TEMPLATES

REMEMBER HOW WE WROTE A FUNCTION TO SWAP THE VALUE OF 2 `int` VARIABLES?

```
void swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

REMEMBER HOW WE WROTE A FUNCTION TO SWAP THE VALUE OF 2 `int` VARIABLES?

```
void swap(int& a,int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

NOW, SAY WE NEED TO WRITE ANOTHER FUNCTION TO SWAP THE VALUE OF 2 `string` VARIABLES

```
void swap(string& a,string& b)
{
  string temp = a;
  a = b;
  b = temp;
}
```

REMEMBER HOW WE WROTE A FUNCTION TO SWAP THE VALUE OF 2 int VARIABLES?

```
void swap(int& a,int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

WE BASICALLY WROTE THE EXACT SAME CODE TWICE, ONCE FOR `int` AND ONCE FOR `string`

NOW SAY WE NEED TO WRITE ANOTHER FUNCTION TO SWAP THE VALUE OF 2 string VARIABLES

```
void swap(string& a,string& b)
{
  string temp = a;
  a = b;
  b = temp;
}
```

WE BASICALLY WROTE THE EXACT SAME CODE TWICE, ONCE FOR `int` AND ONCE FOR `string`

THIS IS TERRIBLE FOR ALL KINDS OF REASONS

TO CHANGE ONE, CHANGE ALL

VERY POOR PROGRAMMING PRACTICE TO COPY-PASTE CODE

TEMPLATES TO THE RESCUE

C++ SUPPORTS TEMPLATES TO EXACTLY REMEDY THIS SITUATION

TEMPLATES

ARE A FORM OF GENERIC PROGRAMMING THAT BECAME VERY WIDESPREAD THANKS TO C++

TEMPLATES IN C++ ARE ALSO THE FOUNDATION OF THE STANDARD-TEMPLATE-LIBRARY (STL)

TEMPLATES AND STL ARE VERY WIDELY USED IN PRODUCTION C++ CODE.

TEMPLATES

ONCE YOU UNDERSTAND HOW TEMPLATES WORK THEY MIGHT INITIALLY (FOR A FEW SECONDS) SEEM LIKE C MACROS

BANISH THE THOUGHT. C MACROS ARE DUMB PRE-COMPILER TEXT SUBSTITUTIONS

TEMPLATES ARE A SOPHISTICATED FORM OF COMPILE-TIME POLYMORPHISM

EXAMPLE 63: CREATE A SIMPLE TEMPLATED FUNCTION

EXAMPLE 63: CREATE A SIMPLE TEMPLATED FUNCTION

WE BASICALLY WROTE THE EXACT SAME CODE TWICE, ONCE FOR `int` AND ONCE FOR `string`

THIS IS TERRIBLE FOR ALL KINDS OF REASONS

TO CHANGE ONE, CHANGE ALL

VERY POOR PROGRAMMING PRACTICE TO COPY-PASTE CODE

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
void swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

```
void swap(string& a,string& b)
{
   string temp = a;
   a = b;
   b = temp;
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
void swap(T& a,T& b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

```
void swap(T& a,T& b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
void swap(T& a,T& b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
    return 0;
}
```

THIS LINE THAT TELLS THE COMPILER THAT WE ARE CREATING A FUNCTION TEMPLATE

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
    return 0;
}
```

THIS TELLS THE COMPILER THAT THE TYPE TO OPERATE ON IS CALLED T

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
  T temp = a;
  a = b;
  b = temp;
  return 0;
}
```

INSIDE THE BODY, JUST USE T AS IF IT WERE INT, OR STRING OR WHATEVER.

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
    return 0;
}
```

T IS CALLED THE TEMPLATE PARAMETER (COULD BE CALLED ANYTHING - LIKE A FUNCTION PARAMETER, THIS IS MERELY A PLACEHOLDER)

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
int swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
    return 0;
}
```

YOU COULD ALSO SAY "typename T" INSTEAD OF "class T"

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

THIS ENTIRE "THING" IS CALLED THE FUNCTION TEMPLATE

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

REMEMBER TWO TERMS HERE

TEMPLATES

T IS CALLED THE TEMPLATE PARAMETER

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

REMEMBER TWO TERMS HERE

TEMPLATES

REMEMBER TWO TERMS HERE

THIS ENTIRE "THING" IS CALLED THE FUNCTION TEMPLATE

T IS CALLED THE TEMPLATE PARAMETER

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

ACTUAL FUNCTIONS WILL ONLY BE CREATED BY THE COMPILER WHEN SOMEONE CALLS A FUNCTION NAMED SWAP

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS. IT ONLY CREATES A FUNCTION TEMPLATE

THEN THE C++ COMPILER WILL "INSTANTIATE THE TEMPLATE"

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

THIS HAPPENS AT COMPILE TIME, AND IS A FORM OF "COMPILE-TIME POLYMORPHISM"

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

THIS HAPPENS AT COMPILE TIME, AND IS A FORM OF "COMPILE-TIME POLYMORPHISM"

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

THE TEMPLATE ACTS AS A BLUEPRINT FOR A FUNCTION, LIKE A CLASS IS A BLUEPRINT FOR AN OBJECT

```
template<class T>
int swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY FUNCTIONS, IT ONLY CREATES A FUNCTION TEMPLATE

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A FUNCTION WHEN..

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A FUNCTION WHEN..

DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
int a = 5;
int b = 10;
swap(a,b);
```

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A FUNCTION WHEN..

DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
int a = 5;
int b = 10;
swap(a,b);
```

THEN, IT WILL INSTANTIATE THE TEMPLATE (STILL AT COMPILE-TIME)

I.E. GENERATE A FUNCTION LIKE THIS

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A FUNCTION WHEN..

DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
int a = 5;
int b = 10;
swap(a,b);
```

THEN, IT WILL INSTANTIATE THE TEMPLATE (STILL AT COMPILE-TIME)

I.E. GENERATE A FUNCTION LIKE THIS

```
int swap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOTE THAT THE SPECIFIC INSTANTIATION HAS NO MENTION OF ANY TEMPLATE AT ALL!

LATER, IF DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
string firstName(“Vitthal”);
string lastName(“Srinivasan”);
swap(firstName,lastName);
```

THEN, IT WILL AGAIN INSTANTIATE THE TEMPLATE - THIS TIME GENERATING A STRING VERSION OF THE FUNCTION

LATER, IF DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
string firstName(“Vitthal”);
string lastName(“Srinivasan”);
swap(firstName,lastName);
```

THEN, IT WILL AGAIN INSTANTIATE THE TEMPLATE - THIS TIME GENERATING A STRING VERSION OF THE FUNCTION

```
int swap(string& a, string& b)
{
   string temp = a;
   a = b;
   b = temp;
   return 0;
}
```

NOTE THAT THE SPECIFIC INSTANTIATION HAS NO MENTION OF ANY TEMPLATE AT ALL!

THE C++ COMPILER WILL GENERATE ONE FUNCTION FOR EACH TYPE THAT IT NEEDS TO

THIS CAN LEAD TO CODE BLOAT - A PROBLEM WITH HEAVILY TEMPLATED CODE

ALSO, THE COMPILER INFERS WHAT TYPE PARAMETER TO PASS IN

THE C++ COMPILER WILL GENERATE ONE FUNCTION FOR EACH TYPE THAT IT NEEDS TO

THIS CAN LEAD TO CODE BLOAT - A PROBLEM WITH HEAVILY TEMPLATED CODE

ALSO, THE COMPILER INFERS WHAT TYPE PARAMETER TO PASS IN

THE TYPE INFERENCE CAN SOMETIMES LEAD TO UNEXPECTED TYPE CONVERSIONS

THE C++ COMPILER WILL GENERATE ONE FUNCTION FOR EACH TYPE THAT IT NEEDS TO

THIS CAN LEAD TO CODE BLOAT - A PROBLEM WITH HEAVILY TEMPLATED CODE

ALSO, THE COMPILER INFERS WHAT TYPE PARAMETER TO PASS IN

THE TYPE INFERENCE CAN SOMETIMES LEAD TO UNEXPECTED TYPE CONVERSIONS

DESPITE THIS, OVERALL TEMPLATES ARE AN INCREDIBLY USEFUL PART OF C++

EXAMPLE 64: OVERRIDE THE DEFAULT INSTANTIATION FOR SOME SPECIFIC TYPE

EXAMPLE 64: OVERRIDE THE DEFAULT INSTANTIATION FOR SOME SPECIFIC TYPE

LET'S SAY WE WOULD LIKE TO WRITE A GENERIC COMPARE FUNCTION

IT WOULD COMPARE NUMBERS EXACTLY AS USUAL

BUT WHILE COMPARING STRINGS, IT WILL TRY AND CONVERT THEM TO NUMBERS FIRST IF POSSIBLE. IF NOT, IT WILL COMPARE AS STRINGS.

LET'S SAY WE WOULD LIKE TO WRITE A
GENERIC COMPARE FUNCTION

IT WOULD COMPARE NUMBERS EXACTLY AS USUAL

BUT WHILE COMPARING STRINGS, IT WILL TRY AND CONVERT THEM TO NUMBERS FIRST IF POSSIBLE. IF NOT, IT WILL COMPARE AS STRINGS.

IN OTHER WORDS, WE NEED TO OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

NOT A PROBLEM! ITS REALLY SIMPLE :-)

OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

DEFINE THE FUNCTION TEMPLATE AS USUAL

