

Labor für *in silico* life Konstruktion

Übungen und Experimente

Das Labor für *in silico* life Konstruktion ist digital. Seine Infrastruktur besteht aus einer integrierten Entwicklungsumgebung (Integrated Development Environment, IDE). IDE ist für die Entwicklung von Programmen für *in silico* life Konstruktionsexperimente notwendig. Einmal entwickelt führt dann das Programm den entsprechenden Experiment jedes Mal durch, wenn es ausgeführt wird.

Hier wird C++ als Programmiersprache und bzw. Microsoft Visual C++ 2008 Express Edition als IDE verwendet. Alternativ wird auch Dev-C++ verwendet. C++ ist eine Mehrzwecksprache und unterstützt mehrere Programmierstile. Sie ist eher klein und einfach. Außerdem kommt sie mit einer umfangreichen Bibliothek mit nützlichen Komponenten, die problemlos ins Programm aufgenommen werden können, um es zu unterstützen.

In der IDE beginnt die Programmentwicklung mit dem Schreiben von Quellcode in C++ mit einem Editor. Der Quellcode wird in der Regel in einer Datei mit der Endung .cpp gespeichert. Ein weiterer Teil des IDE – ein Compiler – wird dann benutzt, um Quellcode in Maschinencode zu übersetzen und in einer Datei mit der Erweiterung .obj zu speichern. Schließlich nimmt der Linker eine oder mehrere von Compiler generierte Dateien, kombiniert sie in einer einzigen ausführbaren Maschinencode und speichert in einer Datei mit der Endung .exe. Die Datei mit dem ausführbaren Maschinencode ist das fertige Programm. Ein IDE kann auch zusätzliche Instrumente enthalten, die die Programmierproduktivität maximieren.

Sobald IDE am Computer installiert ist, ist es sinnvoll, sich mit ein Paar Übungen und Experimenten aufzuwärmen. Die Übungen führen zuerst in C++ ein. Dabei braucht man keine Vorkenntnisse in C++ zu haben. Die Experimente setzen dann die Einführung in C++ fort, zeigen aber gleichzeitig, wie C++-Programme für die einfachsten *in silico* life Konstruktionsexperimente entwickelt werden. Dabei wird man auch in die Biochemie eingeführt. Wiederum sind keine biochemischen Vorkenntnisse notwendig. Dabei wird man sehen, wie ein C++-Programm den Arbeitsspeicher des Computers in einen biochemischen Reaktor umwandelt, wie unzählige Nanosonden mit Kamera ins Innere des Reaktors geschickt werden können, wie ein Experiment im Reaktor durchzuführen ist, und wie der Experimentverlauf auf den Bildschirm zu holen ist. Das Ergebnis des letzten Experiments ist *in silico* life!

Lernziele

- Übung 1:** Lernen wie ein C++-Programm mit IDE zu entwickeln ist
- Übung 2:** Lernen wie C++-Anweisungen zu schreiben sind, die den Computer instruieren, Objekte im Speicher zu konstruieren und mit ihnen zu operieren
- Übung 3:** Lernen wie C++-Code für Kontrollstrukturen zu schreiben ist
- Übung 4:** Lernen wie C++-Code für Funktionen zu schreiben ist

- Experiment 1:** Lernen wie ein C++-Program für ein *in silico* life Konstruktionsexperiment zu schreiben ist
- Experiment 2:** Lernen wie *in silico* Genexpressionsnetzwerke (GENs) zu konstruieren sind
- Experiment 3:** Lernen wie *in silico* Genomexpressionsnetzwerke (GENome) zu konstruieren sind
- Experiment 4:** Lernen wie *in silico* Genommultiplizierungsnetzwerke zu konstruieren sind

Übung 1

Ziel: Lernen wie ein C++-Programm mit IDE zu entwickeln ist.

Starte IDE (hier, Visual C++ 2008). Eine Sammlung von IDE-Fenster wird erscheinen. Auf der linken Seite befindet sich das Fenster mit drei Registerkarten. Wähle **Projektmappen-Explorer** aus. Auf der rechten Seite befindet sich das Editor-Fenster, das **Startseite** zeigt. Unten befindet sich das Fenster mit drei Registerkarten. Wähle **Ausgabe** aus.

Gehe zum Menü **Datei** und wähle **Neu -> Projekt** aus. Das Dialogbox **Neues Projekt** wird angezeigt. Auf der linken Seite befindet sich der Bereich **Projekttypen**. Wähle **Win32** aus. Auf der rechten Seite befindet sich der Bereich **Vorlagen**. Wähle **Win32-Konsolenanwendung** aus. Unten befinden sich drei Felder: **Name**, **Speicherort** und **Projektmappenname**. Gebe einen Projektnamen (hier Exercise1-1) in das Feld **Name** ein. Dieser Name wird automatisch in das Feld **Projektmappenname** übernommen. Falls erforderlich, ändere den Pfad zum Speicherort der Projektmappe im Feld **Speicherort** oder wähle den Pfad mithilfe der Schaltfläche **Durchsuchen** aus. Klicke auf die Schaltfläche **OK**. Ein **Win32-Anwendungs-Assistent** erscheint und zeigt die Seite **Übersicht** mit aktuellen Projekteinstellungen. Klicke auf die Schaltfläche **Weiter**. Die Seite **Anwendungseinstellungen** wird angezeigt. Aktiviere das Kontrollkästchen **Leeres Projekt** und klicke auf die Schaltfläche **Fertig stellen**. Im Fenster **Projektmappen-Explorer** erscheint der Projektmappen-Ordner. Er enthält den Projekt-Ordner mit drei Unterverzeichnisse: **Headerdateien**, **Quelldateien** und **Ressourcendateien**.

Wähle das Unterverzeichnis **Quelldateien** aus und klicke die rechte Maustaste. Ein Kontextmenü erscheint. Wähle **Hinzufügen -> Neues Element** aus. Das Dialogbox **Neues Element hinzufügen** wird angezeigt. Auf der linken Seite befindet sich der Bereich **Kategorien**. Wähle **Code** aus. Auf der rechten Seite befindet sich der Bereich **Vorlagen**. Wähle **C++-Datei (.cpp)** aus. Unten befinden sich zwei Felder: **Name** und **Speicherort**. Gebe einen Dateinamen (hier Exercise1-1) in das Feld **Name** ein und klicke auf die Schaltfläche **Hinzufügen**. Im Fenster **Projektmappen-Explorer** wird eine neue Datei (hier Exercise1-1.cpp) dem Projekt-Ordner im Unterverzeichnis **Quelldateien** hinzugefügt. Im Editor-Fenster erscheint eine leere Datei mit dem gleichen Namen.

Gebe den folgenden Code in die Datei Exercise1-1.cpp ein:

```

1 //Exercise 1-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 int main()
11 {
12     string Welcome("Welcome to the laboratory for in silico life construction!");
13     cout << Welcome << endl << endl;
14     return 0;
15 }
```

Beachte: die Zahlen links von der vertikalen punktierten Linie sind die Zeilennummern und gehören nicht zum Quellcode. Der Quellcode befindet sich rechts von der punktierten Linie. Die Zeilennummern sollen automatisch bei der Codeangabe erscheinen. Falls dies nicht der Fall ist, gehe zum Menü **Extras** und wähle **Optionen** aus. Das Dialogbox **Optionen** wird angezeigt. Im Bereich wähle **Text-Editor** -> **C/C++** -> **Allgemein**. Aktiviere das Kontrollkästchen **Zeilennummern** und klicke auf die Schaltfläche **OK**.

Obwohl der Quellcode in der Datei Exercise1.cpp sehr einfach ist, enthält er bereits alle Komponenten, die ein C++-Quellcode in der Regel hat.

Zeilen von 1 bis 4 enthalten Kommentare. Kommentare haben keinen Einfluss auf das Verhalten des Programms und werden verwendet, um Erläuterungen einzufügen. Hier liefern Kommentare Informationen zum Programm. Einzeilige Kommentare beginnen mit zwei Schrägstrich-Zeichen `//`. Mehrzeilige Kommentare werden in der Regel zwischen `/*` und `*/` angegeben.

Zeilen 6 und 7 enthalten zwei Anweisungen für den Präprozessor des Compilers. Jede Anweisung beginnt mit einem Nummernzeichen `#`. Hier wird der Präprozessor angewiesen, zwei Dateien `string` und `iostream` von der C++-Standardbibliothek in den Quellcode einzubinden. Ihre Funktionalität wird zu einem späteren Zeitpunkt im Programm verwendet. Die Dateinamen werden zwischen den spitzen Klammern `<` und `>` angegeben. Zeile 8 enthält ein Ausdruck, das sehr häufig in den Quellcodes zu sehen ist, die die C++-Standardbibliothek nutzen, da all ihre Komponenten in der Regel in einem Bereich deklariert sind, der als `namespace` bezeichnet wird. Hier hat `namespace` den Namen `std`.

Zeilen von 10 bis 15 enthalten die Definition der `main`-Funktion. Die `main`-Funktion ist obligatorisch. Jedes C++-Programm muss eine `main`-Funktion haben. Auch wenn ein C++-Programm weitere Funktionen enthält, beginnt die Programmausführung immer mit der `main`-Funktion und zwar unabhängig von ihrer Position innerhalb des Quellcodes. Nach dem Namen `main` der `main`-Funktion stehen Klammern `()` gefolgt vom Rumpf der Funktion in geschweiften Klammern `{}`. Hier beginnt der Rumpf der `main`-Funktion mit der Anweisung, die den Computer instruiert, im Speicher ein Objekt vom Objekttyp `string` mit den Namen `welcome` zu konstruieren und dieses Objekt als eine Zeichenfolge `welcome to the laboratory for in silico life construction!` zusammensetzen. Die nächste Anweisung instruiert den Computer, einen Standard-Ausgabestrom `cout` zu konstruieren und ihn zum Transportieren des Bildes vom Objekt `welcome` und zwei Endzeile-Zeichen zum Konsole-Fenster auf dem Bildschirm zu verwenden. Die letzte Anweisung

```
return 0;
```

ist die übliche Instruktion, um ein C++-Programm für das Konsole-Fenster zu beenden.

Darüber hinaus enthält der Quellcode auch Leerräume - Leerzeichen, Tabulatoren, Zeilenvorschübe. Wie Kommentare haben sie keine Auswirkungen auf das Verhalten des Programms sondern dienen der besseren Lesbarkeit des Codes.

Nach der Eingabe des Quellcodes in die Datei Exercise1-1.cpp gehe zum Menü **Erstellen** und wähle **Exercise1-1 erstellen** aus. Im Fenster **Ausgabe** erscheint das Protokoll mit verschiedenen Status-Meldungen. Wenn alles gut abgelaufen ist, endet das Protokoll mit der Meldung:

```
Exercise1 - 0 Fehler, 0 Warnung(en)
===== Erstellen: 1 erfolgreich, Fehler bei 0, 0 aktuell, 0 übersprungen =====
```

Um den ausführbaren Maschinencode auszuführen, gehe zum Menü **Debuggen** und wähle **Starten ohne Debugging** aus. Das Konsole-Fenster wird erscheinen und zeigt das Bild des Objekts `Welcome`:

```
Welcome to the laboratory for in silico life construction!
Drücken Sie eine beliebige Taste . . .
```

Nach Ablauf des Programms macht das Konsole-Fenster in der Regel automatisch eine Pause, zeigt die Zeichenfolge `Drücken Sie eine beliebige Taste . . .` und wartet auf die Aktion des Benutzers. Aber in einigen IDEs (z. B. Dev-C++) schließt sich das Konsole-Fenster unmittelbar nach der Programmausführung so dass nur ein kurzes Blinken auf dem Bildschirm zu sehen ist. In diesem Fall ist es notwendig, den Quellcode mit drei zusätzlichen Anweisungen zu versehen

```
cin.ignore(255, '\n');
cout << "Drücken Sie eine beliebige Taste . . .";
cin.get();
```

und zwar direkt vor der `return`-Anweisung.

Übung 2

Ziel: Lernen wie C++-Anweisungen zu schreiben sind, die den Computer instruieren, Objekte im Speicher zu konstruieren und mit ihnen zu operieren.

Erstelle eine neue/einen neuen Projektmappe/Projekt Exercise2-1 und füge eine neue Datei Exercise2-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 2-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     char Letter1;
12     Letter1 = 'a';
13
14     char Letter2 = 'b';
15     char Letter3('c');
16
17     cout << Letter1 << "\t" << Letter2 << "\t" << Letter3 << "\n\n";
18
19     return 0;
20 }
```

Zeile 11 enthält eine Deklarationsanweisung, die den Computer instruiert, ein Objekt im Speicher zu konstruieren. Die kleinste Speichereinheit ist eine Binärstelle (binary digit, bit), die einen Wert von 0 oder 1 aufnehmen kann. Der Speicher des Computers ist in einzelne Abschnitte unterteilt, genannt Adressen. Die kleinste adressierbare Einheit des Speichers ist eine Gruppe von 8 Bits, bekannt als Byte. Ein Byte ist ausreichend, um ein relativ kleines Objekt wie ein einzelnes Zeichen oder kleine ganze Zahl (zwischen 0 und 255) zu konstruieren. Um komplexere Objekte zu konstruieren, soll der Computer mehrere Bytes irgendwie gruppieren. In C++ gibt es wenige eingebaute Objekttypen. C++ bietet den Anwender auch Instrumente, ihre eigenen Objekttypen erstellen zu können. Diese Instrumente wurden intensiv von C++-Community benutzt, um eine große Anzahl von nützlichen Objekttypen zu erfinden. Viele von ihnen sind in die C++-Standardbibliothek eingefügt. Eine Deklarationsanweisung soll also mindestens zwei Identifikatoren/Namen enthalten. Der erste Identifikator/Name ist für den Objekttyp, während der zweite Identifikator/Name ist für das Objekt selbst. Hier wird der Computer instruiert, ein Objekt vom Typ `char` mit dem Namen `Letter1` im Speicher zu konstruieren. Wenn konstruiert, kann das Objekt für den Rest seines Anwendungsbereichs im Programm verwendet werden. Dieser Bereich wird auf den Block in geschweiften Klammern `{}` begrenzt, wo das Objekt deklariert wurde (hier, im Rumpf der `main`-Funktion).

Zeile 12 enthält eine Zuweisungsanweisung, die den Computer instruiert, einen Wert dem Objekt `Letter1` zuzuweisen. Um einen Wert dem Objekt zuzuweisen, ist der Zuweisungsoperator `=` verwendet. Ein Objekt vom Typ `char` kann entweder eine kleine Zahl oder ein Zeichen aus dem ASCII-Zeichensatz aufnehmen. ASCII definiert eine Zuordnung zwischen den Tasten auf der amerikanischen Tastatur und Zahlen von 1 bis

127. Zum Beispiel, das Zeichen für den Buchstaben `a` ist der Zahl 97 zugeordnet. Hier wird der Computer instruiert, das Zeichen `a` dem Objekt `Letter1` zuzuweisen. Bei Zuweisung wird ein Zeichen immer zwischen einfachen Ausführungszeichen `' '` gesetzt.

Zeilen 14 und 15 enthalten Anweisungen, die die Deklaration mit der Zuweisung verknüpfen. In der Zeile 14 erfolgt die Zuweisung explizit durch die Verwendung des Zuweisungsoperators `=`. In Zeile 15 dagegen erfolgt die Zuweisung implizit durch die Verwendung von Konstruktor-Funktion des Objekts.

Zeile 17 enthält eine Anweisung, die den Computer instruiert, einen Standard-Ausgabestrom `cout` zu konstruieren und ihn zum Transportieren der Bilder von Objekten `Letter1`, `Letter2`, and `Letter3` zum Konsole-Fenster auf dem Bildschirm zu verwenden. Darüber hinaus werden einige Zeichen mit besonderer Bedeutung zum Konsole-Fenster transportiert. Diese Zeichen heißen Escape-Sequenzen und dienen für die Formatierung des Standard-Ausgabestroms. Hier werden der Tabulator `\t` und die Neue Zeile `\n` verwendet. Der Operator `<<` wird verwendet, um Bilder von allen Objekten zum Standard-Ausgabestrom `cout` zu senden.

Erstelle `Exercise2-1` und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird Bilder der Objekte `Letter1`, `Letter2`, und `Letter3` zeigen:

