Abend 2, 3. Semester
Speicherreservierung zur Laufzeit

Ziel, Inhalt

- Wir sind ab heute in der Lage eine beliebige Anzahl Objekte zur Laufzeit zu erzeugen, ohne deren Anzahl durch feste Konstanten festlegen zu müssen

Der operator new zur Speicherreservierung

Betrachte die Klasse MyString aus der letzten Lektion. Vermutlich hast Du genau wie ich eine Konstante MaxChar (oder so) definiert, um anzugeben wie lange ein MyString-Objekt maximal sein kann, das heisst wieviele Zeichen maximal in unserem char-Array drin sein können. Mit dem operator-new entfällt diese Einschränkung. Mit dem Schlüsselwirt *new* können wir zur Laufzeit, aufgrund der Benutzereingaben neuen Speicher reservieren.

Verwendung von new

Mit new können wir Speicher reservieren. Dabei müssen wir bestimmen wieviel Speicher wir wollen. Das Geschieht durch eine Angabe eines Datentypen nach dem Schlüsselwort new. new verhält sich dabei wie eine Funktion, die einen Zeiger auf einen Speicherbereich zurückgibt. Beispiel:

```
double* zeiger = new double;
```

zeiger ist also ein Zeiger auf einen *double*. Für andere Datentypen sieht das analog aus:

```
int* p1 = new int;
float* p2 = new float;
unsigned long* p3 = new unsigned long;
```

Der Zeiger, den man erhält zeigt danach auf einen Speicherblock, der ein uninitialisiertes Element des Datentypen enthält, den man beantragt hat. Um dieses Element direkt zu initialisieren gibt es die Möglichkeit in Klammern anzugeben, womit das Element initialisiert werden kann.

```
double* zeiger = new double(6.0);
int* p1 = new int(34);
float* p2 = new float(2.5);
unsigned long* p3 = new unsigned long(2345656);
```

Verwendung von delete

Der Speicher, den man mit new beantragt muss mit delete wieder freigegeben werden, wenn man ihn nicht mehr braucht. Andernfalls kann der Speicher knapp werden oder ganz ausgehen. Um Speicher, den man reserviert hat wieder freizugeben muss man *delete* aufrufen und den Zeiger übergeben, den man vorher von *new* zurückerhalten hat.

```
delete zeiger;
delete p1;
delete p2;
delete p3;
```

Übung new und delete mit elementaren Datentypen

In Form einer kleinen Hands-on Übung erzeugen wir ein neues Projekt mit unserer Entwicklungsumgebung. Wir brauchen vorerst nur eine main.cpp Datei, in der wir eine main-Funktion einfügen.
Erzeuge im main einige double's, int's und andere Objekte von eingebauten Datentypen, wie im Beispiel oben, ohne diese zu initialisieren.
Kompiliere das Projekt und schau Dir die Zeiger im Debugger im Einzelschrittmodus an. Du kannst Dir auch den Speicher anzeigen lassen, den du mit *new* allozierst. Öffne bei laufendem Debugger das Debug-Fenster „Speicher"(„Memory"). Ziehe den Zeiger, den du untersuchen willst in das „Speicher"-Fenster.
Beachte auch was passiert wenn du den Speicher wieder mit delete freigibst. Beachte, dass diese Dinge nur geschehen, wenn du eine Debug-Version deines Programms erzeugst. Ein sogenannter Release-Build zeigt ein anderes Verhalten.

Heap- und Stack-Speicher

Wenn man von new und delete spricht, ist es unvermeidbar vom sogenannten Heap- und Stack-Speicher zu sprechen. Erzeugt man eine Variable auf die gewohnte Art:

```
int test1 = 0;
int test2 = 10;
```

werden diese auf dem sogenannten Stack-Speicher angelegt. Der Stack (Stapel) ist ein zusammenhängender Speicherbereich, der von unten nach oben verbraucht wird. Er wird wird auch für andere Dinge wie Rücksprungadressen bei Funktionsaufrufen und anderes gebraucht. Hier ein kleines Beispiel, das auch zeigt wie man durch unvorsichtiges Programmieren den Stack zerstört.

```
void foo()
{
    char test[] = "";
    test[4] = 0;
}
int main()
{
    unsigned long test1 = 0x10101010;
    unsigned long test2 = 0x20202020;
    unsigned long test3 = 0x30303030;

    char test[] = "abc";
    test[12] = 'z';

    foo();

    return 0;
}
```

Speicher den man mit new alloziert (reserviert) wird auf dem Heap (Halde) geholt. Dabei ist es der C++Speicherverwaltung überlassen wie sie den Speicher aufteilt und zur Verfügung stellt. Dieser Heap hat beim Programmstart eine gewisse Grösse. Er kann aber auch wachsen, dabei wird vom Betriebssystem neuer Speicher angefordert.
Beachte auch dass bei einer Zeile wie dieser:

```
int* test = new int(0);
```

zweimal Speicher verbraucht wird. Einerseits die Variable *test*, die ein Zeiger auf einen int ist und auf dem Stack erzeugt wird. Andererseits wird ja dynamisch auf dem Heap Platz für einen *int* alloziert.
Übrigens sind Zeigervariablen immer gleich lang, unabhängig davon ob sie auf einen int, double oder einen selbstdefinierten Datentypen zeigen. Sie beinhalten ja immer nur eine Speicheradresse, die bei 32-Bit Windows auch 32 Bit lang sind, denn die Bezeichnung 32 Bit bezieht sich auf die Speicher-Adressierung!