```
template<class T>
int smartCompare(const T& a, const T& b)
{
   if (a > b)
      return 1;
   if (a < b)
      return -1;
   return 0;
}
```

OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

DEFINE THE FUNCTION TEMPLATE AS USUAL

```
template<class T>
int smartCompare(const T& a, const T& b)
{
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
}
```

THEN, EXPLICITLY INSTANTIATE THE TEMPLATE FOR THE SPECIFIC TYPE THAT YOU CARE ABOUT

```
int smartCompare(const string& a, const string& b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }
```

OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

DEFINE THE FUNCTION TEMPLATE AS USUAL

```
template<class T>
int smartCompare(const T& a, const T& b)
{
```

THEN, EXPLICITLY INSTANTIATE THE TEMPLATE FOR THE SPECIFIC TYPE THAT YOU CARE ABOUT

```
int smartCompare(const string& a, const string& b)
{
```

NOW, WHEN THE C++ COMPILER COMES ACROSS A COMPARISON OF 2 STRINGS, IT WILL USE YOUR FUNCTION..

OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

DEFINE THE FUNCTION TEMPLATE AS USUAL

```
template<class T>
int smartCompare(const T& a, const T& b)
{
```

THEN, EXPLICITLY INSTANTIATE THE TEMPLATE FOR THE SPECIFIC TYPE THAT YOU CARE ABOUT

```
int smartCompare(const string& a, const string& b)
{
```

JUST BE CAREFUL TO NOT INCLUDE ANY TEMPLATE PARAMETER, OR TEMPLATE INFORMATION IN YOUR SPECIFIC INSTANTIATION. IT WOULD BE WRONG TO DO SO - WILL CONFUSE THE C++ COMPILER

OVERRIDE THE DEFAULT TEMPLATE IMPLEMENTATION FOR A SPECIFIC TYPE (STRINGS)

DEFINE THE FUNCTION TEMPLATE AS USUAL

```
template<class T>
int smartCompare(const T& a, const T& b)
{
```

THEN, EXPLICITLY INSTANTIATE THE TEMPLATE FOR THE SPECIFIC TYPE THAT YOU CARE ABOUT

```
int smartCompare(const string& a, const string& b)
{
```

NOW, WHEN THE C++ COMPILER COMES ACROSS A COMPARISON OF 2 STRINGS, IT WILL USE YOUR FUNCTION..

NOW, WHEN THE C++ COMPILER COMES ACROSS A COMPARISON OF 2 STRINGS, IT WILL USE YOUR FUNCTION..

```
string firstName("Vitthal");
string lastName("Srinivasan");
i = smartCompare(firstName,lastName);
```

WHILE FOR A COMPARISON OF 2 INTS, THE C++ COMPILER WILL INSTANTIATE THE FUNCTION TEMPLATE

```
int a = 5;
int b = 10;
int i = smartCompare(a,b);
cout << i << endl;
```

NOW, WHEN THE C++ COMPILER COMES ACROSS A COMPARISON OF 2 STRINGS, IT WILL USE YOUR FUNCTION..

```
string firstName("Vitthal");
string lastName("Srinivasan");
i = smartCompare(firstName,lastName);
```

EXPLICIT INSTANTIATION FOUND - USE THAT VERSION

WHILE FOR A COMPARISON OF 2 INTS, THE C++ COMPILER WILL INSTANTIATE THE FUNCTION TEMPLATE

```
int a = 5;
int b = 10;
int i = smartCompare(a,b);
cout << i << endl;
```

NO EXPLICIT INSTANTIATION FOUND - C++ COMPILER WILL INSTANTIATE FUNCTION TEMPLATE

NOW, WHEN THE C++ COMPILER COMES ACROSS A COMPARISON OF 2 STRINGS, IT WILL USE YOUR FUNCTION..

```
string firstName("Vitthal");
string lastName("Srinivasan");
i = smartCompare(firstName,lastName);
```

EXPLICIT INSTANTIATION FOUND - USE THAT VERSION

```
int smartCompare(const string& a, const string& b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
```

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string& a, const string& b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;



}
```

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string& a, const string& b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;



}
```

THE NAME AND SIGNATURE OF THE FUNCTION TALLIES WITH THE FUNCTION TEMPLATE

```
template<class T>
int smartCompare(const T& a, const T& b)
{
```

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string& a, const string& b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
    // try and convert both strings to ints.
    // if possible – compare the 2 strings as numbers
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;



}
```

WHY? BECAUSE IT CAN BE REALLY ANNOYING TO GET NUMBERS SORTED AS STRINGS - LEXICOGRAPHICAL ORDER AND NUMERIC ORDER DON'T ALWAYS MATCH :-)