```
a      b      c
Drücken Sie eine beliebige Taste . . .
```

Erstelle eine neue/einen neuen Projektmappe/Projekt `Exercise2-2` und füge eine neue Datei `Exercise2-2.cpp` hinzu. Gebe den folgenden Code in diese Datei ein:

```
1 //Exercise 2-2
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     int Number1 = 5, Number2 = 10, Number3 = 20;
12     float Number4 = 5.2, Number5 = 10.4, Number6 = 20.6;
13
14     cout << Number1 << "\t" << Number2 << "\t" << Number3 << "\n"
15          << Number4 << "\t" << Number5 << "\t" << Number6 << "\n\n"
16          << Number1 + Number2 << "\t" << Number4 + Number5 << "\n"
17          << Number3 - Number2 << "\t" << Number6 - Number5 << "\n"
18          << Number1 * Number2 << "\t" << Number4 * Number5 << "\n"
19          << Number3 / Number2 << "\t" << Number6 / Number5 << "\n"
20          << Number3 % 7 << "\n\n";
21
22     Number1 += 3; //is equivalent to Number1 = Number1 + 3
23     Number2 -= 3; //is equivalent to Number2 = Number2 - 3
24     Number3 %= 3; //is equivalent to Number3 = Number3 % 3
25     Number4 *= 3; //is equivalent to Number4 = Number4 * 3
26     Number5 /= 3; //is equivalent to Number5 = Number5 / 3
27     cout << Number1 << "\t" << Number2 << "\t" << Number3 << "\n"
28          << Number4 << "\t" << Number5 << "\t" << Number6 << "\n\n";
29
30     int i = 1, j = 10;
```

```

31     i++;    //is equivalent to i = i + 1 and to i += 1
32     j--;    //is equivalent to j = j - 1 and to j -= 1
33     cout << i << "\t" << j << "\n\n";
34
35     return 0;
36 }

```

Zeilen 11 und 12 enthalten Anweisungen, die den Computer instruieren, im Speicher drei Objekte vom Typ `int` und drei Objekte vom Typ `float` zu konstruieren. Objekte vom Typ `int` können nur ganze Zahlen halten. Um Fließkommazahlen zu halten, müssen Objekte des Typs `float` verwendet werden.

Zeilen von 14 bis 20 enthalten eine Anweisung, die den Computer instruiert, einen Standard-Ausgabestrom `cout` zu konstruieren und ihn zum Transportieren der Bilder von Objekten `Number1`, `Number2`, `Number3`, `Number4`, `Number5`, and `Number6` zum Konsole-Fenster auf dem Bildschirm zu verwenden. Darüber hinaus wird der Computer angewiesen, den Ausgabestrom zu nutzen, um einige namenlose Objekte zum Konsole-Fenster zu transportieren. Diese namenlosen Objekte entstehen nur, um Zahlen zu halten, die sich aus arithmetischen Operationen auf Objekte mit Namen resultieren. Die Operatoren `+`, `-`, `*`, und `/` entsprechen der Addition, Subtraktion, Multiplikation und Division. Der Modulo-Operator `%` gibt den Rest nach der Teilung und ist sehr nützlich für die Prüfung, ob eine Zahl restlos teilbar ist oder nicht, wenn geteilt durch eine andere Zahl. Allerdings funktioniert das nur mit ganzen Zahlen.

Zeilen von 22 bis 26 zeigen einige Beispiele für Anweisungen mit zusammengesetzten Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=`, and `%=`. Diese Operatoren dienen, um die in den Kommentaren angezeigten Ausdrücke zu verkürzen. Zeilen 31 und 32 enthalten Anweisungen, die die Operatoren `++` und `--` für weitere Verkürzung von Ausdrücken verwenden.

Erstelle `Exercise2-2` und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird Bilder von allen Objekten zeigen, die beim Standard-Ausgabestrom transportiert wurden:

```

5         10         20
5.2      10.4      20.6

15        15.6
10        10.2
50        54.08
2         1.98077
6

8         7         2
15.6     3.46667  20.6

2         9

Drücken Sie eine beliebige Taste . . .

```

Übung 3

Ziel: Lernen wie C++-Code für Kontrollstrukturen zu schreiben ist.

Erstelle eine neue/einen neuen Projektmappe/Projekt Exercise3-1 und füge eine neue Datei Exercise3-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 3-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     char Letter1 = 'a', Letter2 = 'b', Letter3 = 'c';
12     int Number;
13
14     cout << "Enter number: ";
15     cin >> Number;
16
17     if(Number == 10)
18         cout << Letter1 << "\n\n";
19     else if(Number < 10)
20         cout << Letter2 << "\n\n";
21     else
22         cout << Letter3 << "\n\n";
23
24     return 0;
25 }
```

Zeilen 14 und 15 führen eine gewisse Interaktivität ins Programm ein. Der Standard-Ausgabestrom `cout` soll eine Meldung zum Konsole-Fenster transportieren, die den Benutzer auffordert, eine beliebige Zahl anzugeben. Zeile 15 enthält eine Anweisung, die den Computer instruiert, einen Standard-Eingabestrom `cin` zu konstruieren und ihn zum Transportieren der angegebenen Zahl von der Tastatur zum Objekt `Number` im Speicher des Computers zu verwenden. Der Operator `>>` wird verwendet, um die Zahl vom Standard-Eingabestrom `cin` zum Objekt `Number` zu senden.

Zeilen von 17 bis 22 enthalten ein Beispiel für eine bedingte Kontrollstruktur. Die folgende Tabelle fasst drei verschiedene Formen dieser Kontrollstruktur zusammen:

<pre>if(Bedingung) Anweisung</pre>	<p>Das Programm wertet <code>Bedingung</code> aus. Wenn <code>Bedingung</code> wahr ist, wird <code>Anweisung</code> ausgeführt. Wenn <code>Bedingung</code> falsch ist, wird <code>Anweisung</code> übersprungen und das Programm wird zu der Anweisung nach der Kontrollstruktur übergehen.</p>
<pre>if(Bedingung) Anweisung1 else Anweisung2</pre>	<p>Das Programm wertet <code>Bedingung</code> aus. Wenn <code>Bedingung</code> wahr ist, wird <code>Anweisung1</code> ausgeführt, <code>Anweisung2</code> wird übersprungen und das Programm wird zu der Anweisung nach der Kontrollstruktur übergehen. Wenn <code>Bedingung</code> falsch ist, wird <code>Anweisung1</code> übersprungen, <code>Anweisung2</code> wird ausgeführt und das Programm wird zu der Anweisung nach der Kontrollstruktur übergehen.</p>

<pre> if(Bedingung1) Anweisung1 else if(Bedingung2) Anweisung2 else if(Bedingung3) Anweisung3 ... else Anweisung </pre>	Das Programm wertet Bedingungen aus, um Anweisung zu wählen, die ausgeführt werden soll.
---	--

In Bedingungen werden der Gleichheits-Operator == und relationale Operatoren !=, >, <, >=, <= verwendet. Hier ist die Wahl vom Wert der Zahl abhängig gemacht, die vom Benutzer eingegeben wird.

Erstelle Exercise3-1 und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird zunächst die folgende Anforderung zeigen:

```
Enter number:
```

Abhängig vom Wert der angegebenen Zahl sind drei Varianten möglich. Zum Beispiel:

```
Enter number: 10
a
Drücken Sie eine beliebige Taste . . .
```

```
Enter number: 1
b
Drücken Sie eine beliebige Taste . . .
```

```
Enter number: 100
c
Drücken Sie eine beliebige Taste . . .
```

Erstelle eine neue/einen neuen Projektmappe/Projekt Exercise3-2 und füge eine neue Datei Exercise3-2.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 3-2
2 /*Nikita Tirjatkin
3    Laboratory for in silico life construction
4    Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     int i;
12     cout << "Enter number to start countdown: ";
13     cin >> i;
14
15     while(i > 0)

```

```

16     {
17         cout << i << "! ";
18         i--;
19     }
20     cout << "ZERO-O-O!!!\n\n";
21
22     return 0;
23 }

```

Zeilen vom 15 bis 19 enthalten ein Beispiel für eine iterative Kontrollstruktur. Die folgende Tabelle fasst zwei verschiedene Formen dieser Kontrollstruktur zusammen:

<pre>while(Bedingung) Anweisung</pre>	<p>Das Programm wertet <code>Bedingung</code> aus. Wenn <code>Bedingung</code> falsch ist, wird <code>Anweisung</code> übersprungen und das Programm wird zu der Anweisung nach der Kontrollstruktur übergehen. Wenn <code>Bedingung</code> wahr ist, wird <code>Anweisung</code> ausgeführt und das Programm wird zu <code>Bedingung</code> zurückkehren, um sie erneut auszuwerten. Dieser Zyklus wiederholt sich so lange, bis <code>Bedingung</code> als falsch ausgewertet wird.</p>
<pre>do Anweisung while(Bedingung)</pre>	<p>Das Programm führt zuerst <code>Anweisung</code> aus und dann wertet <code>Bedingung</code> aus. Wenn <code>Bedingung</code> falsch ist, wird das Programm zu der Anweisung nach der Kontrollstruktur übergehen. Wenn <code>Bedingung</code> wahr ist, wird das Programm zu <code>Bedingung</code> zurückkehren, um sie erneut auszuwerten. Dieser Zyklus wiederholt sich so lange, bis <code>Bedingung</code> als falsch ausgewertet wird.</p>

Hier ist die Wahl vom Wert der Zahl abhängig gemacht, die vom Benutzer eingegeben wird. Beachte: die Anweisung ist in Wirklichkeit ein Block von zwei Anweisungen zwischen geschweiften Klammern `{}`.

Erstelle `Exercise3-2` und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird zunächst die folgende Anforderung zeigen:

```
Enter number to start countdown:
```

Nach der Zahlangabe wird das Konsole-Fenster den kompletten Countdown zeigen:

```

Enter number to start countdown: 10
10! 9! 8! 7! 6! 5! 4! 3! 2! 1! ZERO-O-O!!!

Drücken Sie eine beliebige Taste . . .

```

Erstelle eine neue/einen neuen Projektmappe/Projekt `Exercise3-3` und füge eine neue Datei `Exercise3-3.cpp` hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 3-3
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <iostream>
7 using namespace std;

```

```

8
9 int main()
10 {
11     int i;
12     cout << "Enter number to start countdown: ";
13     cin >> i;
14
15     for(int j = i; j > 0; j--)
16         cout << j << "! ";
17     cout << "ZERO-O-O!!!\n\n";
18
19     return 0;
20 }

```

Zeilen vom 15 bis 17 enthalten ein anderes Beispiel für eine iterative Kontrollstruktur. Ihre Form ist:

```

for(Anweisung1; Bedingung; Anweisung3)
    Anweisung2

```

Das Programm führt zuerst `Anweisung1` aus. Generell ist `Anweisung1` eine Initialisierungsanweisung, die eine Zählervariable deklariert und ihr einen Ausgangswert zuweist. Dann wertet das Programm `Bedingung` aus. Wenn `Bedingung` falsch ist, wird `Anweisung2` übersprungen und das Programm wird zu der Anweisung nach der Kontrollstruktur übergehen. Wenn `Bedingung` wahr ist, wird das Programm `Anweisung2` und `Anweisung3` ausführen. Generell ist `Anweisung3` eine Inkrementierungs-/Dekrementierungsanweisung, die den Wert der Zählervariable auf irgendeine reguläre Weise ändert. Danach wird das Programm zu `Bedingung` zurückkehren, um sie erneut auszuwerten. Dieser Zyklus wiederholt sich so lange, bis `Bedingung` als falsch ausgewertet wird.

Erstelle `Exercise3-3` und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird zunächst die folgende Anforderung zeigen:

```
Enter number to start countdown:
```

Nach der Zahlangabe wird das Konsole-Fenster den kompletten Countdown zeigen:

```

Enter number to start countdown: 10
10! 9! 8! 7! 6! 5! 4! 3! 2! 1! ZERO-O-O!!!

Drücken Sie eine beliebige Taste . . .

```

Beachte: das Programm zeigt die gleiche Funktionalität wie in `Exercise3-2`.

Übung 4

Ziel: Lernen wie C++-Code für Funktionen zu schreiben ist.

