Week 5

Exercise 40

../40/40.txt

```
1. Pointer variables and arrays
Pointers and arrays are similar and can sometimes be used in place of the other, but
    are not the same. An array name will decay to a constant pointer to the location
    of the first element of the array, so it can be used in some pointer-like
    notation (i.e. << *(array + 5) = 10 >>). However, an array is a block of memory
    sized to fit the amount of objects matching its dimensions. In other words,
    declaring << int array[5]; >> allocates a block in memory that can hold 5
    integers. As it is not initialised, for now that memory still contains the things
     that were there before. A pointer, on the other hand, is just that: a variable
    that stores the address of a memory location, i.e. it points towards a location.
    So although they are related concepts in C/C++, they are also very different.

2. Pointer variables and reference variables
See the drawing below (Figure 1).

3. Pointer arithmetic
It refers to the fact that as pointers store a memory location, which is just a
    numerical value, they can be used in calculations to traverse the memory. As such
    , the locations (i.e. values) in their close proximity may be relevant. For
    example, as explained, an array takes up a block of consecutive memory. So,
    incrementing a pointer will move it over the amount that is the size of its type
    (i.e. if p is a pointer to an integer, ++p will move it over the size of a single
     integer). An example of this can be found in Figure 1, part b, as well as in
    other exercises of this week. In the figure, << *(pointer[3] + 2) >> is
    equivalent to << array[3][2] >>, as well as << *(*(array + 3) + 2) >>, at least
    in their usage here. << *(array + 3) >> dereferences the location of the fourth
    row (i.e. accesses its actual value), and thus dereferencing that plus two
    accesses the third value within that row. As can be seen, these are pointers to
    the start of a block of memory, and knowing the type that is contained therein
    allows us to access specific elements in that block. In short, realising that
    pointers simply represent numerical values allows us to perform useful arithmetic
     with them.

4. Accessing an element in an array using only a pointer vs. index expression
Using a pointer will skip a step when accessing an element, which is to determine the
     size of the array element and add that to the address of the first element,
    which it also has to determine again and again. Since the index notation is
    simply a representation of operations that are actually performed, that is one
    reason to 'skip the middleman', as it were. Furthermore, adding another variable
    that has to be stored, used and incremented, namely a size_t as per the example
    in the exercise, further and unnecessarily adds to the size and weight of the
    program. Furthermore, given that an array name can decay to a pointer can add
    confusion to what kind of variable is actually being passed / used. Lastly, and
    similarly, using pointer notation makes it clear that the original data can be
    modified, rather than it being passed by value. However, as of yet, I still think
     index notation is more intuitive, but perhaps that will change over time.
```

