PROGRAMMIERUNG 2

INTERMISSION

MEMORY MANAGEMENT AND EXCEPTION HANDLING

Shared/Smart Pointer - Einführung

In C++ haben wir Pointer kennengelernt.

Was ist das Problem mit Pointern?

Du musst sie managen!

Wann ist das Problematisch?

Immer wenn wir Speicher auf dem Heap allokieren…

Anmerkung: Im folgenden werden klassische pointer als raw pointer bezeichnet.

Einführung

Wann immer wir also dynamisch Speicher allokieren (z.B.: new _object)

müssen wir daran denken, den Speicher wieder freizugeben (delete _object)

Wenn man das nicht macht, erhält man einen memory leak!

Für einen Programmierer bedeutet das, er muss selbst einen Überblick über die allokierten Ressourcen haben.

Wenn man die falschen Operatoren aufruft, kann dies zu undefinierten Verhalten führen.

Einführung

Ein weiteres Problem ist „ownership“!

Wenn du einen Pointer bekommst, was ist sein Inhalt?

Ist es dynamisch allokierter Speicher?

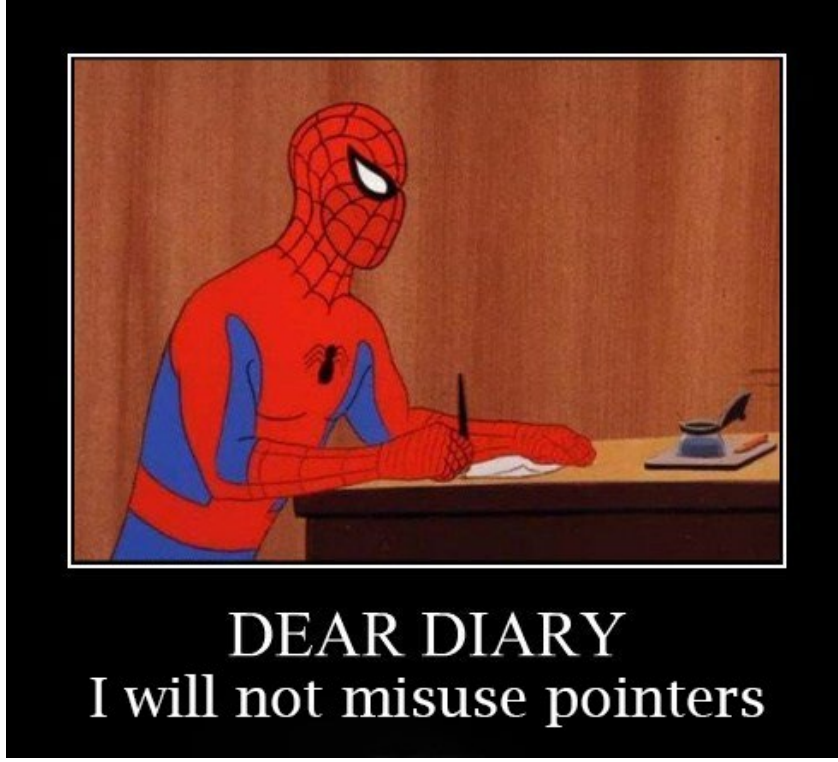
Wer kümmert sich um das aufräumen?

Man kann dieses Wissen nicht vom Rückgabewert alleine extrapolieren…

Zusammengefasst

Pointer sind extrem mächtige Werkzeuge.

...and with great power comes great responsibility...



- Ein kleiner Fehler oder eine Unachtsamkeit, kann zu schwer auffindbaren Fehlern führen.
- Eine sorgsame Planung des Codes ist notwendig.
- Ownership und Zuständigkeiten sind aufwendig zu kommunizieren.

Smart Pointer

Wurde entwickelt um die genannten Probleme zu lösen.

Kurz gesagt, smart pointer bieten automatisches memory management.

Wie realisieren smart pointer das:

relative simpel → wenn ein smart pointer nicht mehr benötigt wird, verliert er seinen scope, m.a.W wird deallokiert.

Smart Pointer

Smart Pointen in C++ lassen sich wie klassische Pointer nutzen.

Bekannte Operatoren wie → und *, werden von diesen Klassen implementiert, um bekanntest Verhalten zu ermöglichen.

Bekannte Syntax bleibt erhalten.

Die Implementierung als Klasse erleichtert die Bereitstellung der gewünschten Funktionalität.

Smart Pointer

- Freigabe des allokierten Speichers im Destruktor. Hierdurch wird automatisch der allokierte Speicher freigegeben, sobald der pointer zerstört wird.
- Da es sich bei diesem Pointer um eine Klasse handelt, lassen sich leicht neue Variationen implementieren.
- Wenn man einen smart pointer übergeben bekommt, lässt sich leichter der Typ bestimmen oder welche Funktionalitäten er mitbringt.
- shared pointer erlauben zudem die dynamisch allokierte Ressource zu teilen, dies wird durch eine intern geführten reference counter implementiert (Stichwort Klassenattribut).

Smart Pointer

Seit C++11 sind smart und shared pointer in der STL erhältlich.

`std::unique_ptr` — ein smart pointer der eine dynamsch allokierte Ressource verwaltet.
Kein anderer smart Pointer hat Zugriff darauf.

`std::shared_ptr` — ein smart pointer der eine geteilte, dynamsch allokierte Ressource verwaltet.

`std::weak_ptr` — ein spezieller shared pointer der den internen counter nicht erhöht, aber auch keinen direkten Zugriff erlaubt.
Man kann so einen Pointer verwenden um zu sehen ob eine Ressource noch verwendet wird.

Bei älteren Version, musste man auf externe Bibliotheken zugreifen (boost).

Shared Pointer in action

Schauen wir uns std::shared pointers mal genauer an.

Rückblick:

Erlaubt mehrere Referenzierungen derselben Ressource.

Pointers

```
bool pointerExample(ExcampleType *pEx)
{
return pEx->execute();
}


int main()
{
ExampleType* pEx = new ExampleType;
pointerExample(pEx);
}
```

Shared Pointers

```
bool pointerExample(std::shared_ptr<ExampleType>
spEx)
{
return spEx->execute();
}


int main()
{
std::shared_ptr<ExampleType> spEx(new ExampleType);
pointerExample(spEx);
}
```

Übung 1

a) Lest die Doku zu std::shared_ptr

b) Überarbeitet eure B-Baum Generierungsklassen und Funktionen so, dass alle auf dem Heap generierter pointer durch shared pointer ersetzt werden.