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
try {
  x = std::stoi (a,&sz);
 }
 catch(...) {
   cout << "Conversion failed " << a << endl;
   convertStringToInt = false;
 }
```

OUR FIRST `try/catch` BLOCK!

BASICALLY - WE `try` TO CONVERT A STRING TO A NUMBER USING THE `stoi` FUNCTION. IF AN ERROR RESULTS, WE `catch` THAT ERROR, AND CONCLUDE THAT THE CONVERSION IS NOT POSSIBLE

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string &a, const string &b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible – compare the 2 strings as numbers
  try {
    x = std::stoi (a,&sz);
   }
   catch(...) {
     cout << "Conversion failed " << a << endl;
     convertStringToInt = false;
   }
  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
```

WE try TO CONVERT A STRING TO A NUMBER

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(string a, string b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible – compare the 2 strings as numbers
  try {
    x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }
  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints " << x << ", " << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
```

WE try TO CONVERT A STRING TO A NUMBER USING THE `stoi` FUNCTION.

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string &a, const string &b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
    x = std::stoi (a,&sz);
   }
   catch(...) {
     cout << "Conversion failed " << a << endl;
     convertStringToInt = false;
   }
  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << ", " << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
```

IF AN ERROR RESULTS, WE `catch` THAT ERROR,

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string &a, const string &b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
    x = std::stoi (a,&sz);
   }
   catch(...) {
     cout << "Conversion failed " << a << endl;
     convertStringToInt = false;
   }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << ", " << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
```

IF AN ERROR RESULTS, WE `catch` THAT ERROR,

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string &a, const string &b)
{
  int x,y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible – compare the 2 strings as numbers
  try {
    x = std::stoi (a,&sz);
   }
   catch(...) {
     cout << "Conversion failed " << a << endl;
     convertStringToInt = false;
   }
  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }
  if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
  // if not, then compare as strings after all
  if (a > b)
    return 1;
  if (a < b)
    return -1;
  return 0;
```

IF AN ERROR RESULTS, WE `catch` THAT ERROR, AND CONCLUDE THAT THE CONVERSION IS NOT POSSIBLE

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
try {
  x = std::stoi (a,&sz);
 }
 catch(...) {
   cout << "Conversion failed " << a << endl;
   convertStringToInt = false;
 }
```

OUR FIRST `try/catch` BLOCK!

BASICALLY - WE `try` TO CONVERT A STRING TO A NUMBER USING THE `stoi` FUNCTION. IF AN ERROR RESULTS, WE `catch` THAT ERROR, AND CONCLUDE THAT THE CONVERSION IS NOT POSSIBLE

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

OUR SECOND `try/catch` BLOCK!

```
try {
  y = std::stoi (b,&sz);
}
catch(...) {
   cout << "Conversion failed " << b << endl;
   convertStringToInt = false;
}
```

DO THE SAME FOR THE SECOND STRING PASSED IN

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

IF BOTH STRINGS WERE CONVERTED FINE TO INTS, THEN USE THE SMART COMPARE TEMPLATE FUNCTION (BUT FOR INTS!)

```
if (convertStringToInt == true) {
   cout << "Converted both strings to ints.." << x << "," << y << endl;
   return smartCompare(x,y);
}
// if not, then compare as strings after all
if (a > b)
   return 1;
if (a < b)
   return -1;
return 0;
```

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

IF BOTH STRINGS WERE CONVERTED FINE TO INTS, THEN USE THE SMART COMPARE TEMPLATE FUNCTION (BUT FOR INTS!)

```
if (convertStringToInt == true) {
   cout << "Converted both strings to ints.." << x << "," << y << endl;
   return smartCompare(x,y);
}
// if not, then compare as strings after all
if (a > b)
   return 1;
if (a < b)
   return -1;
return 0;
```

THIS IS AN INTERESTING FUNCTION, SO LET'S UNDERSTAND IT LINE-BY-LINE

```
int smartCompare(const string& a, const string& b)
{
  int x = 0, y = 0;
  bool convertStringToInt = true;
  std::string::size_type sz;   // alias of size_t
  // try and convert both strings to ints.
  // if possible - compare the 2 strings as numbers
  try {
   x = std::stoi (a,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << a << endl;
    convertStringToInt = false;
  }

  try {
   y = std::stoi (b,&sz);
  }
  catch(...) {
    cout << "Conversion failed " << b << endl;
    convertStringToInt = false;
  }

    if (convertStringToInt == true) {
    cout << "Converted both strings to ints.." << x << "," << y << endl;
    return smartCompare(x,y);
  }
```

IF NOT, NEVER MIND, LEXICOGRAPHICAL COMPARISON IT IS!

