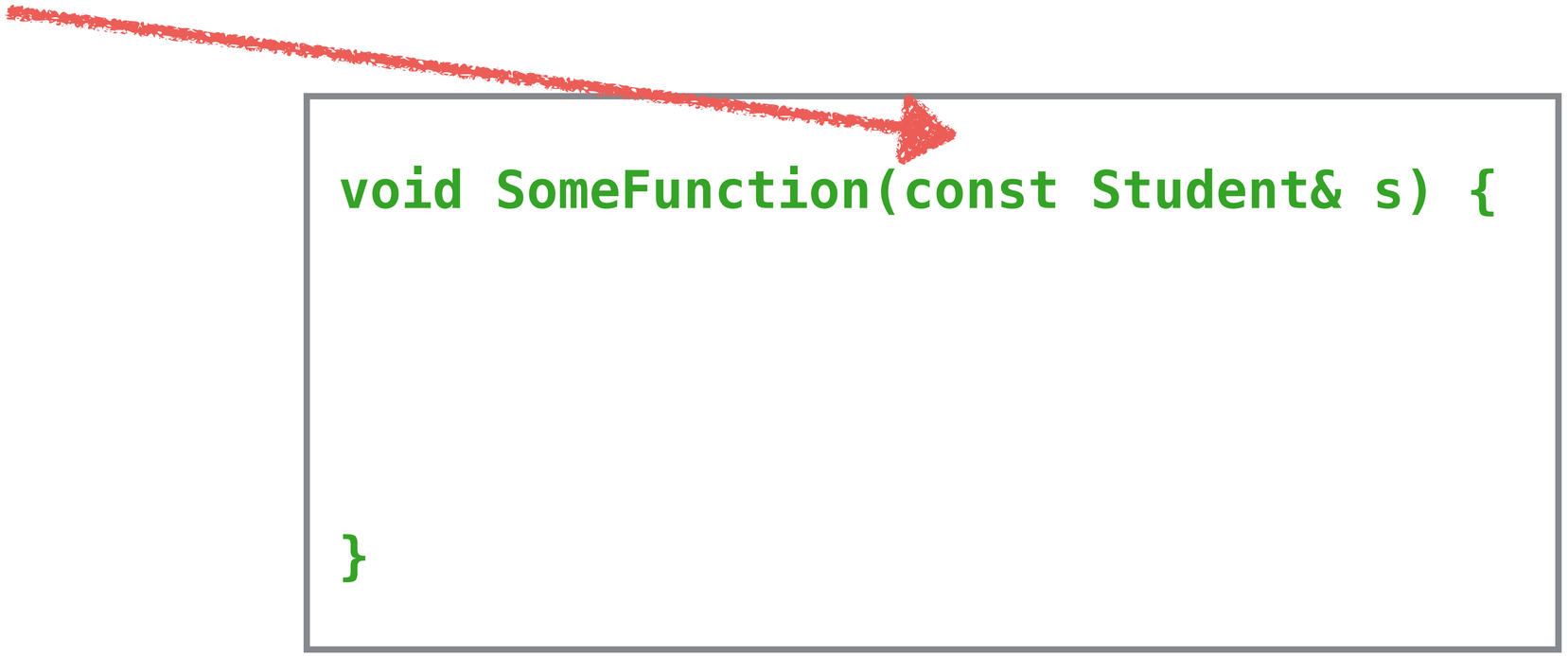
EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

INSTEAD, SIMPLY PASS THE FUNCTION ARGUMENTS AS CONST REFERENCES. NO CONSTRUCTOR OR DESTRUCTOR CALLS WILL RESULT

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

INSTEAD, SIMPLY PASS THE FUNCTION ARGUMENTS AS CONST REFERENCES. NO CONSTRUCTOR OR DESTRUCTOR CALLS WILL RESULT

STUDENT



EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

INSTEAD, SIMPLY PASS THE FUNCTION ARGUMENTS AS CONST REFERENCES. NO CONSTRUCTOR OR DESTRUCTOR CALLS WILL RESULT

STUDENT

```
void SomeFunction(const Student& s) {



}
```

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

INSTEAD, SIMPLY PASS THE FUNCTION ARGUMENTS AS CONST REFERENCES. NO CONSTRUCTOR OR DESTRUCTOR CALLS WILL RESULT

DON'T FORGET THE CONST!

