Objektorientierte Programmierung
(Winter 2006/2007)

Kapitel 11: Dynamische Speicherverwaltung

- Lebensdauer von Objekten
- new, delete
- Beispiel: Verkettete Listen

Lokale Variablen (1)

- Der Speicherplatz für eine lokale Variable wird reserviert, wenn die Deklaration ausgeführt wird, und am Ende des Blockes wieder freigegeben.

  Das gilt nur für "normale" lokale Variablen (Speicherklasse "automatic"), nicht für statische lokale Variablen.
  Wahrscheinlich wird der Compiler den gesamten Speicherplatz, der für lokale Variablen nötig ist, am Beginn der Prozedur reservieren, und am Ende der Prozedur auf einmal freigeben. Dies ist effizienter, als für jede Variable einzeln Speicherplatz zu reservieren. Er könnte aber trotzdem die gleiche Adresse für Variablen in unterschiedlichen Blöcken (die nicht in einander geschachtelt sind) verwenden. So würde der Speicherplatz doch effektiv am Ende des Blockes recycled.
- Bei Variablen von `class`/`struct`-Typ werden Konstruktor und Destruktor entsprechend aufgerufen.

Lokale Variablen (2)

- Lokale Variablen werden auf dem Stack angelegt.

  Wie oben gilt das nur für normale lokale Variablen, nicht `static`.
- Es ist ein Fehler, die Adresse einer lokalen Variable nach Ablauf der Lebensdauer dieser Variable aufzubewahren oder zu verwenden.
- Dies kann zu schwierig zu findenen Fehlern und Programmabstürzen führen.
- Beispiel: Siehe nächste Folie.
  Es wird dabei angenommen, daß `main` nacheinander die beiden Funktionen `f()` und `g()` aufruft.

Lokale Variablen (3)

```
int* p;

void f()
{       int n = 1;
        p = &n; // Das sollte man nicht tun!
}               // Hier endet die Lebensdauer von n

void g()
{       int m;
        cout << *p << '\n'; // Druckt 1 (eventuell)
        m = 2;
        cout << *p << '\n'; // Druckt 2 (eventuell)
        *p = 3;
        cout << m  << '\n'; // Druckt 3 (eventuell)
}
```

Lokale Variablen (4)

- Wenn `p` die Adresse von `n` enthält, ist der Zugriff auf `*p` nach Ablauf der Lebensdauer von `n` ein Fehler.
- Zufällig wurde der für `n` nicht mehr benötigte Speicherplatz an die Variable `m` neu vergeben.

Besonders zufällig ist das nicht, da die Funktionen `f` und `g` gleich aufgebaut sind. Wenn sich `f` beendet, und anschließend `g` gestartet wird, werden normalerweise die gleichen Speicherplätze auf dem Stack benutzt. Es gibt aber viele mögliche Ausnahmen. Z.B. könnte sich der Compiler entscheiden, die Variable `m` in einem Register zu halten, nicht auf dem Stack. Dann würde die Zuweisung an `m` den Wert von `*p` nicht ändern. Außerdem hat MS VC++ 6.0 die Option `/GZ` (siehe Project→Settings→C/C++). Diese soll das Debugging unterstützen und initialisiert alle nicht explizit initialisierten lokalen Variablen mit dem Bitmuster `0xCC`. Dann würde zuerst `-858993460` ausgegeben.

Lokale Variablen (5)

- Große Arrays sollte man übrigens besser nicht auf dem Stack anlegen (d.h. nicht als lokale Variable).
- Bei manchen Compilern hat der Stack eine feste und recht kleine Größe.

Oft kann man die Größe wenigstens als Option des Compilers bzw. des Linkers setzen. Je nach Betriebssystem wird heute aber häufig ein Stacküberlauf erkannt und durch eine Vergrößerung des Speicherbereichs behandelt. Dann ist es weniger kritisch. Ansonsten wären auch sehr tiefe Rekursionen möglicherweise unsicher. Bei einfachen/alten Betriebssystemen könnte der Stack, der sich unkontrolliert weiter ausbreitet, andere Speicherbereiche "überbügeln" (zerstören).

Globale Variablen (1)

- Globale Variablen existieren für die gesamte Laufzeit des Programms. Das obige Problem (Verweise auf gelöschte Objekte) kann hier nicht auftreten.