```
  // if not, then compare as strings after all
  if (a > b)
     return 1;
  if (a < b)
     return -1;
  return 0;
```

```
}
```

```
int a = 5;
int b = 10;
int i = smartCompare(a,b);
cout << i << endl;


string firstName("Vitthal");
string lastName("Srinivasan");
i = smartCompare(firstName,lastName);
cout << i << endl;

i = smartCompare(string("100"),string("203"));
cout << i << endl;
```

RUN THE CODE ATTACHED WITH THIS EXAMPLE AND SEE WHAT THIS PRINTS!

EXAMPLE 65: CREATE A TEMPLATED CLASS (A SMART POINTER, NO LESS)

EXAMPLE 65: CREATE A TEMPLATED CLASS (A SMART POINTER, NO LESS)

EVERYTHING THAT WE JUST SAID ABOUT TEMPLATED FUNCTIONS IS ALSO TRUE ABOUT TEMPLATED CLASSES

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 CLASSES IS THE TYPE THAT THEY OPERATE ON

```
public class Pair
{
  private:
  int first;
  int second;
  public:
  ...
}
```

```
public class Pair
{
  private:
  string first;
  string second;
  public:
  ...
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
public class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

```
public class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
public class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
    T first;
    T second;
  ...
};
```

THIS LINE THAT TELLS THE COMPILER THAT WE ARE CREATING A FUNCTION TEMPLATE

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
    T first;
    T second;
  public:
    ...
}
```

THIS TELLS THE COMPILER THAT THE TYPE TO OPERATE ON IS CALLED T

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

INSIDE THE BODY, JUST USE T AS IF IT WERE INT, OR STRING OR WHATEVER.

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
    T first;
    T second;
  public:
    ...
};
```

T IS CALLED THE TEMPLATE PARAMETER (COULD BE CALLED ANYTHING - LIKE A FUNCTION PARAMETER, THIS IS MERELY A PLACEHOLDER)

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

ALL THAT VARIES ACROSS THESE 2 FUNCTIONS IS THE TYPE THAT THEY OPERATE ON

```
template<class T>
class Pair
{
  private:
    T first;
    T second;
  public:
    ...
};
```

YOU COULD ALSO SAY "typename T" INSTEAD OF "class T"

JUST SPECIFY THE TYPE AS A TEMPLATE PARAMETER!

TEMPLATES

THIS ENTIRE "THING" IS CALLED THE CLASS TEMPLATE

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

REMEMBER TWO TERMS HERE

TEMPLATES

T IS CALLED THE TEMPLATE PARAMETER

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

REMEMBER TWO TERMS HERE

TEMPLATES

REMEMBER TWO TERMS HERE

THIS ENTIRE "THING" IS CALLED THE **CLASS TEMPLATE**

T IS CALLED THE **TEMPLATE PARAMETER**

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

ACTUAL CLASSES WILL ONLY BE CREATED BY THE COMPILER WHEN SOMEONE INSTANTIATES AN OBJECT OF THE TEMPLATED CLASS

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

THEN THE C++ COMPILER WILL "INSTANTIATE THE CLASS FROM THE TEMPLATE"

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

THIS HAPPENS AT COMPILE TIME, AND IS A FORM OF "COMPILE-TIME POLYMORPHISM"

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

THIS HAPPENS AT COMPILE TIME, AND IS A FORM OF "COMPILE-TIME POLYMORPHISM"

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

THE TEMPLATE ACTS AS A BLUEPRINT FOR THE CLASS, LIKE THE CLASS IS A BLUEPRINT FOR THE OBJECT!

```
template<class T>
class Pair
{
  private:
  T first;
  T second;
  public:
  ...
}
```

NOW, HERE'S THE THING: WRITING THIS CODE DOES NOT ACTUALLY CREATE ANY CLASSES, IT ONLY CREATES A CLASS TEMPLATE

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A CLASS WHEN..

THE C++ COMPILER WILL ONLY ACTUALLY CREATE A CLASS WHEN..

DURING COMPILATION, IT ENCOUNTERS CODE LIKE THIS

```
Pair<String> s;
```

WHICH INSTANTIATES AN OBJECT OF THE TEMPLATED CLASS

LET'S USE THIS IDEA OF A TEMPLATED CLASS TO BUILD SOMETHING

REALLY POWERFUL

A SMART POINTER

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING COUNT

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

BTW, ALL OF JAVA IS BUILT USING A MECHANISM TO HAVE POINTERS BE AUTOMATICALLY DELETED

BTW, ALL OF JAVA IS BUILT USING A MECHANISM TO HAVE POINTERS BE AUTOMATICALLY DELETED

ITS PROBABLY FAIR TO SAY THAT THE #1 REASON WHY JAVA BECAME SO POPULAR WAS ITS MEMORY MANAGEMENT

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

BTW, ALL OF JAVA IS BUILT USING A MECHANISM TO HAVE POINTERS BE AUTOMATICALLY DELETED

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

WHEN THE REFERENCE COUNT BECOMES ZERO, THE SMART POINTER DELETES ITSELF

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

WHEN THE REFERENCE COUNT BECOMES ZERO, THE SMART POINTER DELETES ITSELF

OUR SMART POINTER CAN ALSO FILL IN AS A POINTER OF ANY TYPE

OUR SMART POINTER CAN ALSO FILL IN AS A POINTER OF ANY TYPE

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

```
template < typename T > class SmartPointer
{
private:
   T*     pData;          // pointer
   RefCount* reference; // Reference count
```