Erstelle eine neue/einen neuen Projektmappe/Projekt Exercise4-1 und füge eine neue Datei Exercise4-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 4-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 float addNumber(float a, float b)
10 {
11     float c;
12     c = a + b;
13     return (c);
14 }
15
16 int main()
17 {
18     float x, y, z;
19     x = addNumber(12.34, 23.45);
20     y = addNumber(34.56, 45.67);
21     z = addNumber(x, y);
22     cout << x << "\t" << y << "\t" << z << "\n\n";
23
24     return 0;
25 }

```

Zeilen von 9 bis 14 enthalten ein Beispiel für eine Funktion. Generell ist die Funktion eine benannte Gruppe von Anweisungen. Ihre Form ist:

```
Returntype Funktionsname(Parameter1, Parameter2,..){Anweisung}
```

Sobald definiert kann sie mehrmals aufgerufen werden. Um eine Funktion aufzurufen, soll die folgende Form verwendet werden:

```
Functionsname(Argument1, Argument2,..)
```

Beachte: Parameter und Argumente müssen klar miteinander korrespondieren. Hier ist die Funktion `addNumber` für die Addition definiert. Wenn aufgerufen nimmt sie zwei Gleitkommazahlen auf und gibt ihre Summe als eine Gleitkommazahl zurück.

Zeilen 19, 20 und 21 zeigen Beispiele, wie die Funktion `addNumber` innerhalb der `main`-Funktion aufgerufen wird.

Erstelle Exercise4-1 und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird folgende Ergebnisse zeigen:

```

35.79    80.23    116.02
Drücken Sie eine beliebige Taste . . .

```

Erstelle eine neue/einen neuen Projektmappe/Projekt Exercise4-2 und füge eine neue Datei Exercise4-2.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Exercise 4-2
2 /*Nikita Tirjatkin
3    Laboratory for in silico life construction
4    Januar 2010*/
5
6 #include <iostream>
7 using namespace std;
8
9 void countDown(int a)
10 {
11     cout << a << "! ";
12     if(a > 1)
13         countDown(a - 1);
14 }
15
16 int main()
17 {
18     int i;
19     cout << "Enter number to start countdown: ";
20     cin >> i;
21
22     countDown(i);
23     cout << "ZERO-O-O!!!\n\n";
24
25     return 0;
26 }

```

Zeilen von 9 bis 14 enthalten ein Beispiel für eine rekursive Funktion. Eine rekursive Funktion ist die Funktion, die sich selbst aufruft. Hier ist die Funktion `countDown` für das Countdown definiert. Beachte: Sie gibt keinen Wert zurück. In diesem Fall wird einen speziellen Typenamen `void` verwendet.

Erstelle Exercise4-2 und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird zunächst die folgende Anforderung zeigen:

```
Enter number to start countdown:
```

Nach der Zahlangabe wird das Konsole-Fenster den kompletten Countdown zeigen:

```
Enter number to start countdown: 10
10! 9! 8! 7! 6! 5! 4! 3! 2! 1! ZERO-O-O!!!

Drücken Sie eine beliebige Taste . . .
```

Experiment 1

Ziel: Lernen wie ein C++-Programm für ein *in silico* life Konstruktionsexperiment zu schreiben ist.

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment1-1 und füge eine neue Datei Experiment1-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 1-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 #include <ctime>
9 using namespace std;
10
11 void waitSeconds(int Seconds)
12 {
13     clock_t Stop;
14     Stop = clock() + Seconds * CLOCKS_PER_SEC;
15     while(clock() < Stop){}
16 }
17
18 int main()
19 {
20     char Monomers[10] = {'A', 'B', 'C', 'B', 'C', 'A', 'C', 'B', 'A', 'C'};
21     cout << "Picture:\n\n";
22     for(int i = 0; i < 10; i++)
23         cout << Monomers[i] << " ";
24     cout << "\n\n";
25
26     string Polymer;
27     cout << "Movie:\n\n";
28     for(int i = 0; i < 10; i++)
29     {
30         waitSeconds(1);
31         Polymer.push_back(Monomers[i]);
32         cout << Polymer[i];
33     }
34     cout << "\n\n";
35
36     return 0;
37 }

```

Generell hat ein C++-Programm für ein *in silico* life Konstruktionsexperiment

1. ein *in silico* Experiment zu errichten,
2. ein *in silico* Instrumentarium zu errichten,
3. das *in silico* Experiment auszuführen,
4. Ausgaben des *in silico* Instrumentariums zu zeigen/aufzuzeichnen.

Um ein *in silico* Experiment für die *in silico* life Konstruktion zu errichtet ist es notwendig, zunächst Objekttypen für *in silico* Chemikalien auszuwählen. In der Lebenswelt können alle Chemikalien grob in zwei Kategorien eingeteilt werden: Monomere und Polymere. Die bekannteste Form eines Polymers ist eine kovalent-gebundene Kette von Monomeren. Zum Beispiel, ein DNA-Molekülstrang ist eine Kette von Deoxyribonukleotiden, ein RNA-Molekül ist eine Kette von Ribonukleotiden, ein Polypeptid ist eine Kette von Aminosäuren. Der Einfachheit wegen können Objekte

vom Typ `char` als *in silico* Monomere verwendet werden. Dementsprechend können Objekte vom Typ `string` als *in silico* Polymere verwendet werden. Zur Erinnerung: `char` ist ein eingebautes Objekttyp während `string` ist ein Objekttyp, das in der Datei `string` der C++-Standardbibliothek definiert ist. Um in der Lage zu sein, den Computer anzuweisen, Objekte vom Typ `string` zu konstruieren und mit ihnen zu arbeiten, soll ein C++-Programm eine Direktive enthalten, die in Zeile 6 gezeigt ist. Der Objekttyp `string` ist speziell für die Konstruktion von Objekte bestimmt, die eine Folge von Zeichen halten und mit ihr operieren können.

Hier hat das C++-Programm ein Experiment für die *in silico* Polymerisation zu errichten. Zunächst instruiert es den Computer, im Speicher ein Pool von 10 *in silico* Monomere zu konstruieren. Dieser Pool ist ein Array – ein benannten Speicherplatz, der für die Speicherung einer Reihe von Objekten desselben Typs konzipiert ist. Die Objekte sind dem Array mithilfe geschweiften Klammern `{ }` zugewiesen. Sie alle teilen somit den gleichen Namen und können individuell durch den Index von 0 bis $N - 1$ referenziert werden, wobei N die Anzahl der Objekte im Array ist. Beachte: Nach den Namen des Arrays (hier, `Monomers`) folgt eine Zahl in eckigen Klammern `[]`. Abhängig vom Kontext kann diese Zahl entweder die Anzahl der Objekte in dem zu konstruierten Array oder der Index des zu referenzierten Objekts bedeuten. Nächstens instruiert das Programm den Computer, im Speicher ein *in silico* Polymer zu konstruieren. Es ist zuerst ein Ort, an dem die Polymerisation beginnen wird. Anschließend wird der Computer instruiert, *in silico* Monomere zum *in silico* Polymer in einer aufeinander folgenden Reihenfolge mithilfe der Funktion `push_back` zuzufügen. Beachte wie diese Funktion aufgerufen wird. Ihr Name wird mit dem Namen des Objekts `Polymer` mithilfe eines Punktzeichens `.` verbunden.

In C++-Programm für die Konsole-Anwendung ist der Standard-Ausgabestrom `cout` das beste Gerät, das als *in silico* Instrumentarium verwendet werden kann. Es ist ein Objekt des Objekttyps `ostream`, der in der Datei `iostream` der C++-Standardbibliothek definiert ist. Zur Erinnerung: Standard-Ausgabeströme sind sehr gut geeignet, um im Speicher des Computers platziert zu werden, Objekte dort aufzunehmen und ihre Bilder zum Konsole-Fenster zu transportieren. Sie sind wirklich *in silico* Nanosonden mit Kamera. Hier werden Standard-Outputströme verwendet, um Bilder der *in silico* Polymerisation zu liefern. Die Funktion `waitSeconds`, die in Zeilen von 11 bis 15 definiert ist, dient dazu, einige Bilder in ein Film zu verbinden, um die *in silico* Polymerisation in Zeitlupe zu zeigen. Die Funktion `waitSeconds` selbst nutzt einige Objekte und Funktionen, die in der Datei `ctime` der C++-Standardbibliothek definiert sind.

Erstelle Experiment1-1 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird das Bild und den Film zeigen, die das Experiment für die *in silico* Polymerisation dokumentieren:

```
Picture:
A B C B C A C B A C
Movie:
```

ABCBCACBAC

Drücken Sie eine beliebige Taste . . .

Experiment 2

Ziel: Lernen wie *in silico* Genexpressionsnetzwerke (GENs) zu konstruieren sind.

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment2-1 und füge eine neue Datei Experiment2-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment2-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 class gena
11 {
12     public:
13     string RNA;
14     gena(string Gene)
15     {
16         for(int i = 0; i < Gene.size(); i++)
17         {
18             if(Gene[i] == 'A') RNA.push_back('U');
19             else if(Gene[i] == 'C') RNA.push_back('G');
20             else if(Gene[i] == 'G') RNA.push_back('C');
21             else if(Gene[i] == 'T') RNA.push_back('A');
22             else
23                 RNA.push_back('-');
24         }
25     };
26
27     int main()
28     {
29         string s0a, s0b;
30         s0a="CGTACGCTGCTTAAGCCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC";
31         s0b="GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG";
32         cout << "Picture 1:\n\n"
33              << s0a << endl
34              << s0b << "\n\n";
35         cout << "Picture 2:\n\n"
36              << s0a << "\n\n\n"
37              << s0b << "\n\n";
38
39         gena g1(s0a.substr(0, 9));
40         cout << "Picture 3:\n\n"
41              << s0a << endl
42              << g1.RNA << "\n\n"
43              << s0b << "\n\n";
44         cout << "Picture 4:\n\n"
45              << s0a << endl
46              << s0b << "\n\n"
47              << g1.RNA << "\n\n";
48
49         return 0;
50     }

```

Die Lebenswelt ist ein äußerst komplexes Netzwerk, das aus sehr großen Anzahl von verschiedenen chemischen Reaktionen zusammengesetzt ist, die eine äußerst komplexe Matrix aus der verwirrend großen Anzahl von unterschiedlichen an diesen Reaktionen beteiligten Chemikalien kontinuierlich erzeugen. Außerdem trägt auch eine unheimlich große Anzahl von Interaktionen zwischen nichtlebendigen und lebendigen Welten der Lebenskomplexität bei. Allerdings wird die Lebenswelt auf einmal verständlich, sobald man sich damit vertraut macht, wie solche grundlegenden chemischen Reaktionen wie

DNA-Transkription, RNA-Translation und Katalyse in eine strenge Hierarchie von Lebensmustern arrangieren, die in der Tabelle dargestellt ist:

Ebene	Lebensmuster...	... ist etwa gleich zu:
4	Genomdiversifizierungsnetzwerk	Allgemeine Zellprogression (Lebenswelt, Biosphäre)
3	Genommultiplizierungsnetzwerk	Individuelle Zellprogression
2	Genomexpressionsnetzwerk (GENome)	Zelle
1	Genexpressionsnetzwerk (GEN)	

Hier hat C++-Programm ein Experiment für die Konstruktion eines *in silico* Genexpressionsnetzwerks (kurz: GEN) zu errichten. Um in der Lage zu sein, den Computer zu instruieren, solche Objekte wie *in silico* GENs zu konstruieren, müssen einige neue Objekttypen definiert werden. Die Tabelle fasst zwei verschiedene Formen für die Definition neuer Objekttypen in C++ zusammen:

<pre>class Typename { Elementobject1; Elementobject2; ... Elementfunction1; Elementfunction2; ... };</pre>	Per Vorgabe haben Elementobjekte und Elementfunktionen private Zugriffsrechte.
<pre>struct Typename { Elementobject1; Elementobject2; ... Elementfunction1; Elementfunction2; ... };</pre>	Per Vorgabe haben Elementobjekte und Elementfunktionen öffentliche (public) Zugriffsrechte.

Zeilen von 10 bis 26 enthalten Code, der einen neuen Objekttyp `gena` für *in silico* GENs definiert, die auf die DNA-Transkription beschränkt sind.

Während der DNA-Transkription wird ein DNA-Molekülstrang von den anderen getrennt und sein Abschnitt mit einer bestimmten Sequenz von Deoxyribonukleotiden – ein Gen – wird als Vorlage für die Synthese eines RNA-Moleküls benutzt. Dies ist möglich, weil ein Ribonukleotid zu einem Deoxyribonukleotid gebunden werden darf und zwar nach den Regeln der Basenpaarbindung:

Ein Deoxyribonukleotid mit der Base...	Adenin	Cytosin	Guanin	Thymin
... bindet ein Ribonukleotid mit der Base...	Uracil	Guanin	Cytosin	Adenin

Somit bestimmt die Abfolge von Deoxyribonukleotiden im Gen die Abfolge von Ribonukleotiden im zu synthetisierten RNA-Molekül eindeutig. Dementsprechend entsteht auf der DNA-Vorlage eine komplementäre RNA-Replika, die dann vom DNA-Molekülstrang ablöst. Beachte: Die DNA besteht die Transkription, ohne verbraucht zu werden.

Hier enthält der neue Objekttyp `gena` ein Elementobjekt `RNA` des Objekttyps `string` und eine Elementfunktion `gena`. Diese Elementfunktion ist eine Konstruktor-Funktion, die automatisch aufgerufen wird, wenn ein Objekt vom Typ `gena` im Speicher des

Computers konstruiert wird. Die Konstruktor-Funktion muss den gleichen Namen wie der neue Objekttyp haben. Sie kann keinen Rückgabotyp haben. Hier hat die Konstruktor-Funktion einen Parameter `Gene` vom Typ `string`. Der Rumpf der Konstruktor-Funktion enthält Code für die Kontrollstruktur, die die RNA-Replika (hier, RNA) auf der DNA-Vorlage (hier `Gene`) synthetisiert. Zeichen für *in silico* Deoxyribonukleotide (**A**, **C**, **G**, und **T**) und Ribonucleotide (**A**, **C**, **G**, und **U**) sind entsprechend ihrer konventionellen Bezeichnungen gewählt.

Zeilen von 29 bis 37 enthalten Code, der den Computer instruiert, im Speicher ein *in silico* DNA-Molekül zu konstruieren. Die Molekülstränge müssen zunächst beieinander liegen und dann trennen sie sich. Zeile 39 enthält eine Anweisung, die den Computer instruiert, im Speicher ein *in silico* GEN `g1` vom Typ `gena` zu konstruieren, das einen Teilstring von Position 0 bis 8 auf dem ersten Strang des DNA-Moleküls als Argument für seine Konstruktor-Funktion aufnimmt.