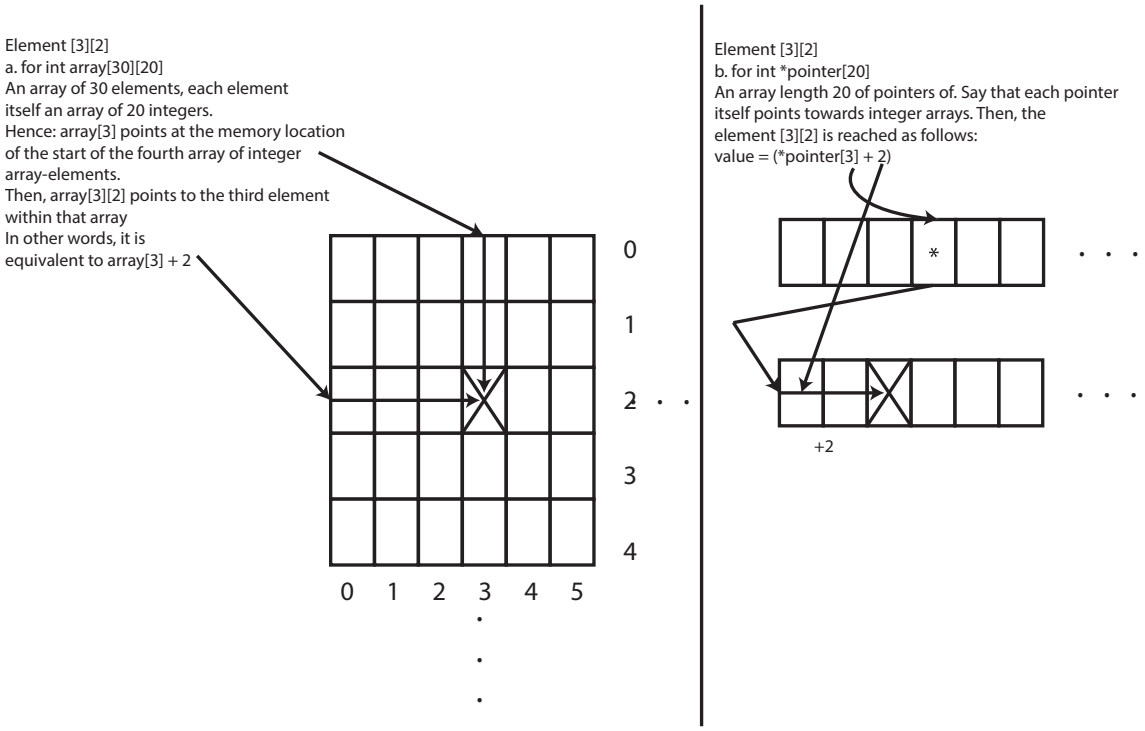

Figure 1: Illustration

Exercise 41

../41/main.cc

```
1  // Main file
2
3  #include "main.ih"
4
5  int main(int argc, char **argv)
6  {
7    char **temporary = argv;  // Create a new pointer to the memory that argv points to
8    argv = environ;           // Point argv to the memory that environ points to
9    environ = temporary;      // Point environ to where temporary points
10                             // (previously where argv was pointing)
11
12   for (size_t index = 0; argv[index] != 0; ++index) // Loop through "argv"
13     cout << argv[index] << '\n';                    // And print elements
14
15   for (size_t index = 0; environ[index] != 0; ++index)  // Loop through "environ"
16     cout << environ[index] << '\n';                     // And print elements
17 }
```

Exercise 42

../42–3/main.ih

```
#include <iostream>

#include "charcount/charcount.h"

using namespace std;

void showChar(CharCount input);
```

../42–3/main.cc

```
#include "main.ih"

int main(int argc, char **argv)
{

  CharCount charObject(cin); //constructing the object

  showChar(charObject);  //printing the values of the object
}
```

../42–3/42.ih

```
#include <iostream>

#include "charcount/charcount.h"

using namespace std;
```

../42–3/showChar.cc

```
// Programming in C/C++
// Week 4: Assignment 42
// Tjalling Otter & Emiel Krol

#include "42.ih"


void showChar(CharCount input)
{
  for (size_t index = 0; index < input.nChar(); ++index)
  {
    char character = input.getChar(index);
      switch (character)
      {
        case '\t':
        {
          cout << "\'\\t\'" << ':' << '\t';
          break;
        }
        case '\n':
        {
          cout << "\'\\n\'" << ':' << '\t';
          break;
        }
      }

      if  (isprint(character))  //printing printable characters
        cout << "\'" << character << "\'" << ':' << '\t';
      else if (character != '\n' && character != '\t')
        cout << static_cast<int>(character) << ':' << '\t';
```

```
31          //printing the other characters, checking for \n  and \t
32          //so they arent printed twice, as a value between 0 and 255
33        cout << input.Count(index) << '\n';
34        //printing the number of occurence of each char
35    }
36  }
37  //printing output
```

