Vorkurs C++ Programmierung

Speicherverwaltung

UNIVERSITÄT SIEGEN

Letzte Stunde

- Call by Value vs. Call by Reference
- Referenzen und Pointer

→Wenn möglich, Referenzen oder Zeiger als Parameter an Funktionen übergeben

→Call-by-Reference.

Heute:

Speicherverwaltung

Rückblick: Prozesse

Was sind Prozesse?

1. Definition

Ein Prozess ist ein Programm in Ausführung

2. Definition

Ein Prozess ist eine sich ändernde Aktivitätseinheit, gekennzeichnet durch:

- eine Ausführungsumgebung mit
  - Adressbereich (Programmcode und Daten)
  - Zustandsinformationen benutzter Ressourcen, z.B. offene Dateien
- einem oder mehreren Ausführungswegen (Threads) mit
  - dem Befehlszähler
  - den Registern des Prozessors
  - dem Zeiger auf den Kellerspeicher (Stack)

Rückblick: Prozesse

UNIVERSITÄT SIEGEN Einführung in Linux ▷ Prozesse und Prozessverwaltung ▷ Was sind Prozesse? 57 / 70

Was sind Prozesse?

1. Definition

Ein Prozess ist ein Programm in Ausführung

2. Definition

Ein Prozess ist eine sich ändernde Aktivitätseinheit, gekennzeichnet durch:

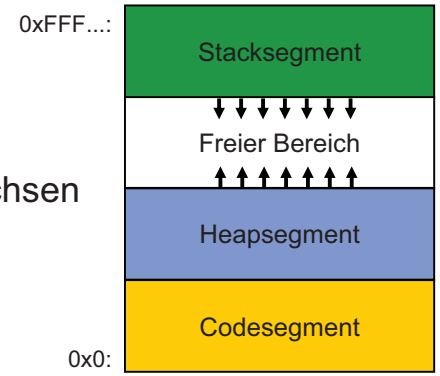
- eine Ausführungsumgebung mit
  - Adressbereich (Programmcode und Daten)
  - Zustandsinformationen benutzter Ressourcen, z.B. offene Dateien
- einem oder mehreren Ausführungswegen (Threads) mit
  - dem Befehlszähler
  - den Registern des Prozessors
  - dem Zeiger auf den Kellerspeicher (Stack)

Speicherverwaltung

- „Programm“ und „Prozess“
  - Prozess: „Ausführungsumgebung“ eines Programms
  - Ein Programm kann in mehreren Prozessen laufen
- Jeder Prozess hat seinen eigenen Speicherbereich
  - „logischer“ bzw. „virtueller“ Adressraum genannt
  - abstraktes Konzept
- „Speicher“ in Hardware
  - „physikalischer“ oder „physischer“ Adressraum
- Analogie: Logisches vs. physikalisches Laufwerk
- Zusammenspiel der beiden ist kompliziert
  → Vorlesungen Betriebssysteme I und Rechnerarchitektur I!

Speicherverwaltung

- Hier wichtig: Der logische/virtuelle Adressraum
  - Die Menge der verwendbaren Speicheradressen ***aus der Sicht eines Prozesses***
  - Ein Prozess sieht nur seinen eigenen Speicherbereich!
  - besteht aus
    - einem Codebereich
    - dem Stack
    - dem Heap
  - Stack und Heap wachsen aufeinander zu

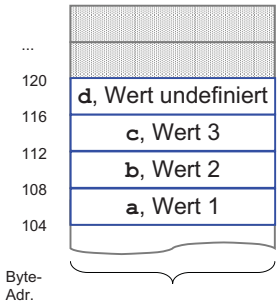


Speicherverwaltung: Stack

- Bisherige Speichernutzung
  - lokale Variablen, falls innerhalb von {}-Blöcken
  - globale Variablen, falls außerhalb von {}-Blöcken
- Diese Variablen werden als **`automatic`**-Variablen bezeichnet (kurz **`auto`**)
  - Speicherung (lokaler) **`auto`**-Variablen erfolgt im **Stack**
  - maximale Größe des Stack ist festgelegt (Standardeinstellung 1 MB)
  - „**`automatic`**“: Speicher reservieren und freigeben geschieht automatisch
- Bisher haben wir nur **`automatic`** Variablen benutzt.