Erstelle Experiment2-1 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genexpression (hier, DNA-Transkription) dokumentieren:

```
Picture 1:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 2:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC

GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 3:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCAUGCGAC

GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 4:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC

Drücken Sie eine beliebige Taste . . .
```

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment2-2 und füge eine neue Datei Experiment2-2.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```
1 //Experiment 2-2
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
```

```

8 using namespace std;
9
10 class gena
11 {
12 public:
13     string RNA;
14     gena(string Gene)
15     {
16         for(int i = 0; i < Gene.size(); i++)
17         {
18             if(Gene[i] == 'A') RNA.push_back('U');
19             else if(Gene[i] == 'C') RNA.push_back('G');
20             else if(Gene[i] == 'G') RNA.push_back('C');
21             else if(Gene[i] == 'T') RNA.push_back('A');
22             else RNA.push_back('-');
23         }
24     }
25 };
26
27 class genb:public gena
28 {
29 public:
30     string Polypeptide;
31     genb(string Gene):gena(Gene)
32     {
33         for(int i = 0; i < RNA.size(); i += 3)
34         {
35             if(RNA.substr(i, 3) == "GCA" ||
36                RNA.substr(i, 3) == "GCC" ||
37                RNA.substr(i, 3) == "GCG" ||
38                RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
39             else if(RNA.substr(i, 3) == "UGC" ||
40                RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
41             else if(RNA.substr(i, 3) == "GAC" ||
42                RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
43             else if(RNA.substr(i, 3) == "GAA" ||
44                RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
45             else if(RNA.substr(i, 3) == "UUC" ||
46                RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
47             else if(RNA.substr(i, 3) == "GGA" ||
48                RNA.substr(i, 3) == "GGC" ||
49                RNA.substr(i, 3) == "GGG" ||
50                RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
51             else if(RNA.substr(i, 3) == "CAC" ||
52                RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
53             else if(RNA.substr(i, 3) == "AUA" ||
54                RNA.substr(i, 3) == "AUC" ||
55                RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
56             else if(RNA.substr(i, 3) == "AAA" ||
57                RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');
58             else if(RNA.substr(i, 3) == "CUA" ||
59                RNA.substr(i, 3) == "CUC" ||
60                RNA.substr(i, 3) == "CUG" ||
61                RNA.substr(i, 3) == "CUU" ||
62                RNA.substr(i, 3) == "UUA" ||
63                RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');
64             else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
65             else if(RNA.substr(i, 3) == "AAC" ||
66                RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
67             else if(RNA.substr(i, 3) == "CCA" ||
68                RNA.substr(i, 3) == "CCC" ||
69                RNA.substr(i, 3) == "CCG" ||
70                RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
71             else if(RNA.substr(i, 3) == "CAA" ||
72                RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
73             else if(RNA.substr(i, 3) == "AGA" ||
74                RNA.substr(i, 3) == "AGG" ||
75                RNA.substr(i, 3) == "CGA" ||
76                RNA.substr(i, 3) == "CGC" ||
77                RNA.substr(i, 3) == "CGG" ||
78                RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
79             else if(RNA.substr(i, 3) == "AGC" ||
80                RNA.substr(i, 3) == "AGU" ||
81                RNA.substr(i, 3) == "UCA" ||

```

```

82         RNA.substr(i, 3) == "UCC" ||
83         RNA.substr(i, 3) == "UCG" ||
84         RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
85     else if(RNA.substr(i, 3) == "ACA" ||
86            RNA.substr(i, 3) == "ACC" ||
87            RNA.substr(i, 3) == "ACG" ||
88            RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
89     else if(RNA.substr(i, 3) == "GUA" ||
90            RNA.substr(i, 3) == "GUC" ||
91            RNA.substr(i, 3) == "GUG" ||
92            RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
93     else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
94     else if(RNA.substr(i, 3) == "UAC" ||
95            RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
96     else
97         Polypeptide.push_back('-');
98     }
99 };
100
101 int main()
102 {
103     string s0a, s0b;
104     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCCGACACTACTC";
105     s0b="GCATGCCGACGAATTCGGACACATAAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCCTGTGATGAG";
106     cout << "Picture 1:\n\n"
107          << s0a << endl
108          << s0b << "\n\n";
109     cout << "Picture 2:\n\n"
110          << s0a << "\n\n\n"
111          << s0b << "\n\n";
112
113     genb g3(s0a.substr(18, 9));
114     cout << "Picture 3:\n\n"
115          << s0a << endl
116          << " " << g3.RNA << "\n\n"
117          << s0b << "\n\n";
118     cout << "Picture 4:\n\n"
119          << s0a << endl
120          << s0b << "\n\n"
121          << " " << g3.RNA << "\n\n";
122     cout << "Picture 5:\n\n"
123          << s0a << endl
124          << s0b << "\n\n"
125          << " " << g3.RNA << endl
126          << " " << g3.Polypeptide << "\n\n";
127     cout << "Picture 6:\n\n"
128          << s0a << endl
129          << s0b << "\n\n"
130          << " " << g3.RNA << "\n\n"
131          << " " << g3.Polypeptide << "\n\n";
132
133     return 0;
134 }

```

Hier hat C++-Programm ein erweitertes Experiment für die Konstruktion eines *in silico* Genexpressionsnetzwerks zu errichten. Zusätzlich zu der DNA-Transkription soll dieser *in silico* GEN die RNA-Translation enthalten.

Während der RNA-Translation wird ein RNA-Molekül als Vorlage für die Synthese eines Polypeptid-Moleküls benutzt. Dabei bestimmen jeweils drei aneinanderfolgende Ribonukleotide der RNA – die Codons – die Aminosäure, die ins Polypeptid eingebaut wird:

Codons..	... für Aminosäure
GCA, GCC, GCG, GCU	Alanin (Ala, A)
UGC, UGU	Cystein (Cys, C)

GAC, GAU	Asparaginsäure (Asp, D)
GAA, GAG	Glutaminsäure (Glu, E)
UUC, UUU	Phenylalanin (Phe, F)
GGA, GGC, GGG, GGU	Glycin (Gly, G)
CAC, CAU	Histidin (His, H)
AUA, AUC, AUU	Isoleucin (Ile, I)
AAA, AAG	Lysin (Lys, K)
CUA, CUC, CUG, CUU, UUA, UUG	Leucin (Leu, L)
AUG	Methionin (Met, M)
AAC, AAU	Asparagin (Asn, N)
CCA, CCC, CCG, CCU	Prolin (Pro, P)
CAA, CAG	Glutamin (Gln, Q)
AGA, AGG, CGA, CGC, CGG, CGU	Arginin (Arg, R)
AGC, AGU, UCA, UCC, UCG, UCU	Serin (Ser, S)
ACA, ACC, ACG, ACU	Threonin (Thr, T)
GUA, GUC, GUG, GUU	Valin (Val, V)
UGG	Tryptophan (Trp, W)
UAC, UAU	Tyrosin (Tyr, Y)

Somit bestimmt die Abfolge von Codons in der RNA die Abfolge von Aminosäuren im zu synthetisierten Polypeptid-Molekül eindeutig. Dementsprechend entsteht auf der RNA-Vorlage eine komplementäre Polypeptid-Replika. Beachte: Die RNA besteht die Translation, ohne verbraucht zu werden.

Ein neuer Objekttyp `genb` für *in silico* GENS mit DNA-Transkription und RNA-Translation kann als abgeleitet vom Typ `gena` definiert werden, wie in Zeilen von 27 bis 99 gezeigt ist. Um ein Objekttyp als abgeleitet von einem anderen zu deklarieren, muss ein Doppelpunkt `:` verwendet werden. Wird ein Objekttyp als abgeleitet von einem anderen deklariert, erhält er automatisch seine Elementobjekte und Elementfunktionen zusätzlich zu eigenen. Hier erhält der abgeleiteten Objekttyp `genb` vom Objekttyp `gena` ein Elementobjekt `RNA` zusätzlich zu eigenem Elementobjekt `Polypeptide` und die Konstruktor-Funktion `gena` zusätzlich zu eigener Konstruktor-Funktion `genb`. Wenn ein Objekt vom Typ `genb` im Speicher des Computers konstruiert wird, wird zuerst die Konstruktor-Funktion `gena` und dann die Konstruktor-Funktion `genb` aufgerufen. Der Rumpf der Konstruktor-Funktion `genb` enthält Code für die Kontrollstruktur, die die Polypeptid-Replika (hier, `Polypeptide`) auf der RNA-Vorlage (hier `RNA`) synthetisiert. Zeichen für *in silico* Ribonukleotide (A, C, G, und U) und Aminosäure (A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W und Y) sind entsprechend ihrer konventionellen Bezeichnungen gewählt. Beachte: Das Zeichen `|` wird als logischer Operator ODER verwendet.

Zeilen von 103 bis 111 enthalten Code, der den Computer instruiert, im Speicher ein *in silico* DNA-Molekül zu konstruieren. Die Molekülstränge müssen zunächst beieinander liegen und dann trennen sie sich. Zeile 113 enthält eine Anweisung, die den Computer instruiert, im Speicher ein *in silico* GEN `g3` vom Typ `genb` zu konstruieren, das einen Teilstring von Position 18 bis 26 auf dem ersten Strang des DNA-Moleküls als Argument für seine Konstruktor-Funktion aufnimmt.

Erstelle Experiment2-2 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genexpression (hier, DNA-Transkription + RNA-Translation) dokumentieren:

```

Picture 1:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 2:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC

GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 3:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
      CACAUAAAA

GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 4:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

      CACAUAAAA

Picture 5:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

      CACAUAAAA
      HIK

Picture 6:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

      CACAUAAAA

      HIK

Drücken Sie eine beliebige Taste . . .

```

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment2-3 und füge eine neue Datei Experiment2-3.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 2-3
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 class gena
11 {
12     public:
13         string RNA;
14         gena(string Gene)
15         {
16             for(int i = 0; i < Gene.size(); i++)

```

```

17     {
18         if(Gene[i] == 'A') RNA.push_back('U');
19     else if(Gene[i] == 'C') RNA.push_back('G');
20     else if(Gene[i] == 'G') RNA.push_back('C');
21     else if(Gene[i] == 'T') RNA.push_back('A');
22     else
23         RNA.push_back('-');
24     }
25 };
26
27 class genb:public gena
28 {
29     public:
30     string Polypeptide;
31     genb(string Gene):gena(Gene)
32     {
33         for(int i = 0; i < RNA.size(); i += 3)
34         {
35             if(RNA.substr(i, 3) == "GCA" ||
36                RNA.substr(i, 3) == "GCC" ||
37                RNA.substr(i, 3) == "GCG" ||
38                RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
39             else if(RNA.substr(i, 3) == "UGC" ||
40                RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
41             else if(RNA.substr(i, 3) == "GAC" ||
42                RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
43             else if(RNA.substr(i, 3) == "GAA" ||
44                RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
45             else if(RNA.substr(i, 3) == "UUC" ||
46                RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
47             else if(RNA.substr(i, 3) == "GGA" ||
48                RNA.substr(i, 3) == "GGC" ||
49                RNA.substr(i, 3) == "GGG" ||
50                RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
51             else if(RNA.substr(i, 3) == "CAC" ||
52                RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
53             else if(RNA.substr(i, 3) == "AUA" ||
54                RNA.substr(i, 3) == "AUC" ||
55                RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
56             else if(RNA.substr(i, 3) == "AAA" ||
57                RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');
58             else if(RNA.substr(i, 3) == "CUA" ||
59                RNA.substr(i, 3) == "CUC" ||
60                RNA.substr(i, 3) == "CUG" ||
61                RNA.substr(i, 3) == "CUU" ||
62                RNA.substr(i, 3) == "UUA" ||
63                RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');
64             else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
65             else if(RNA.substr(i, 3) == "AAC" ||
66                RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
67             else if(RNA.substr(i, 3) == "CCA" ||
68                RNA.substr(i, 3) == "CCC" ||
69                RNA.substr(i, 3) == "CCG" ||
70                RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
71             else if(RNA.substr(i, 3) == "CAA" ||
72                RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
73             else if(RNA.substr(i, 3) == "AGA" ||
74                RNA.substr(i, 3) == "AGG" ||
75                RNA.substr(i, 3) == "CGA" ||
76                RNA.substr(i, 3) == "CGC" ||
77                RNA.substr(i, 3) == "CGG" ||
78                RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
79             else if(RNA.substr(i, 3) == "AGC" ||
80                RNA.substr(i, 3) == "AGU" ||
81                RNA.substr(i, 3) == "UCA" ||
82                RNA.substr(i, 3) == "UCC" ||
83                RNA.substr(i, 3) == "UCG" ||
84                RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
85             else if(RNA.substr(i, 3) == "ACA" ||
86                RNA.substr(i, 3) == "ACC" ||
87                RNA.substr(i, 3) == "ACG" ||
88                RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
89             else if(RNA.substr(i, 3) == "GUA" ||
90                RNA.substr(i, 3) == "GUC" ||

```

```

91         RNA.substr(i, 3) == "GUG" ||
92         RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
93     else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
94     else if(RNA.substr(i, 3) == "UAC" ||
95         RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
96     else
97         Polypeptide.push_back('-');
98     }
99 };
100
101 class genca:public genb
102 {
103     public:
104     char Monomer;
105     genca(string Gene, char a, char b, char c):genb(Gene)
106     {
107         Monomer = c;
108     }
109 };
110
111 int main()
112 {
113     string s0a, s0b;
114     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGCATACCATGCGGACACTACTC";
115     s0b="GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG";
116     cout << "Picture 1:\n\n"
117         << s0a << endl
118         << s0b << "\n\n";
119     cout << "Picture 2:" << "\n\n"
120         << s0a << "\n\n\n"
121         << s0b << "\n\n";
122
123     genca g6(s0a.substr(45, 9), 'U', 'V', 'X');
124     cout << "Picture 3:\n\n"
125         << s0a << endl
126         << "                " << g6.RNA << "\n\n"
127         << s0b << "\n\n";
128     cout << "Picture 4:\n\n"
129         << s0a << endl
130         << s0b << "\n\n"
131         << "                " << g6.RNA << "\n\n";
132     cout << "Picture 5:\n\n"
133         << s0a << endl
134         << s0b << "\n\n"
135         << "                " << g6.RNA << endl
136         << "                " << g6.Polypeptide
137         << "\n\n";
138     cout << "Picture 6:\n\n"
139         << s0a << endl
140         << s0b << "\n\n"
141         << "                " << g6.RNA << "\n\n"
142         << "                " << g6.Polypeptide
143         << "\n\n";
144     cout << "Picture 7:\n\n"
145         << s0a << endl
146         << s0b << "\n\n"
147         << "                " << g6.RNA << "\n\n"
148         << "                " << g6.Polypeptide
149         << endl
150         << "                " << g6.Monomer
151         << "\n\n";
152     cout << "Picture 8:\n\n"
153         << s0a << endl
154         << s0b << "\n\n"
155         << "                " << g6.RNA << "\n\n"
156         << "                " << g6.Polypeptide
157         << endl
158         << "                " << g6.Monomer
159         << "\n\n";
160     cout << "Picture 9:\n\n"
161         << s0a << endl
162         << s0b << "\n\n"
163         << "                " << g6.RNA << "\n\n"
164         << "                " << g6.Polypeptide

```

```

165         << "                                     " << g6.Monomer
166         << "\n\n";
167
168     return 0;
169 }

```

Hier hat C++-Programm ein erweitertes Experiment für die Konstruktion eines *in silico* Genexpressionsnetzwerks zu errichten. Zusätzlich zu der DNA-Transkription und RNA-Translation soll dieser *in silico* GEN die Katalyse enthalten.