Der Konstruktor wird vor Beginn von `main()` aufgerufen, der Destruktor nach Ende von `main()`. Eine explizite Freigabe des Speichers ist nicht nötig, da das Betriebssystem den gesamten vom Programm (genauer Prozess) belegten Speicher nach Ende des Programmlaufes einsammelt und wiederverwendet.

- Globale Variablen werden in einem eigenen Hauptspeicherbereich angelegt (nicht auf dem Stack).

Mindestens bei UNIX-Systemen wird unterschieden zwischen initialisierten globalen Variablen im DATA Segment, und uninitialisierten Variablen im BSS Segment.

Globale Variablen (2)

- Der Compiler kann globalen Variablen bereits feste Adressen zuordnen.

Die allerdings beim Linken möglicherweise noch modifiziert werden, und beim Laden des Programms in den Hauptspeicher eventuell noch einmal (bei älteren CPUs ohne virtuelle Adressierung). Aber wenn mit der Programmausführung begonnen wird, sind die Adressen bekannt und fest, und explizit in die Maschinenbefehle eingetragen.

Lokale Variablen werden dagegen über den Stackpointer angesprochen (ein spezielles Register in der CPU), ihre Adresse wird erst während der Programmausführung berechnet. Bei rekursiven Prozeduren kann es ja gleichzeitig mehrere Kopien der lokalen Variablen geben.

Klassenvariablen

- Statische Attribute in Klassen (Klassenvariablen)
  - sind im Prinzip globale Variablen,

    Sie existieren auch für die gesamte Ausführungsdauer des Programms, und haben auch eine vom Compiler/Linker zugeordnete feste Adresse.
  - deren Sichtbarkeit (meist) eingeschränkt ist,

    Sie sollten als `private:` deklariert werden, so daß sie nur von Funktionen der Klasse aus zugreifbar sind.
  - und die zum Namensraum der Klasse gehören.

    Verschiedene Klassen können Klassenvariablen mit gleichem Namen haben. Dies sind wirklich verschiedene Variablen.

Lokale statische Variablen

- Als "`static`" deklarierte lokale Variablen
  - sind auch im Prinzip globale Variablen,

    Ihr Speicherplatz wird auch erst am Ende des Programmlaufes freigegeben (was dann nicht mehr explizit gemacht werden muß). Sie haben auch eine feste Adresse. Selbst wenn die Prozedur rekursiv ist, gibt es nur ein Exemplar der Variablen.
  - aber sie sind nur in der Funktion zugreifbar,

    Zumindest über den Namen. Die Adresse könnte man ggf. nach außen weitergeben, und über Zeiger auf sie zugreifen.
  - und der Konstruktor wird erst bei der ersten Ausführung der Deklaration aufgerufen.

    Nicht direkt am Beginn des Programms.

Temporäre Variablen (1)

- Wenn eine Funktion ein Objekt "by value" zurückliefert, ist dieses Objekt im Ausdruck, der den Funktionsaufruf enthält, ein temporäres Objekt.
- Man kann ein temporäres Objekt auch durch expliziten Aufruf eines Konstruktors anlegen:

```
Person("Stefan", "Brass").print();
```

- Temporäre Objekte existieren dann bis zum Ende des vollständigen Ausdrucks, der den Funktionsaufruf enthält, also normalerweise bis zum ";".

Temporäre Variablen (2)

- Allgemein ist ein vollständiger Ausdruck ein Ausdruck, der selbst nicht Teil eines größeren Ausdrucks ist.

  Ein Ausdruck, der die Bedingung von `if` oder `while` darstellt, endet bei der entsprechenden ")".
- Man kann ein temporäres Objekt an eine konstante Referenz binden, dann existiert es bis zum Ende des Blockes, in der die Referenz deklariert ist.
- Entsprechendes gilt, wenn man es zur Initialisierung einer lokalen Variablen benutzt.

Dynamischer Speicher (1)

- Die Lebensdauer der obigen Typen von Variablen ist an die statische Struktur des Programmtextes gebunden: Ein Objekt existiert
  - ◇ innerhalb eines Ausdrucks (temporäres Objekt),
  - ◇ innerhalb einer Funktion / eines Blockes (lokale Variable), oder
  - ◇ für die ganze Ausführungszeit des Programms.
- Mit der dynamischen Speicherverwaltung kann man Objekte/Speicher explizit nach Bedarf anlegen und explizit wieder löschen.