Speicherverwaltung

- Prinzip des Stack:




```
void function()
{
        int a = 1;
        int b = 2;
        int c = a + b;
        int d;
}

int main(int argc, char* argv[])
{
        function();
        return 0;
}
```

An dieser Stelle ist function() durchlaufen. Die Variablen werden vom Stack gelöscht und sind nicht mehr zugreifbar.

Speicherverwaltung: Stack

- Vorteil:
  - Wird automatisch verwaltet. Was nicht mehr gebraucht wird, wird gelöscht.
- Nachteil:
  - Begrenzte Größe
  - Größe des benötigten Speichers muss schon beim Kompilieren bekannt sein
  - Stackvariablen werden gelöscht, wenn das Programm den Block verlässt, in dem sie gültig sind

  → keine Benutzung an anderer Stelle mehr möglich.

Was aber, wenn man...

...die Variablen an anderer Stelle noch braucht?

...erst zur Laufzeit weiß, wie viel Speicher man braucht?

Speicherverwaltung: Nachteile Stack

- Nachteil: Stackvariablen werden gelöscht, wenn das Programm den Block verlässt, in dem sie gültig sind
- *Vermeintlich* nahe liegende Lösung: Alles global deklarieren?
  - Bloß nicht!
  - Benutzung globaler Variablen ist keine Lösung!
    - Auch die wären auf dem Stack!
    - Der Platz auf dem Stack ist begrenzt (es gibt nicht nur **`int`**, **`char`**...)
    - Globale Variablen können überall geändert werden.
      - Unerwünschtes, unvorhersehbares Verhalten möglich...

Speicherverwaltung: Nachteile Stack

- Nachteil: Größe des benötigten Speichers muss schon beim kompilieren bekannt sein
  - Der Compiler muss *vor der Ausführung* wissen, wie viel Speicher er auf dem Stack reservieren muss.
    - In „anspruchsvolleren“ Programmen entscheidet sich der Speicherbedarf oft erst zur Laufzeit
    - Programmablauf (und damit auch der Ressourcenbedarf) kann z.B. abhängig sein von Benutzereingaben:

So geht es nicht:

```
int length;
cout << "Bitte Größe des Arrays angeben: ";
cin >> length;
int array[length];
```

Compiler meldet hier einen Fehler, weil erst zur Laufzeit (nach der Benutzereingabe) feststeht, wie viel Speicher **auf dem Stack** gebraucht wird. (g++ meldet keinen Fehler...)

Speicherverwaltung: Heap

- Alternative: „Dynamische Speicherverwaltung“
  - Dynamischer Speicher wird
    - bereitgestellt (alloziert), sobald er im Programmablauf benötigt wird
    - freigegeben (dealloziert), sobald er nicht mehr benötigt wird
  - „Heap“ nennt man den Speicherbereich, der zur dynamischen Speicherung verwendet werden kann
  - Allokation und Deallokation von Speicher auf dem Heap muss selbst vorgenommen werden!
    - sorgfältige Verwaltung dynamischen Speichers ist sehr wichtig

    „memory leak“: Speicher, der alloziert, aber nicht wieder freigegeben wird → Heap läuft voll → Programm „stürzt ab“

Dynamischer Speicher (Heap)

- Speicher wird alloziert/dealloziert mit dem Operatorenpaar **`new`** und **`delete`**

  `new` liefert einen Pointer auf eine Speicherstelle auf dem Heap zurück
- **Rückblick:**

```
int length;
cout << "Bitte Größe des Arrays angeben: ";
cin >> length;
int array[length];
```

Compiler meldet hier einen Fehler, weil erst zur Laufzeit (nach der Benutzereingabe) feststeht, wie viel Speicher **auf dem Stack** gebraucht wird. (g++ meldet keinen Fehler...)

- **Jetzt: Mit `new` auf dem Heap**

```
int length;
cout << "Bitte Größe angeben: ";
cin >> length;
int *array = new int [length];
```

Compiler meldet hier keinen Fehler, weil der Speicher zur Laufzeit auf dem Heap reserviert werden soll.

Dynamischer Speicher (Heap)

- **Jetzt:** Mit **`new`** auf dem Heap

```
int length;
cout << "Bitte Größe angeben: ";
cin >> length;
int *array = new int [length];
```

Compiler meldet hier keinen Fehler, weil der Speicher zur Laufzeit auf dem Heap reserviert werden soll.