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

```
template < typename T > class SmartPointer
{
private:
  T*     pData;          // pointer
  RefCount* reference; // Reference count
```

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

```
T& operator* ()
{
   return *pData;
}
```

```
T* operator-> ()
{
   return pData;
}
```

```
template < typename T > class SmartPointer
{
private:
  T*     pData;          // pointer
  RefCount* reference; // Reference count
```

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

```
  T& operator* ()
  {
    return *pData;
  }
```

```
T* operator-> ()
{
  return pData;
}
```

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

```
template < typename T > class SmartPointer
{
private:
   T*     pData;          // pointer
   RefCount* reference; // Reference count
```

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

```
template < typename T > class SmartPointer
{
private:
  T*     pData;       // pointer
  RefCount* reference; // Reference count
```

```
class RefCount
{
private:
  int count; // Reference count

public:
  void AddRef()
  {
    // Increment the reference count
    count++;
  }

  int Release()
  {
    // Decrement the reference count and
    // return the reference count.
    return --count;
  }
};
```

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

```
template < typename T > class SmartPointer
{
private:
  T*       pData;       // pointer
  RefCount* reference; // Reference count
```

```
class RefCount
{
private:
  int count; // Reference count

public:
  void AddRef()
  {
    // Increment the reference count
    count++;
  }

  int Release()
  {
    // Decrement the reference count and
    // return the reference count.
    return --count;
  }
};
```

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

```
template < typename T > class SmartPointer
{
private:
  T*       pData;       // pointer
  RefCount* reference; // Reference count
```

```
class RefCount
{
private:
  int count; // Reference count

public:
  void AddRef()
  {
    // Increment the reference count
    count++;
  }

  int Release()
  {
    // Decrement the reference count and
    // return the reference count.
    return --count;
  }
};
```

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

```
template < typename T > class SmartPointer
{
private:
  T*       pData;       // pointer
  RefCount* reference; // Reference count
```

```
class RefCount
{
private:
  int count; // Reference count

public:
  void AddRef()
  {
    // Increment the reference count
    count++;
  }

  int Release()
  {
    // Decrement the reference count and
    // return the reference count.
    return --count;
  }
};
```

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer() : pData(0), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(T* pValue) : pData(pValue), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(const SmartPointer<T>& sp) : pData(sp.pData), reference(sp.reference)
 {
   // Copy constructor
   // Copy the data and reference pointer
   // and increment the reference count
   reference->AddRef();
 }
```

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer() : pData(0), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(T* pValue) : pData(pValue), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(const SmartPointer<T>& sp) : pData(sp.pData), reference(sp.reference)
 {
   // Copy constructor
   // Copy the data and reference pointer
   // and increment the reference count
   reference->AddRef();
 }
```

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer() : pData(0), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(T* pValue) : pData(pValue), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(const SmartPointer<T>& sp) : pData(sp.pData), reference(sp.reference)
 {
   // Copy constructor
   // Copy the data and reference pointer
   // and increment the reference count
   reference->AddRef();
 }
```

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer() : pData(0), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(T* pValue) : pData(pValue), reference(0)
 {
   // Create a new reference
   reference = new RefCount();
   // Increment the reference count
   reference->AddRef();
 }

 SmartPointer(const SmartPointer<T>& sp) : pData(sp.pData), reference(sp.reference)
 {
   // Copy constructor
   // Copy the data and reference pointer
   // and increment the reference count
   reference->AddRef();
 }
```

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer<T>& operator = (const SmartPointer<T>& sp)
 {
   // Assignment operator
   if (this != &sp) // Avoid self assignment
     {
       // Decrement the old reference count
       // if reference become zero delete the old data
       if(reference->Release() == 0)
         {
           delete pData;
           delete reference;
         }

       // Copy the data and reference pointer
       // and increment the reference count
       pData = sp.pData;
       reference = sp.reference;
       reference->AddRef();
     }
   return *this;
 }
```

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

```
SmartPointer<T>& operator = (const SmartPointer<T>& sp)
 {
   // Assignment operator
   if (this != &sp) // Avoid self assignment
     {
       // Decrement the old reference count
       // if reference become zero delete the old data
       if(reference->Release() == 0)
         {
           delete pData;
           delete reference;
         }