ELSE THE FUNCTION MIGHT MISTAKENLY MODIFY SOMETHING IT SHOULD NOT!

EXAMPLE 37 PASSING FUNCTION PARAMETERS AS CONST REFERENCES RATHER THAN BY-VALUE

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}

};
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;


  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}

};
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;


  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

WITH A COPY CONSTRUCTOR AND DESTRUCTOR THAT PRINT OUT WHEN CALLED

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}

};
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;


  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon succeeds
  c.setRealPart(3.14);
}

void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

THEY DIFFER IN THE MANNER IN WHICH THEY TAKE IN AN OBJECT OF THE COMPLEX NUMBER CLASS

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

THEY DIFFER IN THE MANNER IN WHICH THEY TAKE IN AN OBJECT OF THE COMPLEX NUMBER CLASS

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

THEY DIFFER IN THE MANNER IN WHICH THEY TAKE IN AN OBJECT OF THE COMPLEX NUMBER CLASS

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

THEY DIFFER IN THE MANNER IN WHICH THEY TAKE IN AN OBJECT OF THE COMPLEX NUMBER CLASS

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

BUT THEY ALL FUNDAMENTALLY DO THE SAME THING - PRINT THE COMPLEX NUMBER, AND THEN TRY TO CHANGE IT

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

BUT THEY ALL FUNDAMENTALLY DO THE SAME THING - PRINT THE COMPLEX NUMBER, AND THEN TRY TO CHANGE IT

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

BUT THEY ALL FUNDAMENTALLY DO THE SAME THING - PRINT THE COMPLEX NUMBER, AND THEN TRY TO CHANGE IT

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}

};
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;


  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "After  passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;



  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
int main()
{
  ComplexNumber c(4,5);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

AND TESTS WHETHER THE CHANGES MADE INSIDE THE FUNCTION ARE STILL REFLECTED IN THE CALLING (MAIN) FUNCTION

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

1. A SIMPLE COMPLEX NUMBER CLASS

```
class ComplexNumber
{
private:
  float realPart;
  float complexPart;
public:
  ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
  {
    cout << "Inside the 2-argument constructor" << endl;
  }

  ComplexNumber(const ComplexNumber& rhs) :
     realPart(rhs.realPart), complexPart(rhs.complexPart)
  {
    cout << "Inside the copy constructor" << endl;
  }

  ~ComplexNumber()
  {
    cout << "Inside the destructor: realPart = " << realPart <<" complexPart = " << complexPart << endl;
  }

  float getRealPart() const { return realPart; }
  void  setRealPart(float r) { realPart = r;}

};
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}

void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}


void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

3. A MAIN METHOD THAT CALLS THOSE 3 METHODS

```
int main()
{
  ComplexNumber c(3.4,5.3);
  cout << "Before passing by value "<< c.getRealPart() << endl;
  printComplexNumberPassByValue(c);
  cout << "Aftere passing by value "<< c.getRealPart() << endl;

  cout << "Before passing by non-const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByReference(c);
  cout << "After passing by non-const reference "<< c.getRealPart() << endl;


  cout << "Before passing by const reference "<< c.getRealPart() << endl;
  printComplexNumberPassByConstReference(c);
  cout << "After passing by const reference "<< c.getRealPart() << endl;

}
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

WHEN WE PASS BY VALUE -

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY VALUE -

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Inside the 2-argument constructor
 Before passing by value 5.3
 Inside the copy constructor
 Inside printComplexNumber
 5.3
 Inside the destructor: realPart = 3.14 complexPart = 3.4
 Aftere passing by value 5.3
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY VALUE -