../42–3/charcount/charcount.h

```
1   #ifndef INCLUDED_CHARCOUNT_
2   #define INCLUDED_CHARCOUNT_
3
4   #include <iosfwd>
5
6
7   class CharCount
8   {
9     public:
10      struct Char
11      {
12        char ch;       //the character
13        size_t count = 0; //number of occurences
14      };
15
16      struct CharInfo
17      {
18        Char *ptr = new Char[1]; //field ptr pointing to struct char objects
19        size_t nChar = 0; //nr char objects stored
20      };
21
22    private:
23      CharInfo d_charObject;
24
25    public:
26      CharCount(std::istream &stream);
27
28      Char *enlarge(Char *old, size_t oldsize, size_t newsize);
29      //increasing the size of the array of Chars
30      char const getChar(size_t index) const; //getChar since just Char as a
31      //function name confused the compiler
32      //returns the character stored in Char[index]
33      size_t const nChar() const;
34      //returns the total number of different characters stored
35      size_t const Count(size_t index) const;
36      //returns the number of times the character stored in Char[index] occured
37      CharInfo const *info() const;
38      //returns a reference to the charInfo object
39      void add(char character, size_t index);
40      //increases the number of occurences of the character stored at Char[index]
41      //by 1
42      void append(char character, size_t index);
43      //Calls enlarge and then adds the character at the highest index of
44      //Char[index]
45      void insert(char character, size_t index);
46      //Calls enlarge, moves all characters at index and higher one to the right
47      //then places the new character at the current index
48      size_t locate(char character);
49      //Finds where to insert the current character
50
51  };
52
53  inline char const CharCount::getChar(size_t index) const
54  {
55    return d_charObject.ptr[index].ch;
56  }
```

```
57 //returns the character stored at Char[index]
58
59 inline size_t const CharCount::Count(size_t index) const
60 {
61   return d_charObject.ptr[index].count;
62 }
63 //returns the number of times the character stored at Char[index] occured
64
65 inline size_t const CharCount::nChar() const
66 {
67   return d_charObject.nChar;
68 }
69 //returns the total number of different characters stored
70
71 inline CharCount::CharInfo const *CharCount::info() const
72 {
73   return &d_charObject;
74 }
75 //returns a reference to the charInfo object
76 #endif
```

../42–3/charcount/charcount.ih

```
1 #include "charcount.h"
2 #include <iostream>
3
4 using namespace std;
```

../42–3/charcount/add.cc

```
1 #include "charcount.ih"
2
3 void CharCount::add(char character, size_t index)
4 {
5   //convenience references
6   Char *charObj = &d_charObject.ptr[index];
7
8     if  ((*charObj).ch == character)
9         ++(*charObj).count; //increasing count by 1
10 }
```

../42–3/charcount/append.cc

```
1 #include "charcount.ih"
2
3 void CharCount::append(char character, size_t index)
4 {
5   //convenience references:
6   size_t &Obj_nChar = d_charObject.nChar;
7   Char *&Obj_ptr = d_charObject.ptr;
8
9   if  (index == Obj_nChar)
10  {
11    if  (Obj_nChar > 0)
12      Obj_ptr = enlarge(Obj_ptr, Obj_nChar, Obj_nChar + 1);
13
14    Obj_ptr[index].ch = character;
15    Obj_ptr[index].count = 1;
16    ++Obj_nChar;
17  }
18 }
```

../42–3/charcount/charcount.h

```
1 #ifndef INCLUDED_CHARCOUNT_
```

```
2  #define INCLUDED_CHARCOUNT_
3
4  #include <iosfwd>
5
6
7  class CharCount
8  {
9    public:
10     struct Char
11     {
12       char ch;       //the character
13       size_t count = 0; //number of occurences
14     };
15
16     struct CharInfo
17     {
18       Char *ptr = new Char[1]; //field ptr pointing to struct char objects
19       size_t nChar = 0; //nr char objects stored
20     };
21
22   private:
23     CharInfo d_charObject;
24
25   public:
26     CharCount(std::istream &stream);
27
28     Char *enlarge(Char *old, size_t oldsize, size_t newsize);
29     //increasing the size of the array of Chars
30     char const getChar(size_t index) const; //getChar since just Char as a
31     //function name confused the compiler
32     //returns the character stored in Char[index]
33     size_t const nChar() const;
34     //returns the total number of different characters stored
35     size_t const Count(size_t index) const;
36     //returns the number of times the character stored in Char[index] occured
37     CharInfo const *info() const;
38     //returns a reference to the charInfo object
39     void add(char character, size_t index);
40     //increases the number of occurences of the character stored at Char[index]
41     //by 1
42     void append(char character, size_t index);
43     //Calls enlarge and then adds the character at the highest index of
44     //Char[index]
45     void insert(char character, size_t index);
46     //Calls enlarge, moves all characters at index and higher one to the right
47     //then places the new character at the current index
48     size_t locate(char character);
49     //Finds where to insert the current character
50
51 };
52
53 inline char const CharCount::getChar(size_t index) const
54 {
55   return d_charObject.ptr[index].ch;
56 }
57 //returns the character stored at Char[index]
58
59 inline size_t const CharCount::Count(size_t index) const
60 {
61   return d_charObject.ptr[index].count;
62 }
63 //returns the number of times the character stored at Char[index] occured
64
65 inline size_t const CharCount::nChar() const
66 {
67   return d_charObject.nChar;
```

```
68 }
69 //returns the total number of different characters stored
70
71 inline CharCount::CharInfo const *CharCount::info() const
72 {
73   return &d_charObject;
74 }
75 //returns a reference to the charInfo object
76 #endif
```