Tipp : Gelöschte Zeiger auf 0 setzen

Setze Zeiger, die du mit delete gelöscht hast wieder auf 0. Angenommen die reservierst mit *new* Platz für einen double:

```
double* pd = new double(3.0);
cout << pd << endl; // Zeiger ausgeben, also Adresse
delete pd;          // Speicher wieder freigeben
cout << pd << endl; // pd zeigt immer noch auf
                    // gelöschten Speicher
```

Wenn du dir angewöhnst Zeiger nach dem Löschen immer auf 0 zu setzen, hast du sozusagen einen Indikator, dass der Speicher freigegeben wurde. Es ist übrigens erlaubt *delete* aufzurufen auch wenn der Zeiger 0 ist. Es passiert dabei nichts. Wenn Du hingegen einen Zeiger zweimal löschst -> CRASH. Probier es aus!

Übung viele Objekte

Erzeuge ein neues Projekt, oder nimm das Projekt, das du für die erste Übung verwendet hast und erzeuge einen neue Klasse Person. Halte diese Klasse einfach mit nur zwei Datenelementen Vornamen und Nachnamen. Schreibe einen Default-Konstruktor. Überschreibe die operatoren << und >> mittels zweier friend-Funktionen (siehe letzter Abend, Operatoren überladen).
Schreibe nun ein kleines main, welches dem Benutzer ermöglicht solange er einverstanden ist, neue Personen zu definieren. Erzeuge die Objekte dabei dynamisch mit *new*. Du musst sie jetzt noch nicht löschen (keine *delete*). Am Ende sind ein Haufen Personen im Speicher... Beim Verlassen des Programm wird der Speicher übrigens dann doch noch freigegeben.
In einem zweiten Schritt erzeugst Du zuerst ein Array von *Person*-Zeigern!

```
Person* Personen[10] = {0}; // Array mit zehn Personen-Zeigern
```

Beachte, dass diese Person-Zeiger mit 0 initialisiert sind. Schreibe nun eine Schleife, die diese Objekte mit *new* erzeugt und lass den Benutzer diese Daten eingeben (mit dem >> operator). Danach kannst du eine weitere Schleife schreiben, die die Daten in eine Datei schreibt (mit << operator) und die einzelnen Objekte mit *delete* löscht.

Der Operator new[]

Es ist auch möglich mehrere Objekte auf einmal dynamisch zu erzeugen.

```
double* daten = new double[20];
```

So wird dynamisch ein Array von 20 double Werten erzeugt. Anstelle von 20 kann auch eine Variable stehen, was nur bei dynamisch allozierten Arrays funktioniert!
Hier ein kleines Beispiel:

```
#include <iostream>

using namespace std;

int main()
{
    unsigned int anzahl = 0;

    cout << "Wieviele Daten wollen sie eingeben ? ";
    cin >> anzahl;

    double* daten = new double[anzahl];

    for(unsigned int i = 0; i < anzahl; ++i)
    {
        cout << i+1 << ": ";
        cin >> daten[i];
    }

    for(i = 0; i < anzahl; ++i)
    {
        cout << i+1 << ": " << daten[i] << endl;
    }

    delete[] daten;

    return 0;
}
```

Beachte auch wie das Array wieder freigegeben werden muss!

Der operator delete[]

Speicher, der mit new[] alloziert (reserviert) wurde muss mit delete[] wieder freigegeben werden. Dabei ist es nicht nötig (möglich) in den eckigen Klammern anzugeben wieviele Objekte fregegeben werden müssen. Hier noch einmal im Beispiel:

```
char* buchstaben = new char[23];
delete[] buchstaben;
```

Dynamische Elemente

Mit dem Wissen, das wir jetzt gesammelt haben, sind wir in der Lage unsere MyString-Klasse bedeutend zu verbessern.
In der Klasse MyString haben wir als Datenelement ein char Array etwa in dieser Form:

```
class MyString
{
        ....
    private:
        char m_daten[MaxChar];
};
```

Um unsere Klasse zu verbessern ist es möglich ein char-Array variabler Länge zu verwenden. Die Klassendefinition sieht dann etwa so aus:

```
class MyString
{
         ....
    private:
        char* m_daten;
};
```

Beachte, dass m_daten immer noch gleich verwendbar sind, unter anderem ist es möglich mit [] – operator (index-operator) auf einzelne Elemente (hier char's) zuzugreifen.
Der m_daten-Zeiger wird dabei je nach Bedarf mit *new* definiert. Dabei werden mit *new* immer soviele char's reserviert, wie gerade gebraucht werden. Ein solches Element ist nun ein dynamisches Element.
Es gibt dabei einige Dinge zu beachten!

Übung Klasse MyString

Es gilt jetzt also unsere Klasse MyString auf den neuesten Stand der Technik zu bringen. Ersetze das Array fixer Grösse mit dem Zeiger den du mittels *new* auf genügend Speicher zeigen lässt.
Beginne bei den Konstruktoren. Initialisiere den Zeiger auf 0!
Je nach Konstruktor reicht das bereits.
Beim Konstruktor mit Parametern musst du genügend Speicher reservieren und die Daten aus dem *const char**(C-String) zu kopieren.
Jetzt brauchst Du auch einen Destruktor, denn der Speicher muss wieder freigegeben werden und das geht am besten im Destruktor.