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Inside the 2-argument constructor
 Before passing by value 5.3
 Inside the copy constructor
 Inside printComplexNumber
 5.3
 Inside the destructor: realPart = 3.14 complexPart = 3.4
 Aftere passing by value 5.3
```

AN EXTRA OBJECT IS CREATED..

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon
  c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY VALUE -

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Inside the 2-argument constructor
Before passing by value 5.3
Inside the copy constructor
Inside printComplexNumber
5.3
Inside the destructor: realPart = 3.14 complexPart = 3.4
Aftere passing by value 5.3
```

AND THIS OBJECT GETS MODIFIED, SO THE ORIGINAL OBJECT IS NOT CHANGED

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

AN EXTRA OBJECT IS CREATED..

WHEN WE PASS BY VALUE -

AND THIS OBJECT GETS MODIFIED, SO THE ORIGINAL OBJECT IS NOT CHANGED

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

WHEN WE PASS BY NON-CONST REFERENCE -

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY NON-CONST REFERENCE -

```
Before passing by non-const reference 5.3
Inside printComplexNumber
5.3
After passing by non-const reference 3.14
```

NO EXTRA OBJECT IS CREATED..

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByReference(ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - succeeds!
  c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY NON-CONST REFERENCE -

```
Before passing by non-const reference 5.3
Inside printComplexNumber
5.3
After passing by non-const reference 3.14
```

CHANGE MADE INSIDE THE FUNCTION IS REFLECTED IN THE CALLING CODE

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

AN EXTRA OBJECT IS CREATED..

WHEN WE PASS BY VALUE -

AND THIS OBJECT GETS MODIFIED, SO THE ORIGINAL OBJECT IS NOT CHANGED

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

NO EXTRA OBJECT IS CREATED..

WHEN WE PASS BY NON-CONST REFERENCE -

CHANGE MADE INSIDE THE FUNCTION IS REFLECTED IN THE CALLING CODE

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

WHEN WE PASS BY CONST REFERENCE -

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY CONST REFERENCE -

```
Before passing by const reference 3.14
Inside printComplexNumber
3.14
After passing by const reference 3.14
```

NO EXTRA OBJECT IS CREATED..

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
{
  cout << "Inside printComplexNumber" << endl;
  cout << c.getRealPart() << endl;
  // Attempt to modify the function argument inside the functon - Compiler catches it!
  // c.setRealPart(3.14);
}
```

2. THREE FUNCTIONS THAT TAKE IN AN OBJECT OF THAT CLASS

WHEN WE PASS BY CONST REFERENCE -

```
Before passing by const reference 3.14
Inside printComplexNumber
3.14
After passing by const reference 3.14
```

NO CHANGE IS POSSIBLE IN THE FUNCTION - COMPILER REJECTS IT, SO HAS TO BE COMMENTED OUT!

THERE IS A LOT GOING ON HERE, SO OUR EXAMPLE HAS 3 COMPONENTS

```
void printComplexNumberPassByValue(ComplexNumber c)
```

AN EXTRA OBJECT IS CREATED..

WHEN WE PASS BY VALUE -

AND THIS OBJECT GETS MODIFIED, SO THE ORIGINAL OBJECT IS NOT CHANGED

```
void printComplexNumberPassByReference(ComplexNumber& c)
```

NO EXTRA OBJECT IS CREATED..

WHEN WE PASS BY NON-CONST REFERENCE -

CHANGE MADE INSIDE THE FUNCTION IS REFLECTED IN THE CALLING CODE

```
void printComplexNumberPassByConstReference(const ComplexNumber& c)
```

NO EXTRA OBJECT IS CREATED..

WHEN WE PASS BY CONST REFERENCE -

NO CHANGE IS POSSIBLE IN THE FUNCTION - COMPILER REJECTS IT