../42–3/charcount/constructor.cc

```
1  #include "charcount.ih"
2
3  CharCount::CharCount(std::istream &stream)
4  {
5    char character;
6
7    while  (stream.get(character))
8    {
9      size_t index = locate(character);
10
11     add(character, index);
12
13     append(character, index);
14
15     insert(character, index);
16
17   }
18
19 }
```

../42–3/charcount/enlarge.cc

```
1  #include "charcount.ih"
2
3  CharCount::Char *CharCount::enlarge(Char *old, size_t oldsize, size_t newsize)
4  {
5    Char *tmp = new Char[newsize];
6
7    for  (size_t idx = 0; idx != oldsize ; ++idx)
8      *(tmp + idx) = *(old + idx);
9
10   delete[] old;
11
12   return tmp;
13 }
```

../42–3/charcount/insert.cc

```
1  #include "charcount.ih"
2
3  void CharCount::insert(char character, size_t index)
4  {
5    //convenience characters
6    size_t &Obj_nChar = d_charObject.nChar;
7    Char *&Obj_ptr = d_charObject.ptr;
8
9    if  (index < Obj_nChar && Obj_ptr[index].ch != character)
10   { //makign sure we are not at the end of the string and the character
11     //does not equal the character at the current index
12     Obj_ptr = enlarge(Obj_ptr, Obj_nChar, Obj_nChar + 1);
13     //increasing size of object array by one
14
15     for  (size_t counter = Obj_nChar; index < counter; --counter)
```

```
16        Obj_ptr[counter] = Obj_ptr[counter - 1];
17      //moving all objects at index or past index to the right by one
18
19      Obj_ptr[index].ch = character;
20      Obj_ptr[index].count = 1;
21      ++Obj_nChar;
22      //assigning new values
23    }
24
25  }
```

../42–3/charcount/locate.cc

```
1   #include "charcount.ih"
2
3   size_t CharCount::locate(char character)
4   {
5     if  (d_charObject.ptr[0].ch == 0)
6       return 0;
7     //for initial assignment
8
9     size_t index = 0;
10    while  (d_charObject.ptr[index].ch < character && index <= d_charObject.nChar)
11      ++index;
12    return index;
13    //finding where to insert the current character
14  }
```

Exercise 43

../43/43.txt

```
-------------------------------------------------------------------------
  definition:          rewrite:
-------------------------------------------------------------------------
  int x[8];            x[3] = x[2];

pointer notation:     *(x + 3) = *(x + 2)
       semantics:     x + 3 points to the location of the 3th int beyond x.
                      Which is set to be equal to x + 2 which is the location
                      of the 2nd int beyond x.
-------------------------------------------------------------------------
  char *argv[8];       cout << argv[2];

pointer notation:      cout << *(argv + 2);
       semantics:      argv + 2 points to the location of the 2nd argument
                       beyond the first argument which is the programs name.
                       Which is then passed to cout.
-------------------------------------------------------------------------
  int x[8];            &x[10] - &x[3];

pointer notation:      &*(x+10)-&*(x+3) = 7
       semantics:      *(x+10) points to the the 10th int beyond x. Then
                       the reference & makes it instead return its location.
                       The same happens for &*(x+3). Since one points to the
                       3rd int beyond x and the other points to the 10th int
                       beyond x the result is the difference 10 - 3 = 7.
-------------------------------------------------------------------------
  char *argv[8];       argv[0]++;

pointer notation:      (*argv)++;
       semantics:      (*argv)++ points to the start of the programs name. Which
                       is a null terminated byte string. Which is dereferenced to
                       char values. Then by adding 1 to it, it still points to
                       the programs name. But now it points to 1 byte beyond the
                       start of the programs name. Such that the programs name
                       would be /45 instead of the initial ./45 if it were passed
                       to cout.
-------------------------------------------------------------------------
  char *argv[8];       argv++[0];

pointer notation:      *(argv++)
       semantics:      *(argv++) points to the first argument beyond the
                       programs name. Which is a null terminated byte string that
                       is dereferenced to a char.
-------------------------------------------------------------------------
  char *argv[8];       ++argv[0];

pointer notation:      ++(*argv)
       semantics:      ++(*argv) points to the start of the programs name. Which
                       is a null terminated byte string. Which is dereferenced to
                       char values. Then by adding 1 to it, it still points to
                       the programs name. But now it points to 1 byte beyond the
                       start of the programs name. Such that the programs name
                       would be /45 instead of the initial ./45 if it were passed
                       to cout.
-------------------------------------------------------------------------
  char **argv;         ++argv[0][2];

pointer notation:      ++*(*argv + 2)
       semantics:      First the outer pointer (the 2) points to the 2nd
                       column. Then the inner pointer
                       points the the 0th element in the 2nd row. Then 1 is
```

```
                        added to its contents. Which is of type char.
-------------------------------------------------------------------------------
```