Dynamischer Speicher (2)

- Man kann beliebig viele Variablen dynamisch anlegen (bis der Hauptspeicher verbraucht ist).

  Bzw. der virtuelle Speicher, den das Betriebssystem dem einzelnen Prozess zugesteht.
- Daher ist die dynamische Speicherverwaltung dann wichtig, wenn beim Schreiben des Programms nicht bekannt ist, wie viele Variablen man braucht, sondern es sich erst zur Laufzeit entscheidet.

  Abhängig von den Daten/Eingaben.

Dynamischer Speicher (3)

- Man könnte sich in solchen Fällen natürlich auch ein genügend großes Array deklarieren.
- Das hat man früher viel gemacht.
  Aber es hat Nachteile:
  - ◇ Häufig ist das Array viel zu groß für die Eingabe.

    Wenn es ein Array von Objekten ist, können die Aufrufe des Konstruktors viel Zeit kosten. Auch sonst ist je nach Betriebssystem die Belegung und Freigabe von Speicher am Programmstart/Ende nicht ganz kostenlos.
  - ◇ Manchmal ist das Array zu klein.

    Dann muß das Programm erst neu compiliert werden.

Dynamischer Speicher (4)

- Während man ein einzelnes Array meist so groß machen kann, daß man kaum an Grenzen stossen wird, ist es bei mehreren Arrays schwieriger: Diese konkurrieren um den vorhandenen Speicher.

  Beispiel: Beim Datenbank-Managementsystem (DBMS) Oracle gibt es (mindestens in Ver. 8.0) sehr viele Optionen, um die Größe von einzelnen Tabellen im System festzulegen.
  Ein DBMS wird einen möglichst großen Teil des Hauptspeichers für die Pufferung (Caching) von Plattenblöcken verwenden. Macht man die Tabellen für Sperren, Sitzungen, Transaktionen, etc. sehr groß, so fehlt dieser Teil für den Puffer.
  Immerhin sind die Tabellen bei Oracle semi-dynamisch: Man muß das Programm nicht neu compilieren (man hat die Quellen ja auch nicht), wenn man die Array-Größe ändert.

Dynamischer Speicher (5)

- Mit dynamischer Speicherverwaltung kann man dagegen genau so viel Speicher anfordern, wie man für die konkrete Eingabe benötigt.
- Man kann den Speicher wieder freigeben, sobald man ihn nicht mehr benötigt.
- Falls man später wieder Speicher braucht (z.B. für Objekte einer anderen Klasse), kann der freigegebene Speicher wiederverwendet werden.

  Wenn man die Arrays dagegen nicht in einer Prozedur deklariert (was nicht anzuraten ist, s.o.), würden sie für die ganze Laufzeit des Programms Speicher belegen.

Dynamischer Speicher (6)

- Normalerweise hat eine Variablendeklaration zwei Funktionen:
  - ◇ Es wird Speicher reserviert.
  - ◇ Es wird ein Name eingeführt, mit dem man den Speicher ansprechen kann.
- Bei Referenzen hat man (im wesentlichen) nur den zweiten Punkt.
- Dynamische Speicheranforderung hat dagegen nur die erste Wirkung.

Dynamischer Speicher (7)

- Da eine dynamisch angelegte Variable keinen Namen hat, kann man sie nur über ihre Adresse ansprechen (mit einem Zeiger).
- Nun hat man aber auch nur eine feste Anzahl benannter Zeigervariablen.
- Dynamische Speicherverwaltung ist daher oft mit untereinander verzeigerten Datenstrukturen (z.B. Listen) verbunden: Der Zeiger ist dann selbst in einer dynamisch angeforderten Variable enthalten.

  Man kann aber auch Arrays mit einer erst zur Laufzeit bekannten Größe anfordern. In diesem Fall reicht eine benannte Zeigervariable.

Dynamischer Speicher (8)

- Der Speicherbereich, in dem dynamisch angeforderte Variablen angelegt werden, heißt der "Heap".

  Zu Deutsch "Haufen". Stroustrup nennt ihn "Free Store". Viele Leute sprechen auch von "dynamic memory".
- Dieser Speicher wird vom Betriebssystem nach Bedarf angefordert, während das Programm läuft.