Während der Katalyse dient der Katalysator als Vorlage für eine Reaktion, die sonst zu langsam fürs Leben ablaufen würde. Der Katalysator macht seinen Job der Katalyse durch den engsten Kontakt mit einem oder mehreren Substratmoleküle und durch die Interaktion mit ihnen, um einige chemische Bindungen zu knüpfen oder zu brechen. Der Katalysator ist sehr spezifisch für die chemische Reaktion, die er katalysiert, und diese Spezifität liegt in einer ausgeklügelten Anordnung der Atome in einem oder mehreren aktiven Zentren des Katalysators. Nur bestimmte Substratmoleküle können diese Konfiguration erkennen und an sie binden. Im Katalysator führt diese Bindung zu einer Konformationsänderung, was die katalysierte Reaktion auf irgendwelche Weise fördert. Danach gibt der Katalysator die Reaktionsprodukte frei, erwirbt seine ursprüngliche Gestalt und ist für die Katalyse erneut verfügbar. Beachte: Der Katalysator besteht die Katalyse, ohne verbraucht zu werden.

Ein neuer Objekttyp für *in silico* GENs mit DNA-Transkription, RNA-Translation und Katalyse kann als abgeleitet vom Typ `genb` definiert werden. Zeilen von 101 bis 109 enthalten Definition des Objekttyps `genca` für *in silico* GENs mit der Katalyse der Reaktion $A + B = C$, wobei A, B und C Monomere sind. Weil der Objekttyp `genb` selbst vom Objekttyp `gena` abgeleitet ist, erhält der Objekttyp `genca` vom Objekttyp `genb` Elementobjekte `RNA` und `Polypeptide` zusätzlich zu eigenem Elementobjekt `Monomer` und Konstruktor-Funktionen `gena` und `genb` zusätzlich zu eigener Konstruktor-Funktion `genca`. Wenn ein Objekt vom Typ `genca` im Speicher des Computers konstruiert wird, werden Konstruktor-Funktionen in Reihenfolge `gena`, `genb`, `genca` aufgerufen. Zusätzlich zum Parameter `Gene` vom Typ `string` hat die Konstruktor-Funktion `genca` drei Parameter `a`, `b`, und `c` vom Typ `char`. Ihr Rumpf enthält Code für *in silico* Katalyse der Reaktion $A + B = C$ auf der Polypeptid-Replika (hier, `Polypeptide`). Zeichen für *in silico* Monomere (`u`, `v` und `x`) sind willkürlich gewählt.

Zeilen von 113 bis 121 enthalten Code, der den Computer instruiert, im Speicher ein *in silico* DNA-Molekül zu konstruieren. Die Molekülstränge müssen zunächst beieinander liegen und dann trennen sie sich. Zeile 123 enthält eine Anweisung, die den Computer instruiert, im Speicher ein *in silico* GEN `g6` vom Typ `genca` zu konstruieren, das einen Teilstring von Position 45 bis 53 auf dem ersten Strang des DNA-Moleküls als Argument für seine Konstruktor-Funktion aufnimmt. Zusätzlich werden Zeichen `u`, `v`, und `x` als andere drei Argumente verwendet.

Erstelle Experiment2-3 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genexpression (hier, DNA-Transkription + RNA-Translation + Katalyse, wobei die Reaktion $A + B = C$ katalysiert wird) dokumentieren:

Picture 1:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

Picture 2:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

Picture 3:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
AGCACAGUA
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

Picture 4:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
AGCACAGUA
```

Picture 5:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
AGCACAGUA
STV
```

Picture 6:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
AGCACAGUA
STV
```

Picture 7:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
AGCACAGUA
STV
U V
```

Picture 8:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
AGCACAGUA
STV
X
```

Picture 9:

```
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

```

AGCACAGUA
STV
X
Drücken Sie eine beliebige Taste . . .

```

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment2-4 und füge eine neue Datei Experiment2-4.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 2-4
2 /*Nikita Tirjatkin
3 Laboratory for in silico life construction
4 Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 class gena
11 {
12 public:
13     string RNA;
14     gena(string Gene)
15     {
16         for(int i = 0; i < Gene.size(); i++)
17         {
18             if(Gene[i] == 'A') RNA.push_back('U');
19             else if(Gene[i] == 'C') RNA.push_back('G');
20             else if(Gene[i] == 'G') RNA.push_back('C');
21             else if(Gene[i] == 'T') RNA.push_back('A');
22             else RNA.push_back('-');
23         }
24     }
25 };
26
27 class genb:public gena
28 {
29 public:
30     string Polypeptide;
31     genb(string Gene):gena(Gene)
32     {
33         for(int i = 0; i < RNA.size(); i += 3)
34         {
35             if(RNA.substr(i, 3) == "GCA" ||
36                RNA.substr(i, 3) == "GCC" ||
37                RNA.substr(i, 3) == "GCG" ||
38                RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
39             else if(RNA.substr(i, 3) == "UGC" ||
40                    RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
41             else if(RNA.substr(i, 3) == "GAC" ||
42                    RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
43             else if(RNA.substr(i, 3) == "GAA" ||
44                    RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
45             else if(RNA.substr(i, 3) == "UUC" ||
46                    RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
47             else if(RNA.substr(i, 3) == "GGA" ||
48                    RNA.substr(i, 3) == "GGC" ||
49                    RNA.substr(i, 3) == "GGG" ||
50                    RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
51             else if(RNA.substr(i, 3) == "CAC" ||
52                    RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
53             else if(RNA.substr(i, 3) == "AUA" ||
54                    RNA.substr(i, 3) == "AUC" ||
55                    RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
56             else if(RNA.substr(i, 3) == "AAA" ||
57                    RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');

```

```

58         else if(RNA.substr(i, 3) == "CUA" ||
59                RNA.substr(i, 3) == "CUC" ||
60                RNA.substr(i, 3) == "CUG" ||
61                RNA.substr(i, 3) == "CUU" ||
62                RNA.substr(i, 3) == "UUA" ||
63                RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');
64         else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
65         else if(RNA.substr(i, 3) == "AAC" ||
66                RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
67         else if(RNA.substr(i, 3) == "CCA" ||
68                RNA.substr(i, 3) == "CCC" ||
69                RNA.substr(i, 3) == "CCG" ||
70                RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
71         else if(RNA.substr(i, 3) == "CAA" ||
72                RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
73         else if(RNA.substr(i, 3) == "AGA" ||
74                RNA.substr(i, 3) == "AGG" ||
75                RNA.substr(i, 3) == "CGA" ||
76                RNA.substr(i, 3) == "CGC" ||
77                RNA.substr(i, 3) == "CGG" ||
78                RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
79         else if(RNA.substr(i, 3) == "AGC" ||
80                RNA.substr(i, 3) == "AGU" ||
81                RNA.substr(i, 3) == "UCA" ||
82                RNA.substr(i, 3) == "UCC" ||
83                RNA.substr(i, 3) == "UCG" ||
84                RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
85         else if(RNA.substr(i, 3) == "ACA" ||
86                RNA.substr(i, 3) == "ACC" ||
87                RNA.substr(i, 3) == "ACG" ||
88                RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
89         else if(RNA.substr(i, 3) == "GUA" ||
90                RNA.substr(i, 3) == "GUC" ||
91                RNA.substr(i, 3) == "GUG" ||
92                RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
93         else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
94         else if(RNA.substr(i, 3) == "UAC" ||
95                RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
96         else
97             Polypeptide.push_back('-');
98     }
99 };
100
101 class genCb:public genB
102 {
103     public:
104         char Monomer1;
105         char Monomer2;
106         genCb(string Gene, char a, char b, char c):genB(Gene)
107         {
108             Monomer1 = b;
109             Monomer2 = c;
110         }
111 };
112
113 int main()
114 {
115     string s0a, s0b;
116     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC";
117     s0b="GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCCTGTGATGAG";
118     cout << "Picture 1:\n\n"
119           << s0a << endl
120           << s0b << "\n\n";
121     cout << "Picture 2:\n\n"
122           << s0a << "\n\n\n"
123           << s0b << "\n\n";
124
125     genCb g7(s0a.substr(54, 9), 'W', 'Y', 'Z');
126     cout << "Picture 3:\n\n"
127           << s0a << endl
128           << "
129           << "\n\n" << s0b << "\n\n";
130     cout << "Picture 4:\n\n"
131           << s0a << endl

```

```

132         << s0b << "\n\n"
133         << " " << g7.RNA
134         << "\n\n";
135     cout << "Picture 5:\n\n"
136         << s0a << endl
137         << s0b << "\n\n"
138         << " " << g7.RNA
139         << endl
140         << " "
141         << g7.Polypeptide << "\n\n";
142     cout << "Picture 6:\n\n"
143         << s0a << endl
144         << s0b << "\n\n"
145         << " " << g7.RNA
146         << "\n\n"
147         << " "
148         << g7.Polypeptide << "\n\n";
149     cout << "Picture 7:\n\n"
150         << s0a << endl
151         << s0b << "\n\n"
152         << " " << g7.RNA
153         << "\n\n"
154         << " "
155         << g7.Polypeptide << endl
156         << " " << W
157         << "\n\n";
158     cout << "Picture 8:\n\n"
159         << s0a << endl
160         << s0b << "\n\n"
161         << " " << g7.RNA
162         << "\n\n"
163         << " "
164         << g7.Polypeptide << endl
165         << " "
166         << g7.Monomer1 << " " << g7.Monomer2 << "\n\n";
167     cout << "Picture 9:\n\n"
168         << s0a << endl
169         << s0b << "\n\n"
170         << " " << g7.RNA
171         << "\n\n"
172         << " "
173         << g7.Polypeptide << "\n\n"
174         << " "
175         << g7.Monomer1 << " " << g7.Monomer2 << "\n\n";
176
177     return 0;
178 }

```

Hier hat C++-Programm ein Experiment für die Konstruktion eines *in silico* Genexpressionsnetzwerks zu errichten, der die DNA-Transkription, RNA-Translation und Katalyse enthält, wobei die Reaktion $A = B + C$ katalysiert werden soll. Zeilen von 101 bis 111 enthalten Definition des Objekttyps `gencb` für solche *in silico* GENS. Erwartungsgemäß sieht `gencb` wie `genca` aus. Allerdings hat er zwei Elementobjekte. Der Unterschied ist auch im Rumpf der Konstruktor-Funktion zu sehen, die die katalysierte Reaktion spezifiziert. Zeichen für *in silico* Monomere (`w`, `y` und `z`) sind willkürlich gewählt.

Erstelle Experiment2-4 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genexpression (hier, DNA-Transkription + RNA-Translation + Katalyse, wobei die Reaktion $A = B + C$ katalysiert wird) dokumentieren:

Picture 1:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 2:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC

GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 3:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
UGGUACGCC

GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 4:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC

Picture 5:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC
WYA

Picture 6:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC
WYA

Picture 7:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC
WYA
W

Picture 8:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC
WYA
Y Z

Picture 9:

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

UGGUACGCC
WYA
Y Z

Drücken Sie eine beliebige Taste . . .

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment2-5 und füge eine neue Datei Experiment2-5.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 2-5
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 class gena
11 {
12     public:
13     string RNA;
14     gena(string Gene)
15     {
16         for(int i = 0; i < Gene.size(); i++)
17         {
18             if(Gene[i] == 'A') RNA.push_back('U');
19             else if(Gene[i] == 'C') RNA.push_back('G');
20             else if(Gene[i] == 'G') RNA.push_back('C');
21             else if(Gene[i] == 'T') RNA.push_back('A');
22             else
23                 RNA.push_back('-');
24         }
25     };
26
27     class genb:public gena
28     {
29     public:
30     string Polypeptide;
31     genb(string Gene):gena(Gene)
32     {
33         for(int i = 0; i < RNA.size(); i += 3)
34         {
35             if(RNA.substr(i, 3) == "GCA" ||
36                RNA.substr(i, 3) == "GCC" ||
37                RNA.substr(i, 3) == "GCG" ||
38                RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
39             else if(RNA.substr(i, 3) == "UGC" ||
40                RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
41             else if(RNA.substr(i, 3) == "GAC" ||
42                RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
43             else if(RNA.substr(i, 3) == "GAA" ||
44                RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
45             else if(RNA.substr(i, 3) == "UUC" ||
46                RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
47             else if(RNA.substr(i, 3) == "GGA" ||
48                RNA.substr(i, 3) == "GGC" ||
49                RNA.substr(i, 3) == "GGG" ||
50                RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
51             else if(RNA.substr(i, 3) == "CAC" ||
52                RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
53             else if(RNA.substr(i, 3) == "AUA" ||
54                RNA.substr(i, 3) == "AUC" ||
55                RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
56             else if(RNA.substr(i, 3) == "AAA" ||
57                RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');
58             else if(RNA.substr(i, 3) == "CUA" ||
59                RNA.substr(i, 3) == "CUC" ||
60                RNA.substr(i, 3) == "CUG" ||
61                RNA.substr(i, 3) == "CUU" ||
62                RNA.substr(i, 3) == "UUA" ||
63                RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');