Exercise 44

../44/main.ih

```
1  // Main: internal header file
2
3  using namespace std;
4
5  #include <cstddef>
6
7  enum FIXEDVARS
8  {
9    DIM = 10
10 };
11
12 void inv_identity(int (*entryRow)[DIM]);
13 void allOnes(int (*entryRow)[DIM]);
14 void diagZeroes(int (*entryRow)[DIM]);
15
16 void printArray(int const (*square)[DIM]);  // Testing purposes
17
18 // It does seem nicer to write a separate header file for these functions.
19 // Would it suffice to move them to a subdirectory and include a /matrixF/matrixF.h
20 // or /matrixF/matrixF.ih in main.ih?
```

../44/main.cc

```
1  // Main file
2
3  #include "main.ih"
4
5  int main()
6  {
7    int square[DIM][DIM]; // Declare square 2D array
8
9    int (*row)[DIM] = square;  // Define row as pointing to rows of 2D array
10
11   inv_identity(row);  // Pass row to function
12   // printArray(square); // Only for testing purposes
13 }
```

../44/inv_identity.cc

```
1  // Matrix function: make array into inverted identity matrix
2
3  #include "main.ih"
4
5  void inv_identity(int (*entryRow)[DIM])
6  {
7    allOnes(entryRow);    // Make all entries ones
8    diagZeroes(entryRow); // Make diagonal zeroes
9  };
```

../44/allOnes.cc

```
1  // Matrix function: make 2D array into matrix of ones
2
3  #include "main.ih"
4
5  void allOnes(int (*entryRow)[DIM])
6  {
7    for (int (*row)[DIM] = entryRow; row != entryRow + DIM; ++row)
8    {
9      for (int *column = *row, *end = column + DIM; column != end; ++column)
```

```
10        (*column) = 1;
11    }
12  };
13  // Loop through all the rows, then within those rows the columns (now individual
14  // elements) and set them all to one.
```

../44/diagZeroes.cc

```
1   // Matrix function: make diagnonal zeroes
2
3   #include "main.ih"
4
5   void diagZeroes(int (*entryRow)[DIM])
6   {
7     for (int *entry = *entryRow, index = 0; index != DIM; ++index, entry += DIM + 1)
8       (*entry) = 0;
9   };
10  // Sets every eleventh (dimension + 1) element to zero
```

Week 5

Exercises 45 & 47

../45–47/main.ih

```
// Main file: internal header

#include "strings/strings.h"

#include <string>
#include <iostream>

extern const char **environ;

using namespace std;
```

../45–47/main.cc

```
// Main file
// This is just an example main file to demonstrate the workings of the Strings class
// The constructors will also work for other NTBSs, but environ and argc/argv
// are convenient examples to use.

#include "main.ih"

int main(int argc, char const **argv)
{
  Strings objectA = Strings(cin);     // Create Strings using cin
  Strings objectB = Strings(environ); // Create Strings using environ
  Strings objectC = Strings(argc, argv);  // Create Strings based on argc, argv

  Strings::stringsSwap(objectA, objectB); // Swap environ and istream Strings

  // objectA.printStrings(); // Print what is now environ Strings
  // objectB.printStrings(); // Print what is now istream Strings
  // objectC.printStrings(); // Print the unchanged objectC
  // These are for testing purposes
}
```