       // Copy the data and reference pointer
       // and increment the reference count
       pData = sp.pData;
       reference = sp.reference;
       reference->AddRef();
     }
   return *this;
 }
```

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

A SMART POINTER

IT TAKES IN A TEMPLATE PARAMETER, AND THEN OVERLOADS THE POINTER OPERATORS FOR THAT TEMPLATE PARAMETER TYPE.

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

```
~SmartPointer()
{
  int refCount = reference->Release();
  cout << "Destructor called..current reference count = " << refCount << endl;
  // Destructor
  // Decrement the reference count
  // if reference become zero delete the data
  if(refCount == 0)
    {
      delete pData;
      delete reference;
      cout << "Okey-dokey! Ref count is now 0, delete the underlying data" << endl;
    }
}
```

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

```
~SmartPointer()
{
  int refCount = reference->Release();
  cout << "Destructor called..current reference count = " << refCount << endl;
  // Destructor
  // Decrement the reference count
  // if reference become zero delete the data
  if(refCount == 0)
    {
      delete pData;
      delete reference;
      cout << "Okey-dokey! Ref count is now 0, delete the underlying data" << endl;
    }
}
```

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

```
~SmartPointer()
{
  int refCount = reference->Release();
  cout << "Destructor called..current reference count = " << refCount << endl;
  // Destructor
  // Decrement the reference count
  // if reference become zero delete the data
  if(refCount == 0)
    {
      delete pData;
      delete reference;
      cout << "Okey-dokey! Ref count is now 0, delete the underlying data" << endl;
    }
}
```

WHEN THE REFERENCE COUNT BECOMES ZERO, THE SMART POINTER DELETES ITSELF

A SMART POINTER

A SMART POINTER KNOWS WHEN IT HAS TO DELETE ITSELF.

HOW? BY KEEPING REFERENCE COUNT

THE SMART POINTER ADDS 1 EACH TIME SOME VARIABLE REFERENCES IT (COPIES IT, OR ASSIGNS ITS VALUE)

THE SMART POINTER SUBTRACTS 1 EACH TIME SOME VARIABLE DEREFERENCES IT (GOES OUT OF SCOPE, OR DEASSIGNS IT)

WHEN THE REFERENCE COUNT BECOMES ZERO, THE SMART POINTER DELETES ITSELF

A SMART POINTER

IS REALLY SIMPLE TO USE, THAT'S THE WHOLE IDEA :-)

```
SmartPointer<string> p(new string("Vitthal Srinivasan"));

 {
   SmartPointer<string> q = p;
   cout << *q << endl;

   SmartPointer<string> r;
   r = p;
   cout << *r << endl;
   // Destructor of r and p will be called here
   // because the variable goes out of scope..
 }
 cout << *p << endl;
 // Destructor of p will be called here
 // and smart pointer will be deleted
```

A SMART POINTER

IS REALLY SIMPLE TO USE, THAT'S THE WHOLE IDEA :-)

```
SmartPointer<string> p(new string("Vitthal Srinivasan"));

{
    SmartPointer<string> q = p;
    cout << *q << endl;

    SmartPointer<string> r;
    r = p;
    cout << *r << endl;
    // Destructor of r and p will be called here
    // because the variable goes out of scope..
}
  cout << *p << endl;
  // Destructor of p will be called here
  // and smart pointer will be deleted
```

THE SCOPE ENDS AT THE CLOSING BRACE

A SMART POINTER

IS REALLY SIMPLE TO USE, THAT'S THE WHOLE IDEA :-)

```
SmartPointer<string> p(new string("Vitthal Srinivasan"));

{
 Vitthal Srinivasan
 Vitthal Srinivasan
 Destructor called..current reference count = 2
 Destructor called..current reference count = 1
 Vitthal Srinivasan
 Destructor called..current reference count = 0
 Okey-dokey! Ref count is now 0, delete the_underlying data
  // because the variable goes out of scope..
}
cout << *p << endl;
// Destructor of p will be called here
// and smart pointer will be deleted
```

EXAMPLE 66: UNDERSTAND TEMPLATE SPECIALISATION

EXAMPLE 66: UNDERSTAND TEMPLATE SPECIALISATION

SAY WE HAVE A TEMPLATED CLASS WITH 2 TEMPLATE PARAMETERS

```
template<class KeyType, class ValueType>
class KeyValuePair
{
private:
  KeyType key;
  ValueType value;
public:
  KeyValuePair(KeyType k,ValueType v) : key(k), value(v)
  {
    cout << "Inside the class template (not a specialisation)" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  ValueType getValue() const
  {
    return value;
  }

};
```

SAY WE HAVE A TEMPLATED CLASS WITH 2 TEMPLATE PARAMETERS

```
template<class KeyType, class ValueType>
class KeyValuePair
{
private:
  KeyType key;
  ValueType value;
public:
  KeyValuePair(KeyType k,ValueType v) : key(k), value(v)
  {
  }
  KeyType getKey() const
  {
    return key;
  }
  ValueType getValue() const
  {
    return value;
  }