```

```

64         else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
65         else if(RNA.substr(i, 3) == "AAC" ||
66                RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
67         else if(RNA.substr(i, 3) == "CCA" ||
68                RNA.substr(i, 3) == "CCC" ||
69                RNA.substr(i, 3) == "CCG" ||
70                RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
71         else if(RNA.substr(i, 3) == "CAA" ||
72                RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
73         else if(RNA.substr(i, 3) == "AGA" ||
74                RNA.substr(i, 3) == "AGG" ||
75                RNA.substr(i, 3) == "CGA" ||
76                RNA.substr(i, 3) == "CGC" ||
77                RNA.substr(i, 3) == "CGG" ||
78                RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
79         else if(RNA.substr(i, 3) == "AGC" ||
80                RNA.substr(i, 3) == "AGU" ||
81                RNA.substr(i, 3) == "UCA" ||
82                RNA.substr(i, 3) == "UCC" ||
83                RNA.substr(i, 3) == "UCG" ||
84                RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
85         else if(RNA.substr(i, 3) == "ACA" ||
86                RNA.substr(i, 3) == "ACC" ||
87                RNA.substr(i, 3) == "ACG" ||
88                RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
89         else if(RNA.substr(i, 3) == "GUA" ||
90                RNA.substr(i, 3) == "GUC" ||
91                RNA.substr(i, 3) == "GUG" ||
92                RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
93         else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
94         else if(RNA.substr(i, 3) == "UAC" ||
95                RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
96         else Polypeptide.push_back('-');
97     }
98 }
99 };
100
101 class gencz:public genb
102 {
103     public:
104     string Replica;
105     gencz(string Gene, string Template):genb(Gene)
106     {
107         for(int i = 0; i < Template.size(); i++)
108         {
109             if(Template[i] == 'A') Replica.push_back('T');
110             else if(Template[i] == 'C') Replica.push_back('G');
111             else if(Template[i] == 'G') Replica.push_back('C');
112             else if(Template[i] == 'T') Replica.push_back('A');
113             else Replica.push_back('-');
114         }
115     }
116 };
117
118 int main()
119 {
120     string s0a, s0b;
121     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC";
122     s0b="GCATGCGACGAATTCGGACACATAAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG";
123     cout << "Picture 1:\n\n"
124           << s0a << endl
125           << s0b << "\n\n";
126     cout << "Picture 2:\n\n"
127           << s0a << "\n\n"
128           << s0b << "\n\n";
129
130     gencz g0a(s0a.substr(63, 9), s0a);
131     gencz g0b(s0a.substr(63, 9), s0b);
132     cout << "Picture 3:\n\n"
133           << s0a << endl
134           << "
135           << g0a.RNA << "\n\n"
136           << s0b << "\n\n";
137     cout << "Picture 4:\n\n"

```

```

138         << s0a << endl
139         << s0b << "\n\n"
140         << "
141         << g0a.RNA << "\n\n";
142     cout << "Picture 5:\n\n"
143         << s0a << endl
144         << s0b << "\n\n"
145         << "
146         << g0a.RNA << endl
147         << "
148         << g0a.Polypeptide << "\n\n";
149     cout << "Picture 6:\n\n"
150         << s0a << endl
151         << s0b << "\n\n"
152         << "
153         << g0a.RNA << "\n\n"
154         << "
155         << g0a.Polypeptide << "\n\n";
156     cout << "Picture 7:\n\n"
157         << s0a << endl
158         << g0a.Polypeptide << "\n\n"
159         << "
160         << g0b.Polypeptide << endl
161         << s0b << "\n\n";
162     cout << "Picture 8:\n\n"
163         << s0a << endl
164         << g0a.Replica << "\n\n"
165         << g0b.Replica << endl
166         << s0b << "\n\n";
167
168     return 0;
169 }

```

In der Lebenswelt sind nahezu alle Reaktionen zu katalysieren, inklusive alle Reaktionen der Polymerisation. Die wichtigste Reaktion der Polymerisation ist die DNA-Replikation.

Während der DNA-Replikation werden DNA-Molekülstränge voneinander getrennt und jeder Strang wird als Vorlage für die Synthese jeweils anderen Stranges benutzt. Dies ist möglich, weil beide Stränge komplementär zueinander sind. Die Synthese erfolgt nach den Regeln der Basenpaarbindung:

Ein Deoxyribonukleotid mit der Base...	Adenin	Cytosin	Guanin	Thymin
... bindet ein Deoxyribonukleotid mit der Base...	Thymin	Guanin	Cytosin	Adenin

Dementsprechend entsteht auf jeder DNA-Molekülstrang-Vorlage eine komplementäre DNA-Molekülstrang-Replika. Wenn die DNA-Replikation abgeschlossen ist, wird das ursprüngliche DNA-Molekül durch zwei identischen Kopien ersetzt. Beachte: Die DNA-Molekülstränge bestehen die Replikation, ohne verbraucht zu werden.

Hier hat C++-Programm ein Experiment für die Konstruktion eines *in silico* Genexpressionsnetzwerks zu errichten, der die DNA-Transkription, RNA-Translation und Katalyse enthält, wobei die DNA-Replikation katalysiert werden soll. Zeilen von 101 bis 116 enthalten Definition des Objekttyps `gencz` für solche *in silico* GENs. Er ist auch als abgeleitet vom `genb` definiert. Seine Konstruktor-Funktion hat zwei Parameter `Gene` und `Template` des Typs `string`. Der Rumpf der Konstruktor-Funktion enthält Code für die Kontrollstruktur, die die DNA-Replika (hier, `Replica`) auf der DNA-Vorlage (hier `Template`) synthetisiert. Zeichen für *in silico* Deoxyribonukleotide (**A**, **C**, **G**, und **T**) sind entsprechend ihrer konventionellen Bezeichnungen gewählt.

Zeilen von 120 bis 128 enthalten Code, der den Computer instruiert, im Speicher ein *in silico* DNA-Molekül zu konstruieren. Die Molekülstränge müssen zunächst beieinander liegen und dann trennen sie sich. Zeilen 130 und 131 enthalten Anweisungen, die den Computer instruieren, im Speicher zwei *in silico* GENs `g0a` und `g0b` vom Typ `gencz` zu konstruieren, das einen Teilstring von Position 63 bis 71 auf dem ersten Strang des DNA-Moleküls als Argument für seine Konstruktor-Funktion aufnimmt. Zusätzlich nimmt jede Konstruktor-Funktion den entsprechenden DNA-Molekülstrang als zweites Argument.

Erstelle Experiment2-5 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genexpression (hier, DNA-Transkription + RNA-Translation + Katalyse, wobei die DNA-Replikation katalysiert wird) dokumentieren:

```

Picture 1:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 2:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC

GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 3:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
UGUGAUGAG

GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 4:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
UGUGAUGAG

GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

Picture 5:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
UGUGAUGAG
CDE

Picture 6:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
UGUGAUGAG
CDE

Picture 7:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
CDE
CDE
    
```

```
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

Picture 8:

```
CGTACGCTGCTTAAGCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC  
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

```
CGTACGCTGCTTAAGCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC  
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
```

Drücken Sie eine beliebige Taste . . .

Experiment 3

Ziel: Lernen wie *in silico* Genomexpressionsnetzwerke (GENome) zu konstruieren sind.

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment3-1 und füge eine neue Datei Experiment3-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 3-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <vector>
8 #include <iostream>
9 using namespace std;
10
11 class gena
12 {
13 public:
14   string RNA;
15   gena(string Gene)
16   {
17     for(int i = 0; i < Gene.size(); i++)
18     {
19       if(Gene[i] == 'A') RNA.push_back('U');
20       else if(Gene[i] == 'C') RNA.push_back('G');
21       else if(Gene[i] == 'G') RNA.push_back('C');
22       else if(Gene[i] == 'T') RNA.push_back('A');
23       else
24         RNA.push_back('-');
25     }
26   };
27
28   class genb:public gena
29   {
30 public:
31   string Polypeptide;
32   genb(string Gene):gena(Gene)
33   {
34     for(int i = 0; i < RNA.size(); i += 3)
35     {
36       if(RNA.substr(i, 3) == "GCA" ||
37          RNA.substr(i, 3) == "GCC" ||
38          RNA.substr(i, 3) == "GCG" ||
39          RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
40       else if(RNA.substr(i, 3) == "UGC" ||
41              RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
42       else if(RNA.substr(i, 3) == "GAC" ||
43              RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
44       else if(RNA.substr(i, 3) == "GAA" ||
45              RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
46       else if(RNA.substr(i, 3) == "UUC" ||
47              RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
48       else if(RNA.substr(i, 3) == "GGA" ||
49              RNA.substr(i, 3) == "GGC" ||
50              RNA.substr(i, 3) == "GGG" ||
51              RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
52       else if(RNA.substr(i, 3) == "CAC" ||
53              RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
54       else if(RNA.substr(i, 3) == "AUA" ||
55              RNA.substr(i, 3) == "AUC" ||
56              RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
57       else if(RNA.substr(i, 3) == "AAA" ||
58              RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');
59       else if(RNA.substr(i, 3) == "CUA" ||
60              RNA.substr(i, 3) == "CUC" ||
61              RNA.substr(i, 3) == "CUG" ||

```

```

62         RNA.substr(i, 3) == "CUU" ||
63         RNA.substr(i, 3) == "UUA" ||
64         RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');
65     else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
66     else if(RNA.substr(i, 3) == "AAC" ||
67            RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
68     else if(RNA.substr(i, 3) == "CCA" ||
69            RNA.substr(i, 3) == "CCC" ||
70            RNA.substr(i, 3) == "CCG" ||
71            RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
72     else if(RNA.substr(i, 3) == "CAA" ||
73            RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
74     else if(RNA.substr(i, 3) == "AGA" ||
75            RNA.substr(i, 3) == "AGG" ||
76            RNA.substr(i, 3) == "CGA" ||
77            RNA.substr(i, 3) == "CGC" ||
78            RNA.substr(i, 3) == "CGG" ||
79            RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
80     else if(RNA.substr(i, 3) == "AGC" ||
81            RNA.substr(i, 3) == "AGU" ||
82            RNA.substr(i, 3) == "UCA" ||
83            RNA.substr(i, 3) == "UCC" ||
84            RNA.substr(i, 3) == "UCG" ||
85            RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
86     else if(RNA.substr(i, 3) == "ACA" ||
87            RNA.substr(i, 3) == "ACC" ||
88            RNA.substr(i, 3) == "ACG" ||
89            RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
90     else if(RNA.substr(i, 3) == "GUA" ||
91            RNA.substr(i, 3) == "GUC" ||
92            RNA.substr(i, 3) == "GUG" ||
93            RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
94     else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
95     else if(RNA.substr(i, 3) == "UAC" ||
96            RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
97     else
98         Polypeptide.push_back('-');
99     }
100 };
101
102 class genca:public genb
103 {
104     public:
105     char Monomer;
106     genca(string Gene, char a, char b, char c):genb(Gene)
107     {
108         Monomer = c;
109     }
110 };
111
112 class gencb:public genb
113 {
114     public:
115     char Monomer1;
116     char Monomer2;
117     gencb(string Gene, char a, char b, char c):genb(Gene)
118     {
119         Monomer1 = b;
120         Monomer2 = c;
121     }
122 };
123
124 class gencz:public genb
125 {
126     public:
127     string Replica;
128     gencz(string Gene, string Template):genb(Gene)
129     {
130         for(int i = 0; i < Template.size(); i++)
131         {
132             if(Template[i] == 'A') Replica.push_back('T');
133             else if(Template[i] == 'C') Replica.push_back('G');
134             else if(Template[i] == 'G') Replica.push_back('C');
135             else if(Template[i] == 'T') Replica.push_back('A');

```

```

136         else                               Replica.push_back('-');
137     }
138 }
139 };
140
141 class cell
142 {
143     public:
144     vector<string> DNAs;
145     vector<string> tRNAs;
146     vector<string> rRNAs;
147     vector<string> Polypeptides1;
148     vector<string> Polypeptides2;
149     vector<string> Polypeptides3;
150     vector<char> Monomers1;
151     vector<char> Monomers2;
152     vector<char> Monomers3;
153
154     cell(vector<string> v0,
155          vector<string> v1,
156          vector<string> v2,
157          vector<string> v3,
158          vector<string> v4,
159          vector<string> v5,
160          vector<char> v6,
161          vector<char> v7,
162          vector<char> v8)
163     {
164         DNAs = v0;
165         tRNAs = v1;
166         rRNAs = v2;
167         Polypeptides1 = v3;
168         Polypeptides2 = v4;
169         Polypeptides3 = v5;
170         Monomers1 = v6;
171         Monomers2 = v7;
172         Monomers3 = v8;
173
174         for(int i = 0; i < 10; i++)
175         {
176             gena g1(DNAs[0].substr(0, 9)); tRNAs.push_back(g1.RNA);
177             gena g2(DNAs[0].substr(9, 9)); rRNAs.push_back(g2.RNA);
178             genb g3(DNAs[0].substr(18, 9)); Polypeptides1.push_back(g3.Polypeptide);
179             genb g4(DNAs[0].substr(27, 9)); Polypeptides2.push_back(g4.Polypeptide);
180             genb g5(DNAs[0].substr(36, 9)); Polypeptides3.push_back(g5.Polypeptide);
181             gena g6(DNAs[0].substr(45, 9), 'U', 'V', 'X');
182             Monomers1.push_back(g6.Monomer);
183             genb g7(DNAs[0].substr(54, 9), 'W', 'Y', 'Z');
184             Monomers2.push_back(g7.Monomer1);
185             Monomers3.push_back(g7.Monomer2);
186         }
187         vector<string> DNArePLICAs;
188         genCz g0a(DNAs[0].substr(63, 9), DNAs[0]);
189         DNArePLICAs.push_back(g0a.Replica);
190         genCz g0b(DNAs[0].substr(63, 9), DNAs[1]);
191         DNArePLICAs.push_back(g0b.Replica);
192         DNAs.insert(DNAs.begin() + 1, DNArePLICAs.begin(), DNArePLICAs.end());
193     }
194 };
195
196 int main()
197 {
198     string s0a, s0b;
199     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTTCTTCGTGTCATACCATGCGGACACTACTC";
200     s0b="GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG";
201     vector<string> V0;
202     V0.push_back(s0a);
203     V0.push_back(s0b);
204     vector<string> V1(10, "GCAUGCGAC"), V2(10, "GCAUGCGAC");
205     vector<string> V3(10, "HIK"), V4(10, "LMN"), V5(10, "PQR");
206     vector<char> V6(10, 'X'), V7(10, 'Y'), V8(10, 'Z');
207     cout << "Picture 1:" << "\n\n";
208     for(int i = 0; i < V0.size(); i++)
209         cout << V0[i] << endl;

```