../45–47/strings/strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <cstddef>
#include <string>
// #include <ioforward>

class Strings
{
  size_t d_size = 0;      // Number of elements in d_str
  std::string *d_str = 0; // Stored strings

  public:
    Strings(size_t numStrings, char const **strings); // argc, argv constructor
    Strings(char const **strings);                    // environ constructor
    Strings(std::istream &input);                     // istream constructor
    Strings();                                        // default constructor

    void printStrings() const;                        // Just for testing

    // 46
    size_t size() const;
    // std::string* data(); // Not implemented
```

```
24      // std::string* at(size_t index, bool) const; // Not implemented
25      // std::string* at(size_t index); // Not implemented
26
27      // 47
28      static void stringsSwap(Strings &objectA, Strings &objectB);
29
30    private:
31      void add(char const *novelString);          // Add char array to d_str
32  };
33
34  #endif
```

../45–47/strings/strings.ih

```
1   #include "strings.h"
2   #include <iostream>
3   //#define CERR std::cerr << __FILE__": "
4
5   using namespace std;
```

../45–47/strings/addChar.cc

```
1   #include "strings.ih"
2
3   void Strings::add(char const *novelString)
4   {
5     std::string *temporary = new string[d_size + 1];
6     // Create a pointer temporary that points towards a newly allocated
7     // piece of memory in which an array of
8     // d_size + 1 initialised strings are held
9
10    for (size_t index = 0; index != d_size; ++index)
11      temporary[index] = d_str[index];
12    // Transfer over the current array of strings to temporary
13
14    temporary[d_size] = novelString;
15    // Add the new element to the end of temporary
16
17    delete[] d_str;
18    // Delete/deallocate the memory currently pointed at by d_str
19
20    d_str = temporary;
21    // Point d_str to the memory pointed at by temporary
22
23    ++d_size;
24    // Increment d_size
25  }
```

../45–47/strings/c_argcargv.cc

```
1   #include "strings.ih"
2
3   Strings::Strings(size_t numStrings, char const **strings)
4   {
5     std::cout << "Argc / argv constructor called. \n";
6
7     for (size_t index = 0; index != numStrings; ++index)
8       add(strings[index]);
9     // For NTBSs 0 to numStrings within strings, pass them to the add function
10  }
```

../45–47/strings/c_default.cc

```
1   #include "strings.ih"
```

```
2
3  Strings::Strings()
4  {
5    std::cout << "Default constructor called. \n";
6  };
```

../45–47/strings/c_environ.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(char const **strings)
4  {
5    std::cout << "environ constructor called. \n";
6
7    for (size_t index = 0; strings[index] != 0; ++index)
8      add(strings[index]);
9    // For NTBSs 0 to when a null char is encountered, pass them to the add function
10 };
```

../45–47/strings/c_istream.cc

```
1  #include "strings.ih"
2
3  Strings::Strings(std::istream &input)
4  {
5    std::cout << "istream constructor called. \n"
6              << "Enter an empty line (enter/return) to hault input. \n";
7
8    std::string newEntry; // Define string newEntry
9    while (getline(input, newEntry))  // Loop while getline works, setting
10   {                                 // newEntry to the new line
11     if (newEntry.empty()) // If getline creates an empty string
12       break;  // Break out of the whie loop (happens when enter/return is pressed)
13
14     add(newEntry.c_str());  // Call the add using the newly entered string.
15     // Note that the string is converted to a NTBS to work with the add
16     // function. Alternatively another add function could be written.
17   }
18 }
```

../45–47/strings/stringsSwap.cc

```
1  #include "strings.ih"
2
3  void Strings::stringsSwap(Strings &objectA, Strings &objectB)
4  {
5    Strings temporary = objectA;
6    // First, a Strings object temporary is created using an implicit (i.e. non-user
7    // defined) / trivial copy constructor. In other words, temporary is constructed
8    // based on a constant reference to objectA and temporary is now a copy of objectA
9    // (in a new location in memory). Strings temporary(objectA); would do the same.
10   objectA = objectB;
11   // This a default class assignment. Now, both objectA and objectB point to the same
12   // memory, which must be remedied.
13   objectB = temporary;
14   // This assigns objectB to the same memory as temporary.
15   // Since temporary is not destroyed, this solution works fine, but really an
16   // overloaded assignment operator and copy constructor should be written.
17 };
```