Verfasser:
D. Gachet / HTA-FR - Telekommunikation

HTA-FR – Kurs Telekommunikation

Embedded systems 1 und 2

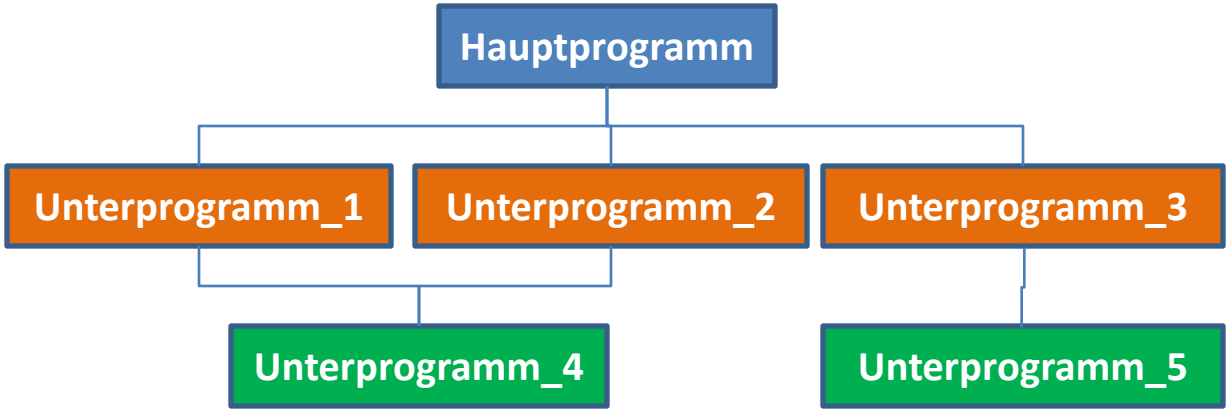
Programmiersprache C – Die Funktionen

Klasse T-2 // 2018-2019

Inhalt

- Einführung
- Deklaration und Definition
- Argumente
- Variablen
- Aufrufe und Rekursivität
- Wiedereintritt

- **Ein gutes Programm verwendet das Modularitätsprinzip, das daraus besteht, das Programm in mehrere Unterprogramme aufzuteilen.**
  - Das Hauptprogramm enthält nur die logische Struktur des Algorithmus
  - Die Unterprogramme befassen sich nur mit den Details der Implementierug



- **Vorteil der Verwendung von Unterprogrammen**
  - Kürzeres Hauptprogramm
  - Einfachere Entwicklung grosser Programme (Bibliotheken)
  - Einfachere Überprüfung grosser Programme (einheitliche Tests)

Deklaration

C unterstützt wie viele andere Sprachen das Konzept der Funktion, das die Aufteilung langer und komplexer Verarbeitungsalgorithmen in kleine einfache und klare Prozeduren erlaubt.

Die Definition einer Funktion hat die folgende Form:

```
[extern] <type_name> <function_name> (argument-list);
```

Die Deklaration einer Funktion erlaubt, ihre Signatur/ihren Prototyp zu kennen (Typ und Liste der Argumente/Parameter) ohne ihre Implementierung kennen zu müssen (forward declaration). Wenn die Funktion keinen Wert zurück gibt, muss der Typ `void` verwendet werden.

Die Funktionen lassen sich natürlich in Bibliotheken zusammenfassen. Diese Bibliotheken erlauben den Softwareentwicklern, neue Anwendungen zu schreiben, indem sie auf Bibliotheken zurückgreifen, die andere bereits realisiert haben. Dadurch wird vermieden, jedes Mal wieder bei null beginnen zu müssen. Die Funktionsdeklaration erfolgt daher in den Kopfdateien (header-files) und der Deklaration der Funktion muss das Präfix `extern` vorangestellt werden, das anzeigt, dass die Funktion nicht Teil des Kompilierungsmoduls ist, sondern aus einem andern Modul oder einer Bibliothek stammt.

Definition

Die Definition (Implementierung) einer Funktion hat die folgende Form:

```
[static] <type_name> <function_name> (argument-list) {
   statements
   [return [expression];]
}
```

Der Rückgabeausdruck `return` muss vom gleichen Typ sein wie die Funktion (`<type_name>`).

Das Präfix `static` erlaubt, eine Funktion zu privatisieren, d. h. sie nur im Innern des Kompilierungsmoduls sichtbar zu machen.

z. B.

```
int square (int n) {
   return n*n;
}

static void do_nothing () {
}
```

Achtung:

- **Es ist nicht möglich, Funktionsdefinitionen zu verschachteln.**
- **Der Funktionsname muss innerhalb eines Kompilierungsmoduls oder, wenn die Funktion global ist, innerhalb einer Anwendung eindeutig sein.**
- **Es ist daher sinnvoll, den Befehl "*return*" nur am Ende der Funktion zu verwenden.**

Argumente

- **Bei der Parameterübergabe eines Programms kommen zwei Techniken zum Einsatz:**
  - Wertübergabe
  - Referenzübergabe (Adresse)

- **Die Technik der Wertübergabe (call by value) legt eine Kopie des aktuellen Datenelements auf dem Stapel (stack) ab:**
  - Das Unterprogramm erhält eine Kopie des Datenelements
  - Es kann es lesen und ändern, ohne dass das ursprüngliche Datenelement verändert wird
  - Es sind zwei getrennte Speicherstellen vorhanden
  - Die Änderung einer Stelle hat keine Auswirkungen auf die andere