```

210     cout << "\n";
211     for(int i = 0; i < V1.size(); i++)
212         cout << V1[i] << " ";
213     cout << "\n\n";
214     for(int i = 0; i < V2.size(); i++)
215         cout << V2[i] << " ";
216     cout << "\n\n";
217     for(int i = 0; i < V3.size(); i++)
218         cout << V3[i] << " ";
219     cout << "\n\n";
220     for(int i = 0; i < V4.size(); i++)
221         cout << V4[i] << " ";
222     cout << "\n\n";
223     for(int i = 0; i < V5.size(); i++)
224         cout << V5[i] << " ";
225     cout << "\n\n";
226     for(int i = 0; i < V6.size(); i++)
227         cout << V6[i] << " ";
228     cout << "\n\n";
229     for(int i = 0; i < V7.size(); i++)
230         cout << V7[i] << " ";
231     cout << "\n\n";
232     for(int i = 0; i < V8.size(); i++)
233         cout << V8[i] << " ";
234     cout << "\n\n";
235
236     cell Cell(V0, V1, V2, V3, V4, V5, V6, V7, V8);
237     cout << "Picture 2:" << "\n\n";
238     for(int i = 0; i < Cell.DNAs.size(); i++)
239         cout << Cell.DNAs[i] << endl;
240     cout << "\n";
241     for(int i = 0; i < Cell.tRNAs.size(); i++)
242         cout << Cell.tRNAs[i] << " ";
243     cout << "\n\n";
244     for(int i = 0; i < Cell.rRNAs.size(); i++)
245         cout << Cell.rRNAs[i] << " ";
246     cout << "\n\n";
247     for(int i = 0; i < Cell.Polypeptides1.size(); i++)
248         cout << Cell.Polypeptides1[i] << " ";
249     cout << "\n\n";
250     for(int i = 0; i < Cell.Polypeptides2.size(); i++)
251         cout << Cell.Polypeptides2[i] << " ";
252     cout << "\n\n";
253     for(int i = 0; i < Cell.Polypeptides3.size(); i++)
254         cout << Cell.Polypeptides3[i] << " ";
255     cout << "\n\n";
256     for(int i = 0; i < Cell.Monomers1.size(); i++)
257         cout << Cell.Monomers1[i] << " ";
258     cout << "\n\n";
259     for(int i = 0; i < Cell.Monomers2.size(); i++)
260         cout << Cell.Monomers2[i] << " ";
261     cout << "\n\n";
262     for(int i = 0; i < Cell.Monomers3.size(); i++)
263         cout << Cell.Monomers3[i] << " ";
264     cout << "\n\n";
265
266     return 0;
267 }

```

Hier hat C++-Programm ein Experiment für die Konstruktion eines *in silico* Genomexpressionsnetzwerks (kurz: GENom) zu errichten. Dieses Lebensmuster ist etwa gleich der Zelle.

Das Genom ist eine begrenzte Menge von Genen und jedes Gen wird in der Regel separat exprimiert, um in das entsprechende Element der Zellstruktur oder -funktion umgewandelt zu werden. Demzufolge ist die Zelle eine Komposition von GENs. In der Zelle agieren GENs miteinander, um sich gegenseitig zu unterstützen. Während der

Exprimierung eines bestimmten Gens ist es gerade die Aufgabe der anderen GENs, die notwendigen Elemente für die Genexpressionsmaschinerie zu liefern. Gemeinsam arbeiten GENs für die vollständige DNA-Replikation. So beginnt die Lebensgeschichte einer einzelnen Zelle mit einer Zelle aber endet mit zwei. Generell beginnt die Lebensgeschichte einer Zelle am Zeitpunkt, wenn die beiden neugeborenen Schwesterzellen die von der Mutter-Zelle geerbte Matrix halbieren und jede Zelle beginnt ihr selbständiges Leben. Was jede neugeborene Zelle zu tun hat, ist genau dasselbe, was ihre Mutter vorher getan hatte: sie startet ihre eigene Genomexpression, die letztendlich in die Genomreplikation und in die darauf folgende Teilung in zwei Tochterzellen mündet.

Hier ist es sinnvoll, mit einem Experiment für die Konstruktion eines *in silico* Genomexpressionsnetzwerks mit dem einfachsten Genom und der einfachsten Lebensgeschichte zu beginnen. Auch Umweltbedingungen müssen zunächst extrem günstig sein: Energie und alle Monomere für die Synthese von DNA, RNA und Polypeptidmolekülen sind in einem Überfluss vorhanden. Zeilen von 141 bis 194 enthalten Definition des neuen Objekttyps `cell` für *in silico* Genomexpressionsnetzwerke (GENome). Für die Einfachheit wird die Anzahl der Gene im Genom auf 8 begrenzt. Sie müssen in der linearen Aufeinanderfolge exprimiert werden, um Inhalte von 9 Pools von *in silico* Chemikalien zu verdoppeln.

Dementsprechend hat die neue Objekttyp `cell` 9 Elementobjekte des Objekttyps `vector`, der in der Datei `vector` der C++-Standardbibliothek definiert ist. Der Objekttyp `vector` ist speziell für die Konstruktion von Objekten konzipiert ist, die einen Pool von anderen Objekten halten und mit ihnen operieren können. In der C++-Standardbibliothek ist der Objekttyp `vector` als Vorlage (Template) definiert. Ein Template nimmt andere Objekttypen als Parameter. Beachte: Die Template-Parameter werden zwischen den spitzen Klammern `<` und `>` angegeben. Hier werden Objekttypen `string` und `char` als Template-Parameter verwendet. Ein Container ist notwendig, um DNA-Molekülstränge zu halten. Zwei Container sind für Pools von RNAs (hier tRNAs und ribosomale RNAs). Drei Container sind für Pools von Polypeptiden (hier RNA-Polymerasen, ribosomale Polypeptide, und Polypeptide, die an der Zellteilung beteiligt sind). Weitere drei Container sind für Pools von Monomeren (hier X, Y, und Z).

Erwartungsgemäß hat die Konstruktor-Funktion des neuen Objekttyps `cell` 9 Parameter. Der Rumpf der Konstruktor-Funktion enthält Code für die Kontrollstruktur, die bestimmt, wie Gene exprimiert werden müssen, um Inhalte von 9 Pools zu verdoppeln.

Zeilen von 198 bis 206 enthalten Code, der den Computer instruiert, im Speicher 9 Pools von *in silico* Chemikalien zu konstruieren. Zeile 236 enthält Anweisung, die den Computer instruiert, im Speicher ein *in silico* Genomexpressionsnetzwerk `cell` vom Typ `cell` zu konstruieren und dabei alle 9 Pools als Argumente für seine Konstruktor-Funktion aufzunehmen.

Erstelle Experiment3-1 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genomexpression dokumentieren:

```
Picture 1:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC

HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK
LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN
PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR
X X X X X X X X X
Y Y Y Y Y Y Y Y Y
Z Z Z Z Z Z Z Z Z

Picture 2:
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG
CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC
GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK
LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN
PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR
X X X X X X X X X X X X X X X X X X
Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
Drücken Sie eine beliebige Taste . . .
```

Experiment 4

Ziel: Lernen wie *in silico* Genommultiplizierungsnetzwerke zu konstruieren sind.

Erstelle eine neue/einen neuen Projektmappe/Projekt Experiment4-1 und füge eine neue Datei Experiment4-1.cpp hinzu. Gebe den folgenden Code in diese Datei ein:

```

1 //Experiment 4-1
2 /*Nikita Tirjatkin
3   Laboratory for in silico life construction
4   Januar 2010*/
5
6 #include <string>
7 #include <vector>
8 #include <deque>
9 #include <iostream>
10 using namespace std;
11
12 class gena
13 {
14 public:
15     string RNA;
16     gena(string Gene)
17     {
18         for(int i = 0; i < Gene.size(); i++)
19         {
20             if(Gene[i] == 'A') RNA.push_back('U');
21             else if(Gene[i] == 'C') RNA.push_back('G');
22             else if(Gene[i] == 'G') RNA.push_back('C');
23             else if(Gene[i] == 'T') RNA.push_back('A');
24             else RNA.push_back('-');
25         }
26     }
27 };
28
29 class genb:public gena
30 {
31 public:
32     string Polypeptide;
33     genb(string Gene):gena(Gene)
34     {
35         for(int i = 0; i < RNA.size(); i += 3)
36         {
37             if(RNA.substr(i, 3) == "GCA" ||
38                RNA.substr(i, 3) == "GCC" ||
39                RNA.substr(i, 3) == "GCG" ||
40                RNA.substr(i, 3) == "GCU") Polypeptide.push_back('A');
41             else if(RNA.substr(i, 3) == "UGC" ||
42                    RNA.substr(i, 3) == "UGU") Polypeptide.push_back('C');
43             else if(RNA.substr(i, 3) == "GAC" ||
44                    RNA.substr(i, 3) == "GAU") Polypeptide.push_back('D');
45             else if(RNA.substr(i, 3) == "GAA" ||
46                    RNA.substr(i, 3) == "GAG") Polypeptide.push_back('E');
47             else if(RNA.substr(i, 3) == "UUC" ||
48                    RNA.substr(i, 3) == "UUU") Polypeptide.push_back('F');
49             else if(RNA.substr(i, 3) == "GGA" ||
50                    RNA.substr(i, 3) == "GGC" ||
51                    RNA.substr(i, 3) == "GGG" ||
52                    RNA.substr(i, 3) == "GGU") Polypeptide.push_back('G');
53             else if(RNA.substr(i, 3) == "CAC" ||
54                    RNA.substr(i, 3) == "CAU") Polypeptide.push_back('H');
55             else if(RNA.substr(i, 3) == "AUA" ||
56                    RNA.substr(i, 3) == "AUC" ||
57                    RNA.substr(i, 3) == "AUU") Polypeptide.push_back('I');
58             else if(RNA.substr(i, 3) == "AAA" ||
59                    RNA.substr(i, 3) == "AAG") Polypeptide.push_back('K');
60             else if(RNA.substr(i, 3) == "CUA" ||
61                    RNA.substr(i, 3) == "CUC" ||
62                    RNA.substr(i, 3) == "CUG" ||
63                    RNA.substr(i, 3) == "CUU" ||

```

```

64         RNA.substr(i, 3) == "UUA" ||
65         RNA.substr(i, 3) == "UUG") Polypeptide.push_back('L');
66     else if(RNA.substr(i, 3) == "AUG") Polypeptide.push_back('M');
67     else if(RNA.substr(i, 3) == "AAC" ||
68         RNA.substr(i, 3) == "AAU") Polypeptide.push_back('N');
69     else if(RNA.substr(i, 3) == "CCA" ||
70         RNA.substr(i, 3) == "CCC" ||
71         RNA.substr(i, 3) == "CCG" ||
72         RNA.substr(i, 3) == "CCU") Polypeptide.push_back('P');
73     else if(RNA.substr(i, 3) == "CAA" ||
74         RNA.substr(i, 3) == "CAG") Polypeptide.push_back('Q');
75     else if(RNA.substr(i, 3) == "AGA" ||
76         RNA.substr(i, 3) == "AGG" ||
77         RNA.substr(i, 3) == "CGA" ||
78         RNA.substr(i, 3) == "CGC" ||
79         RNA.substr(i, 3) == "CGG" ||
80         RNA.substr(i, 3) == "CGU") Polypeptide.push_back('R');
81     else if(RNA.substr(i, 3) == "AGC" ||
82         RNA.substr(i, 3) == "AGU" ||
83         RNA.substr(i, 3) == "UCA" ||
84         RNA.substr(i, 3) == "UCC" ||
85         RNA.substr(i, 3) == "UCG" ||
86         RNA.substr(i, 3) == "UCU") Polypeptide.push_back('S');
87     else if(RNA.substr(i, 3) == "ACA" ||
88         RNA.substr(i, 3) == "ACC" ||
89         RNA.substr(i, 3) == "ACG" ||
90         RNA.substr(i, 3) == "ACU") Polypeptide.push_back('T');
91     else if(RNA.substr(i, 3) == "GUA" ||
92         RNA.substr(i, 3) == "GUC" ||
93         RNA.substr(i, 3) == "GUG" ||
94         RNA.substr(i, 3) == "GUU") Polypeptide.push_back('V');
95     else if(RNA.substr(i, 3) == "UGG") Polypeptide.push_back('W');
96     else if(RNA.substr(i, 3) == "UAC" ||
97         RNA.substr(i, 3) == "UAU") Polypeptide.push_back('Y');
98     else
99         Polypeptide.push_back('-');
100     }
101 };
102
103 class genca:public genb
104 {
105     public:
106         char Monomer;
107         genca(string Gene, char a, char b, char c):genb(Gene)
108         {
109             Monomer = c;
110         }
111 };
112
113 class gencb:public genb
114 {
115     public:
116         char Monomer1;
117         char Monomer2;
118         gencb(string Gene, char a, char b, char c):genb(Gene)
119         {
120             Monomer1 = b;
121             Monomer2 = c;
122         }
123 };
124
125 class gencz:public genb
126 {
127     public:
128         string Replica;
129         gencz(string Gene, string Template):genb(Gene)
130         {
131             for(int i = 0; i < Template.size(); i++)
132             {
133                 if(Template[i] == 'A') Replica.push_back('T');
134                 else if(Template[i] == 'C') Replica.push_back('G');
135                 else if(Template[i] == 'G') Replica.push_back('C');
136                 else if(Template[i] == 'T') Replica.push_back('A');
137                 else

```