  Der andere Speicher wird dagegen belegt, wenn das Programm gestartet wird (mit Ausnahme eventueller Stack-Verlängerungen, s.o.). Selbstverständlich wird der Compiler/die Bibliothek versuchen, Speicher in größeren Stücken vom Betriebssystem anzufordern, und nicht für jede dynamisch angelegte Variable einzeln. Ein Betriebssystem-Aufruf kostet relativ viel Laufzeit.

Anfordern von Speicher (1)

- Man reserviert dynamisch Speicher mit dem Operator `new` und der Angabe des Datentyps:

```
int *p = new int;
```

- Danach zeigt `p` auf `int`-Variable im dynamischen Speicherbereich (Heap).
- Die Variable ist nicht initialisiert.
- Man kann der neuen Variable aber natürlich über `p` einen Wert zuweisen: `*p = 5;`
- Es geht auch die Konstruktor-Syntax:

```
int *p = new int(5);
```


Anfordern von Speicher (2)

- Wird eine Variable von einem Klassen-/Struktur-Typ angelegt, müssen Werte für die Parameter eines Konstruktors angegeben werden.

  Sofern es keinen Default-Konstruktor gibt.

- Beispiel:

```
Person* pers = new Person("Stefan", "Brass");
```

- Die Variable `pers` zeigt jetzt auf ein neues Objekt im Heap, dieses ist natürlich initialisiert (der Konstruktor wurde aufgerufen). Nun z.B. möglich:

```
cout << pers->get_nachname();
```


Anfordern von Speicher (3)

- Man kann auch Arrays von einfachen Datentypen oder Objekten anfordern.
- Ein Array mit `n` Integers bekommt man durch:

```
int* a = new int[n];
```

- Nun kann man z.B.

```
a[3] = 27;
```

  schreiben (falls `n` mindestens 4 war).
- Offiziell wird hier der Operator `new[]` benutzt.

Erschöpfter Speicher (1)

- Es ist immer möglich, daß der Speicher “alle” ist, die Anforderung also nicht erfüllt werden kann.

  Z.B. könnten andere große Prozesse gerade parallel laufen. Bei Windows kann der Speicherbereich auf der Platte, der normalerweise für “virtuellen Hauptspeicher” benutzt wird, nicht mehr erweitert werden, wenn die Platte voll ist.

- Normalerweise würde dann die “Exception” (Laufzeitfehler, Ausnahme) “`bad_alloc`” ausgelöst werden, die das Programm beendet, wenn man sie nicht abfängt.

Erschöpfter Speicher (2)

- Exceptions werden in Kapitel 13 näher betrachtet.
- Man kann diese Exception so abfangen:

```
try {
      ... // Beliebiger Programmcode
          // (ruft direkt oder indirekt new auf)
}
catch(bad_alloc) {
      cerr << "\nSpeicher alle!\n";
}
```

Falls in dem durch `try` überwachten Bereich der dynamische Speicher ausgeht, springt die Programmausführung zur Ausgabe der Fehlermeldung auf `cerr` (Standard-Kanal für Fehlermeldungen). Falls man das Problem nicht wirklich behandeln kann, kann man dann aber doch nur das Programm beenden (ggf. vorher Daten sichern).

Erschöpfter Speicher (3)

- Man kann auch eine Prozedur deklarieren, die aufgerufen wird, wenn es zu diesem Fehler kommt:

```
void speicher_alle()
{
        cerr << "Leider Speicher alle ...\n";
        throw bad_alloc;
}
```

- Man trägt die Adresse dieser Funktion als "new handler" ein mittels

```
set_new_handler(speicher_alle);
```

Funktioniert in MS VC++ 6.0 nicht. Geboten wird ähnliche Funktion `_set_new_handler`. Siehe Handbuch.

Speicher freigeben (1)

- Einen mit `new` angeforderten Speicherbereich kann man mit `delete` wieder freigeben:

```
delete p;
delete pers;
delete[] a;
```

- Bei Arrays muß man `delete[]` verwenden.