};
```

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

(VERY MUCH LIKE WHEN WE CUSTOMISED OUR SORT FUNCTIONS FOR STRINGS)

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k,string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

THE SYNTAX IS A BIT DIFFERENT THAN FOR FUNCTIONS, SO LET'S PARSE IT

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k,string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

THIS TELLS THE C++ COMPILER THAT WE ARE "CUSTOMIZING THE TEMPLATE"

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k, string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
};
```

THIS TELLS THE C++ COMPILER THAT WE ARE CREATING A TOTAL TEMPLATE SPECIALISATION

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k,string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

TOTAL TEMPLATE SPECIALISATION

BECAUSE ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k,string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

TOTAL TEMPLATE SPECIALISATION

BECAUSE ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<>
class KeyValuePair<string,string>
{
private:
  string key;
  string value;
public:
  KeyValuePair(string k,string v) : key(k), value(v)
  {
    cout << "Inside the complete template specialisation" << endl;
  }
  string getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

TOTAL TEMPLATE SPECIALISATION

BECAUSE ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

TOTAL TEMPLATE SPECIALISATION

ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

TOTAL TEMPLATE SPECIALISATION

ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<class KeyType>
class KeyValuePair<KeyType,int>
{
private:
  KeyType key;
  int value;
public:
  KeyValuePair(KeyType k,int v) : key(k), value(v)
  {
    cout << "Inside the partial template specialisation!" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<class KeyType>
class KeyValuePair<KeyType,int>
{
private:
  KeyType key;
  int value;
public:
  KeyValuePair(KeyType k,int v) : key(k), value(v)
  {
    cout << "Inside the partial template specialisation" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<class KeyType>
class KeyValuePair<KeyType,int>
{
private:
  KeyType key;
  int value;
public:
  KeyValuePair(KeyType k,int v) : key(k), value(v)
  {
    cout << "Inside the partial template specialisation" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<class KeyType>
class KeyValuePair<KeyType,int>
{
private:
  KeyType key;
  int value;
public:
  KeyValuePair(KeyType k,int v) : key(k), value(v)
  {
    cout << "Inside the partial template specialisation" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

HERE IS HOW WE WOULD DO IT

```
template<class KeyType>
class KeyValuePair<KeyType,int>
{
private:
  KeyType key;
  int value;
public:
  KeyValuePair(KeyType k,int v) : key(k), value(v)
  {
    cout << "Inside the partial template specialisation!" << endl;
  }
  KeyType getKey() const
  {
    return key;
  }
  string getValue() const
  {
    return value;
  }
```

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

SAY WE WOULD LIKE TO CUSTOMISE THE IMPLEMENTATION OF THIS CLASS FOR SPECIFIC TYPES

TOTAL TEMPLATE SPECIALISATION

ALL THE TEMPLATE PARAMETERS ARE SPECIFIED

PARTIAL TEMPLATE SPECIALISATION

SOME OF THE TEMPLATE PARAMETERS ARE SPECIFIED

TEMPLATE SPECIALISATION

```
string firstName("Vitthal");
string lastName("Srinivasan");

KeyValuePair<string,string> pair1(firstName,lastName);
KeyValuePair<string,int> pair2(firstName,1);
KeyValuePair<float,float> pair3(9.2,1.4);
```

```
Inside the complete template specialisation
Inside the partial template specialisation!
Inside the class template (not a specialisation)
```

TEMPLATE SPECIALISATION

```
template<>
class KeyValuePair<string,string>

KeyValuePair<string,string> pair1(firstName,lastName);
KeyValuePair<string,int> pair2(firstName,1);
KeyValuePair<float,float> pair3(9.2,1.4);
```

```
Inside the complete template specialisation
Inside the partial template specialisation!
Inside the class template (not a specialisation)
```

TEMPLATE SPECIALISATION

```
string firstName("Vitthal");
string lastName("Srinivasan");

template<class KeyType>
class KeyValuePair<KeyType,int>

KeyValuePair<string,string> pair1(firstName,lastName);
KeyValuePair<string,int> pair2(firstName,1);
KeyValuePair<float,float> pair3(9.2,1.4);
```

```
Inside the complete template specialisation
Inside the partial template specialisation!
Inside the class template (not a specialisation)
```

TEMPLATE SPECIALISATION

```
string firstName("Vitthal");
string lastName("Srinivasan");
```

```
template<class KeyType, class ValueType>
class KeyValuePair<KeyType,ValueType>
```

```
KeyValuePair<string,string> pair1(firstName,lastName);
KeyValuePair<string,int> pair2(firstName,1);
```

```
KeyValuePair<float,float> pair3(9.2,1.4);
```

```
Inside the complete template specialisation
Inside the partial template specialisation!
Inside the class template (not a specialisation)
```