```

138     }
139   }
140 };
141
142 class cell
143 {
144   public:
145     vector<string> DNAs;
146     vector<string> tRNAs;
147     vector<string> rRNAs;
148     vector<string> Polypeptides1;
149     vector<string> Polypeptides2;
150     vector<string> Polypeptides3;
151     vector<char> Monomers1;
152     vector<char> Monomers2;
153     vector<char> Monomers3;
154
155     cell(vector<string> v0,
156         vector<string> v1,
157         vector<string> v2,
158         vector<string> v3,
159         vector<string> v4,
160         vector<string> v5,
161         vector<char> v6,
162         vector<char> v7,
163         vector<char> v8)
164     {
165         DNAs = v0;
166         tRNAs = v1;
167         rRNAs = v2;
168         Polypeptides1 = v3;
169         Polypeptides2 = v4;
170         Polypeptides3 = v5;
171         Monomers1 = v6;
172         Monomers2 = v7;
173         Monomers3 = v8;
174
175         for(int i = 0; i < 10; i++)
176         {
177             gena g1(DNAs[0].substr(0, 9)); tRNAs.push_back(g1.RNA);
178             gena g2(DNAs[0].substr(9, 9)); rRNAs.push_back(g2.RNA);
179             genb g3(DNAs[0].substr(18, 9)); Polypeptides1.push_back(g3.Polypeptide);
180             genb g4(DNAs[0].substr(27, 9)); Polypeptides2.push_back(g4.Polypeptide);
181             genb g5(DNAs[0].substr(36, 9)); Polypeptides3.push_back(g5.Polypeptide);
182             gena g6(DNAs[0].substr(45, 9), 'U', 'V', 'X');
183             Monomers1.push_back(g6.Monomer);
184             genb g7(DNAs[0].substr(54, 9), 'W', 'Y', 'Z');
185             Monomers2.push_back(g7.Monomer1);
186             Monomers3.push_back(g7.Monomer2);
187         }
188         vector<string> DNArePLICAS;
189         genz g0a(DNAs[0].substr(63, 9), DNAs[0]);
190             DNArePLICAS.push_back(g0a.Replica);
191         genz g0b(DNAs[0].substr(63, 9), DNAs[1]);
192             DNArePLICAS.push_back(g0b.Replica);
193         DNAs.insert(DNAs.begin() + 1, DNArePLICAS.begin(), DNArePLICAS.end());
194     }
195 };
196
197 class cp
198 {
199   public:
200     deque<cell> Cells;
201     cp(cell ccell)
202     {
203         Cells.push_back(ccell);
204
205         while(Cells.size() <= Cells.max_size())
206         {
207             ccell = Cells.front();
208
209             cout << "\nCell Number: " << Cells.size() << "\n\n";
210             for(int i = 0; i < ccell.DNAs.size(); i++)
211                 cout << ccell.DNAs[i] << " ";

```

```

212     cout << "\n\n";
213     for(int i = 0; i < ccell.tRNAs.size(); i++)
214         cout << ccell.tRNAs[i] << " ";
215     cout << "\n\n";
216     for(int i = 0; i < ccell.rRNAs.size(); i++)
217         cout << ccell.rRNAs[i] << " ";
218     cout << "\n\n";
219     for(int i = 0; i < ccell.Polypeptides1.size(); i++)
220         cout << ccell.Polypeptides1[i] << " ";
221     cout << "\n\n";
222     for(int i = 0; i < ccell.Polypeptides2.size(); i++)
223         cout << ccell.Polypeptides2[i] << " ";
224     cout << "\n\n";
225     for(int i = 0; i < ccell.Polypeptides3.size(); i++)
226         cout << ccell.Polypeptides3[i] << " ";
227     cout << "\n\n";
228     for(int i = 0; i < ccell.Monomers1.size(); i++)
229         cout << ccell.Monomers1[i] << " ";
230     cout << "\n\n";
231     for(int i = 0; i < ccell.Monomers2.size(); i++)
232         cout << ccell.Monomers2[i] << " ";
233     cout << "\n\n";
234     for(int i = 0; i < ccell.Monomers3.size(); i++)
235         cout << ccell.Monomers3[i] << " ";
236     cout << "\n\n";
237
238     Cells.pop_front();
239
240     vector<string> lDNAs(ccell.DNAs.begin(),
241                        ccell.DNAs.begin() +
242                        ccell.DNAs.size()/2);
243     vector<string> rDNAs(ccell.DNAs.begin() +
244                        ccell.DNAs.size()/2,
245                        ccell.DNAs.end());
246     vector<string> ltRNAs(ccell.tRNAs.begin(),
247                          ccell.tRNAs.begin() +
248                          ccell.tRNAs.size()/2);
249     vector<string> rtRNAs(ccell.tRNAs.begin() +
250                          ccell.tRNAs.size()/2,
251                          ccell.tRNAs.end());
252     vector<string> lrRNAs(ccell.rRNAs.begin(),
253                          ccell.rRNAs.begin() +
254                          ccell.rRNAs.size()/2);
255     vector<string> rrRNAs(ccell.rRNAs.begin() +
356                          ccell.rRNAs.size()/2,
257                          ccell.rRNAs.end());
258     vector<string> lPolypeptides1(ccell.Polypeptides1.begin(),
259                                  ccell.Polypeptides1.begin() +
260                                  ccell.Polypeptides1.size()/2);
261     vector<string> rPolypeptides1(ccell.Polypeptides1.begin() +
262                                  ccell.Polypeptides1.size()/2,
263                                  ccell.Polypeptides1.end());
264     vector<string> lPolypeptides2(ccell.Polypeptides2.begin(),
265                                  ccell.Polypeptides2.begin() +
266                                  ccell.Polypeptides2.size()/2);
267     vector<string> rPolypeptides2(ccell.Polypeptides2.begin() +
268                                  ccell.Polypeptides2.size()/2,
269                                  ccell.Polypeptides2.end());
270     vector<string> lPolypeptides3(ccell.Polypeptides3.begin(),
271                                  ccell.Polypeptides3.begin() +
272                                  ccell.Polypeptides3.size()/2);
273     vector<string> rPolypeptides3(ccell.Polypeptides3.begin() +
274                                  ccell.Polypeptides3.size()/2,
275                                  ccell.Polypeptides3.end());
276     vector<char> lMonomers1(ccell.Monomers1.begin(),
277                             ccell.Monomers1.begin() +
278                             ccell.Monomers1.size()/2);
279     vector<char> rMonomers1(ccell.Monomers1.begin() +
280                             ccell.Monomers1.size()/2,
281                             ccell.Monomers1.end());
282     vector<char> lMonomers2(ccell.Monomers2.begin(),
283                             ccell.Monomers2.begin() +
284                             ccell.Monomers2.size()/2);
285     vector<char> rMonomers2(ccell.Monomers2.begin() +

```

```

286         ccell.Monomers2.size()/2,
287         ccell.Monomers2.end());
288     vector<char> lMonomers3(ccell.Monomers3.begin(),
289         ccell.Monomers3.begin() +
290         ccell.Monomers3.size()/2);
291     vector<char> rMonomers3(ccell.Monomers3.begin() +
292         ccell.Monomers3.size()/2,
293         ccell.Monomers3.end());
294
295     cell lcell(lDNAs,
296         ltrRNAs,
297         lrRNAs,
298         lPolypeptides1,
299         lPolypeptides2,
300         lPolypeptides3,
301         lMonomers1,
302         lMonomers2,
303         lMonomers3);
304     cell rcell(rDNAs,
305         rtrRNAs,
306         rrRNAs,
307         rPolypeptides1,
308         rPolypeptides2,
309         rPolypeptides3,
310         rMonomers1,
311         rMonomers2,
312         rMonomers3);
313     Cells.push_back(lcell);
314     Cells.push_back(rcell);
315 }
316 }
317 };
318
319 int main()
320 {
321     string s0a, s0b;
322     s0a="CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC";
323     s0b="GCATGCGACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG";
324     vector<string> V0;
325     V0.push_back(s0a);
326     V0.push_back(s0b);
327     vector<string> V1(10, "GCAUGCAC"), V2(10, "GCAUGCAC");
328     vector<string> V3(10, "HIK"), V4(10, "LMN"), V5(10, "PQR");
329     vector<char> V6(10, 'X'), V7(10, 'Y'), V8(10, 'Z');
330     cell Cell(V0, V1, V2, V3, V4, V5, V6, V7, V8);
331
332     cp CellProgression(Cell);
333
334     return 0;
335 }

```

Hier hat C++-Programm ein Experiment für die Konstruktion eines *in silico* Genommultiplizierungsnetzwerks zu errichten. Dieses Lebensmuster ist etwa gleich der individuellen Zellprogression.

Die Genomreplikation bei der Genomexpression führt zur Genomverdoppelung. Eine progressive Genomreplikation ist in der Regel mit einer progressiven Zellvermehrung verbunden. Dabei entsteht die Zellprogression: eine Zelle, zwei Zellen, vier Zellen, acht Zellen, und so weiter. Die gesamte Lebenswelt ist eine einzige Zellprogression, die aus einer einzigen Urzelle entstanden ist und 3 oder 4 Milliarden Jahren ununterbrochener Geschichte hat. Sie kann als allgemeine Zellprogression genannt werden. Die heutige Biosphäre ist nur ein dünnes Abschnitt von ihr, die sichtbare Spitze eines Eisbergs im Ozean der Zeit. Der antike Teil dieses gigantischen Lebensmusters hat kaum Spuren hinterlassen. Die Genommultiplizierung ist eng mit der Genomdiversifizierung verbunden, die auf spontane Mutationen aber auch auf stark regulierte

Sequenzübertragung zurückzuführen ist. Die Genomdiversifizierung produziert Zellprogressionen, jede von denen jeweils durch ein bestimmtes Genom zu unterscheiden ist. Sie können als individuelle Zellprogressionen bezeichnet werden. Dementsprechend ist die allgemeine Zellprogression als eine wachsende Zusammensetzung von einer wachsenden Zahl von einzelnen individuellen Zellprogressionen anzusehen.

Hier ist es sinnvoll, mit einem Experiment für die Konstruktion eines *in silico* Genommultiplizierungsnetzwerks mit dem einfachsten Genom und der einfachsten Lebensgeschichte zu beginnen. Auch Umweltbedingungen müssen zunächst extrem günstig sein: Energie und alle Monomere für die Synthese von DNA, RNA und Polypeptidmolekülen sind in einem Überfluss vorhanden. Zeilen von 197 bis 317 enthalten Definition des neuen Objekttyps `cp` für *in silico* Genommultiplizierungsnetzwerke.

Im Wesentlichen ist eine individuelle Zellprogression ein binärer Baum. Dementsprechend hat der neue Objekttyp `cp` eine Konstruktor-Funktion, die den speziellen Container `cells` mit Zellen in der so genannten in-level Order füllt, so dass jede Mutterzelle durch ihre beiden Tochterzellen ersetzt wird, sobald sie sich teilt. Der Container `cells` ist ein Objekt des Objekttyps `deque`, der in der Datei `deque` der C++-Standardbibliothek definiert ist. Der Objekttyp `deque` ist speziell für die Konstruktion von Objekten konzipiert, die einen Pool von anderen Objekten halten und mit ihnen operieren können. Ähnlich wie der Objekttyp `vector` ist der Objekttyp `deque` als Vorlage (Template) definiert und kann andere Objekttypen als Parameter aufnehmen. Hier wird der Objekttyp `cell` als Template-Parameter verwendet. Der Objekttyp `deque` ist sehr gut für die Konstruktion eines Containers geeignet, der in der in-level Order gefüllt wird, so dass ein binärer Baum entstehen wird. Der Rumpf der Konstruktor-Funktion vom Objekttyp `cp` enthält Code für die Kontrollstruktur, die bestimmt, wie der Container `cells` mit Zellen in der in-level Order gefüllt werden soll. Beachte: Die Ausführung der `while`-Schleife ist durch die Bedingung beschränkt:

```
Cells.size() <= Cells.max_size()
```

wo `Cells.max_size()` durch eine andere angemessene Variable oder Zahl ersetzt werden kann, um die Ausführungszeit zu verkürzen.

Darüber hinaus enthält die Konstruktor-Funktion des neuen Objekttyps `cp` einen einfachen Code für *in silico* Instrumentarium.

Zeilen von 321 bis 329 enthalten Code, der den Computer instruiert, im Speicher 9 Pools von *in silico* Chemikalien zu konstruieren. Zeile 330 enthält Anweisung, die den Computer instruiert, im Speicher ein *in silico* Genomexpressionsnetzwerk `cell` vom Typ `cell` zu konstruieren und dabei alle 9 Pools als Argumente für seine Konstruktor-Funktion aufzunehmen. Zeile 332 enthält Anweisung, die den Computer instruiert, im Speicher ein *in silico* Genommultiplizierungsnetzwerk `CellProgression` vom Typ `cp` zu konstruieren und dabei den Objekt `cell` als Argument für seine Konstruktor-Funktion aufzunehmen.

Erstelle Experiment4-1 und führe den ausführbaren Maschinencode aus.

Das Konsole-Fenster wird Bilder zeigen, die das Experiment für die *in silico* Genommultiplizierung dokumentieren. Allerdings fließen Bilder zu schnell. Schließe das Konsole-Fenster, gehe zu der Zeile 205 der Datei Experiment4-1.cpp und ersetze die Bedingung

```
Cells.size() <= Cells.max_size()
```

durch

```
Cells.size() <= 7
```

Erstelle Experiment4-1 erneut und führe den ausführbaren Maschinencode aus. Das Konsole-Fenster wird nur erste 7 Bilder zeigen:

```
Cell Number: 1

CGTACGCTGCTTAAGCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCG
ACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG CGTACGCTGCTTAA
GCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCGACGAATTCGGACAC
ATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK

LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN

PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR

X X X X X X X X X X X X X X X X X X
Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z

Cell Number: 2

CGTACGCTGCTTAAGCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCG
ACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG CGTACGCTGCTTAA
GCCTGTGTATTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCGACGAATTCGGACAC
ATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK
```

LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN

PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR

X X

Y Y

Z Z

Cell Number: 3

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCG
ACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG CGTACGCTGCTTAA
GCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCGACGAATTCGGACAC
ATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC

GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK

LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN

PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR

X X

Y Y

Z Z

Cell Number: 4

CGTACGCTGCTTAAGCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCG
ACGAATTCGGACACATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG CGTACGCTGCTTAA
GCCTGTGTATTTTGATTACTTGGGTGTTTCTTCGTGTCATACCATGCGGACACTACTC GCATGCGACGAATTCGGACAC
ATAAACTAATGAACCCACAAAGAAGCACAGTATGGTACGCCTGTGATGAG

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC

GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GCAUGCGAC GCAUGCGAC GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK

LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN

PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR

X X

Y Y

Z Z


```
GCAUGCGAC GCAUGCGAC GCAUGCGAC GCAUGCGAC
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA
GAAUUCGGA GAAUUCGGA GAAUUCGGA GAAUUCGGA

HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK HIK
LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN LMN
PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR PQR
X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
Drücken Sie eine beliebige Taste . . .
```

© 2010 Nikita Tirjatkin