- Definition eines Pointers **`array`**, der auf ein int-Array der Länge **`length`** zeigt, welches auf dem Heap angelegt wird
- Der *Pointer selbst* liegt aber nach wie vor auf dem Stack
  - Für ihn gelten die gleichen Gültigkeitsgrenzen wie für alle lokalen Variablen

Dynamischer Speicher: **`new`** und **`delete`**

- Speicher wird alloziert/dealloziert mit dem Operatorenpaar **`new`** und **`delete`**
  - **`new`** liefert einen Pointer auf eine Speicheradresse auf dem Heap zurück. Syntax:

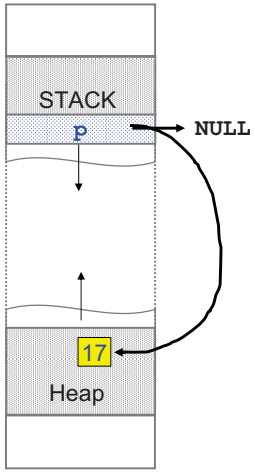
    entweder: `type *pName = new type; //Dekl. und All. in einem`

    oder `type *pName = NULL; //Deklaration`

    `pName = new type; //Allokation`
  - **`NULL`**: Zeiger sollten auf **`NULL`** zeigen, wenn sie noch nicht auf etwas anderes zeigen können (Vermeidung von Speicherzugriffsfehlern)
- Alles, was mit **`new`** alloziert wurde, muss mit **`delete`** gelöscht werden. Syntax:

```
delete pName;     //Deallokation
pName = NULL;     //Auf NULL zeigen lassen
```


Stack und Heap




```
bool isPrimzahl(int *p) {...}

void function()
{
        int *p = new int;
            *p = 17;
        bool b = isPrimzahl(p);
        //Hier brauchen wir den Pointer
        //nicht mehr
        delete p;  //Freigabe des Speichers
        p = NULL;  //Zeiger auf NULL setzen
}
```

Der Speicher **muss** hier freigegeben werden!

Dynamischer Speicher: **new** und **delete**

- Speicher wird alloziert/dealloziert mit dem Operatorenpaar **new** und **delete**
  - Sonderfall Arrays: Zusätzlich ein Klammernpaar `[]`, wenn Speicherplatz für Arrays alloziert/dealloziert wird:

```
float *fArray = new float[10]; //alloziert 10 floats
...
delete[] fArray; //Freigabe der 10 floats
fArray = NULL;   //Zeiger auf NULL setzen
```


Dynamischer Speicher

(Schlechtes!) Beispiel

```
{
   //reserviere Platz für 20 ints
   int *pIntArr = new int [20];
   for (int i = 0; i < 20; i++)
   {
      //Belegung mit Werten
      pIntArr[i] = i;
   }
   //tu irgendwas mit den Daten
}
//Hier ist der Zugriff auf den
//Pointer verloren, der
//Speicherplatz wurde aber nie
//freigegeben!
//→ Memory leak
```

Korrektes Löschen:

```
{
   //reserviere Platz für 20 ints
   int *pIntArr = new int [20];
   for (int i = 0; i < 20; i++)
   {
      //Belegung mit Werten
      pIntArr[i] = i;
   }
   //tu irgendwas mit den Daten
   //wenn fertig, gib den Speicher
   //frei
   delete [] pIntArr;
   pIntArr = NULL;
}
```


Dynamischer Speicher

- Pointer auf Speicherstellen auf dem Heap dürfen nicht verloren gehen
  - Pointer selbst steht auf dem Stack, d.h. der Pointer ist nichts anderes als eine lokale Variable
    - → geht außerhalb ihres Blocks verloren
    - → keine Möglichkeit mehr, den Speicher auf dem Heap freizugeben!
    - → memory leak!

Bisher unbeantwortet: Was aber, wenn man die Variablen an anderer Stelle noch braucht?

- „Weiterreichen“ von Pointern auf Heapvariablen über Rückgabedaten von Funktionen
  - → Zugriff über Funktionsgrenzen hinweg
  - → Vorsicht: Der Speicher muss irgendwo wieder freigegeben werden!

    Hier kann man leicht den Überblick verlieren.

Speicherverwaltung

- Die Vorteile dynamischer Speicherverwaltung werden deutlicher, wenn wir nicht mehr nur einfache Datentypen haben, sondern komplexere, die mehr Speicherplatz brauchen.
- Komplexere Datentypen werden durch **Klassen** definiert!