Bei Arrays muß `new` in den Speicherbereich eintragen, wie groß das Array ist, um die gleiche Speichergröße wieder freizugeben (es wird sich diese Größe typischerweise direkt unterhalb der gelieferten Adresse merken). Bei normalen Variablen gibt der Typ des Zeigers dagegen die Information, wie groß der Speicherbereich ist (natürlich nur, falls keine Typ-Umwandlungen vorgenommen wurden, siehe virtuelle Destruktoren in Kapitel 10).

Speicher freigeben (2)

- Natürlich darf man `delete` absolut nur auf Adressen anwenden, die man vorher von `new` bekommen hat.

Und entsprechend `delete[]` auf Adressen, die man von der Array-Variante von `new` bekommen hat.

- Außerdem darf man jede solche Adresse nur einmal freigeben.
- Und man darf nach dem `delete` nicht mehr auf die Adressen zugreifen.

Die Lebensdauer der Variable ist beendet, der Speicherbereich ist möglicherweise für eine andere dynamische Variable neu genutzt.

Speicherlecks

- Wenn das Programm sich beendet, wird aller Speicher ohnehin freigegeben, man muß also nicht unbedingt `delete` für alle mit `new` angeforderten Variablen aufrufen.

  Destruktoren werden dann nicht aufgerufen. Wahrscheinlich ist es besserer Stil, Speicher explizit freizugeben. Dann kann man das Programm später eher in eine Schleife einbetten und mehrfach ausführen.

- Wenn ein Programm aber lange läuft (z.B. Server-Prozess), und Speicher anfordert, aber nicht freigibt, wird der Speicher irgendwann alle sein.
- Man spricht dann von einem Speicher-Leck.

Garbage Collection

- In anderen Sprachen (wie z.B. Java) gibt es einen "Garbage Collector", der nicht mehr erreichbare Speicherbereiche im Heap einsammelt und dem Recycling zuführt.

  Deutsch: "Müll-Einsammler". Er geht dazu von den benannten Variablen im Programm aus und folgt allen Zeigern. Was auf diese Art nicht erreichbar ist, kann vom Programm nicht mehr verwendet werden.

- Auf die Art braucht man kein explizites `delete`.

  Und vermeidet die damit verbundenen Fehlermöglichkeiten. Man muß dann Pointer auf nicht mehr benötigte Datenstrukturen mit dem Nullzeiger überschreiben.

- Aber es ist natürlich nicht so effizient.

Verkettete Listen (1)

- Beispiel: Man soll ein Programm schreiben, das eine unbekannte Anzahl von Zahlen einliest, und sie dann in umgekehrter Reihenfolge wieder ausgibt.

  Dies wäre tatsächlich ein Beispiel für eine Stack-Datenstruktur. Ein Stack arbeitet nach dem LIFO-Prinzip: "Last in, first out". Eine Warteschlange ("queue") arbeitet dagegen nach dem FIFO-Prinzip: "First in, first out". Mit verketteten Listen kann man beides implementieren, je nachdem, ob man vorne oder hinten an die Liste anfügt.

- Die Eingaben sollen mit der Zahl 0 beendet werden.
- Es soll eine verkettete Liste zur Implementierung verwendet werden. Es reicht, wenn das Programm eine einzige verkettete Liste enthält.

Verkettete Listen (2)

```
class list_node_c {
private:
        int Val;
        list_node_c* Next;
        static list_node_c* First;
        list_node_c(int val, list_node_c* next);
public:
        int get_val() const { return Val; }
        list_node_c* get_next() const {return Next;}
        static list_node_c* get_first(int val)
                { return First; }
        static void insert_front(int val);
};
```

Verkettete Listen (3)

```
list_node_c* list_node_c::First = 0;

list_node_c::list_node_c(int val, list_node_c* next)
{
        Val = val;
        Next = next;
}

void list_node_c::insert_front(int val)
{
        list_node_c* p = new list_node_c(val, First);
        First = p;
}

typedef list_node_c* list_node_t;
```


Verkettete Listen (4)

```
int main()
{
        for(int i; ; ) {
                cout << "Bitte Zahl eingeben: ";
                cin >> i;
                if(i == 0) break;
                list_node_c::insert_front(i);
        }
        cout << "\nUmgekehrte Reihung:\n";
        for(list_node_t p = list_node_c::get_first();
                        p; p = p->get_next())
                cout << p->get_val() << '\n';
        return 0;
}
```