Speicherarten

- Schlüsselwörter, um über die Art und den Ort der Speicherung eines Objekts zu entscheiden:
  `auto`, `static`, `const`, `register`, `extern`, `mutable`, `volatile`
- Schon bekannt: **`auto`**-Variablen: Automatisches anlegen und löschen auf dem Stack
- **`static:`**

  static-Variablen sind wie lokale Variablen nur innerhalb ihres Blockes sichtbar, ihre Lebensdauer ist aber erhöht, d.h. sie werden nicht gelöscht, wenn der Block verlassen wird!

  Beispiel: Eine Zählfunktion in einem Programm zur Verwaltung von Kunden. Wird ein neuer Kunde in die Liste aufgenommen, wird die Funktion aufgerufen, die die Anzahl der Kunden erhöht

Speicherarten - static

Beispiel: Eine Zählfunktion in einem Programm zur Verwaltung von Kunden. Wird ein neuer Kunde in die Liste aufgenommen, wird die Funktion aufgerufen, die die Anzahl der Kunden erhöht

```
unsigned int addCustomer()
{
   //wird nur einmal initialisiert:
   static unsigned int numCustomers = 0;
   //wird bei jedem Funktionsaufruf inkrementiert
   numCustomers++;
   return numCustomers;
}
//numCustomers behält seine Gültigkeit über die Blockgrenze
//hinaus (ist aber nur innerhalb des Blocks sichtbar).
```


Speicherarten

- **`static`** kann noch mehr (s. Schneeweiß Kap. 2.2.24)
  - U.a. erst im Kontext von Klassen
- **`const`** ist ebenso vielfältig, deshalb auch hier nur eine Auswahl...
  - Definieren von Konstanten, d.h. Attributen, deren Wert einmal festgelegt wird und sich nicht ändern darf.
  - **`const`** verbietet dabei den Schreibzugriff auf den betreffenden Speicherbereich

    `const unsigned long obergrenze = 1000;` oder

    `unsigned long const obergrenze = 1000;`
- Der Wert von **`obergrenze`** kann zur Laufzeit nicht mehr geändert werden

Anmerkung zu Konstanten

- Konstanten werden oft mit der Präprozessordirektive **`#define`** definiert, die eine reine Textersetzung durchführt:

```
#include <iostream>
#define OBERGRENZE 1000

int main(int argc, char* argv[])
{
   for(int i = 0; i <= OBERGRENZE; i++)
   {
       //tu etwas
   }
   return 0;
}
```

Konvention: GROSSBUCHSTABEN!

Konstanten mit const

- Das gleiche Beispiel mit const:

```
#include <iostream>
const unsigned int obergrenze = 1000;
int main(int argc, char* argv[])
{
   for(int i = 0; i <= obergrenze; i++)
   {
       //tu etwas
   }
   return 0;
}
```


Konstanten

- Für beide Varianten gibt es gute Gründe
- **`#define`**-Variante:
  - Präprozessor-Kommando: Einfache Textersetzung
  - Verbraucht keinen Speicher
  - Einfache Änderung
  - Negativ: Meistens keine Typprüfung (außer mit Literalkonstanten, s. Schneeweiß Kap. 2.2.5)
    `#define OBERGRENZE 1000UL`
  - Mit **`#define`** erstellte Konstanten sind nicht referenzierbar
- **`const`**-Variante:
  - Deklaration von typisierten Konstanten
  - Fehler bei der Syntaxüberprüfung fallen auf
  - Compiler kann bei Bedarf eine Typkonvertierung machen
  - **`const`**-Konstanten sind per Zeiger referenzierbar

Zusammenfassung

- Adressraum von Prozessen
- **`auto`**-Variablen auf dem Stack
  - Prinzip des Stack, Vorteile, Nachteile
  - Compiler, Speicherreservierung auf dem Stack
- dynamische Speicherverwaltung auf dem Heap
  - Allokation: **`new, new[]`**
  - Deallokation: **`delete, delete[]`**
  - memory leaks
- **`new/new[]`** liefert Pointer auf eine Speicherstelle
  - Pointer dürfen nicht verloren gehen → Speicherfreigabe nicht vergessen

    Immer: **`delete pointer;`**
    `pointer = NULL;`

Zusammenfassung

- Speicherklassen: **`auto, static, const`**
  - **`static`** und **`const`** können noch viel mehr
- Konstanten
  - **`const`** und **`#define`**

- Nächste Stunde: Klassen!