  Beispiel: `int fnct1 (int a, char c);`

- **Die Technik der Referenzübergabe (call by referenz) legt die Adresse des Datenelements auf dem Stapel (stack) ab:**
  - Das Unterprogramm erhält die Adresse des Datenelements
  - Dieses Datenelement wird auf diese Weise von der aufrufenden Funktion und der aufgerufenen Funktion gemeinsam benutzt
  - Das Datenelement befindet sich nur an einer Speicherstelle
  - Die Parameteränderung durch die aufgerufene Funktion ist auch durch die aufrufende Funktion sichtbar

  Beispiel: `int fnct2 (char* s, int* b);`

Argumente – durch Referenz / konstante Adresse

Um zu vermeiden, dass die durch die Adresse übergebenen Parameter durch die Funkion verändert werden, müssen sie als konstante Werte `const` übergeben werden.

z. B.

1. `int fnct3 (const char* s, int* b);`
   der Inhalt von 's' kann nicht geändert werden

2. `int fnct4 (const char* const s, int* b);`
   weder der Inhalt noch die Adresse können geändert werden

Achtung:

Es ist wichtig zu wissen, dass alle Parameter einer Funktion auf den Stapel (stack) kopiert werden. Wenn die Argumente durch Werte übergeben werden und gross sind, verliert das Programm enorm an Leistungsfähigkeit und es kann ein Stapelüberlauf ("*stack overflow*") auftreten. Um solche Unannehmlichkeiten zu vermeiden, wird empfohlen, solche Parameter durch Referenz (mit konstanter Adresse) zu übergeben.

Argumente – Ellipsenoperator

Wenn die Anzahl Argumente bei der Deklaration einer Funktion nicht definiert werden kann, erlaubt C, die Signatur der Funktion mithilfe des Operators zu definieren

"..."
(ellipsis operator).

z. B.

```
extern void printargs (int argc, ...);
```

Die Bibliothek `<stdarg.h>` erlaubt dann, auf der Liste der Argumente zu iterieren, z. B.

```
void printargs (int argc, ...)
{
  va_list ap;
  va_start(ap, argc);

  while (argc-- > 0) {
    int i = va_arg(ap, int);
    printf("%d ", i);
  }
  va_end(ap);

  printf ("\n");
}
```

→ Zeiger auf die nicht deklarierten Argumente

→ macht, dass 'ap' auf das 1. nicht deklarierte Argument zeigt

→ wandelt das Argument in den gewünschten Typ um

→ stellt den Zustand nach der Verwendung wieder her (leert die Liste)

Argumente – ARM-Konventionen

- **C-Konventionen für die ARM-Prozessoren**
  - Die 4 ersten Parameter werden in den Registern R0 bis R3 abgelegt, die andern Parameter werden von rechts nach links auf dem Stapel (stack) abgelegt, d. h. der am weitesten rechts stehende Parameter wird als Erster und der am weitesten links stehende Parameter als Letzter auf dem Stapel abgelegt.
  - In einem 32-Bit-Prozessor werden Zeichen, Konstanten und Aufzählypen vor dem Ablegen auf dem Stapel (stack) auf 32 Bit erweitert.
  - Die Verwendung der Register ist in *04_ARM_Architecture_Procedure_Call_Standard.pdf* wie folgt definiert:

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Lokale Variablen

Die Funktionen können lokale Variablen definieren; das sind Variablen, die nur innerhalb der Funktion oder eines Funktionsblocks sichtbar sind.

Diese Variablen werden initialisiert, sobald die Programmausführung ihre Definition erreicht Diese Initialisierung erfolgt bei jedem Funktionsaufruf.

Die Variablen werden bei jedem Funktionsaufruf (eine Kopie pro Aufruf) erzeugt, z. B.

```
int fnct (int a) {
   int b=8;                    // initialized at each call
   return a * b;
}
```

Achtung:

Die lokalen Variablen werden auf dem Stapel ("*stack*") erzeugt und können einen Stapelüberlauf ("*stack overflow*") verursachen.

Remanente Variablen

Wenn eine Funktion den Wert einer Variablen zwischen zwei Aufrufen beibehalten muss, kann dies durch die Deklaration der Variablen als `static` erreicht werden. Diese wird dann nur einmal beim Start der Anwendung initialisiert.

z. B.

```
int fnct (int a) {
  int b=8;                  // initialized at each call
  static int count = 0;     // initialized once (0, 1, 2 … n calls)
  count++;
  return a * b + count;
}
```

Achtung:

Die Verwendung von statischen Variablen führt dazu, dass der Code nicht "re-entrant" bleibt und kann in Anwendungen, die Threads verwenden, zu grossen Schwierigkeiten führen.

Globale Variablen

Funktionen können auch auf globale Variablen zugreifen; das sind Variablen, die ausserhalb der Funktion definiert sind. Diese Variablen werden natürlich durch alle Funktionen eines Moduls/Programms gemeinsam genutzt

z. B.

```
int g_var = 10;  //initialized once, while launching the program


int fnct (int a) {
   int b=8;                  // initialized at each call
   static int count = 0;     // initialized once (0, 1, 2 … n calls)
   count++;
   return (a * b + count) * g_var;
}
```

Der Zugriff auf diese globale Variablen durch Funktionen ist unter dem Begriff Seiteneffekt (side effect) bekannt. Seine Verwendung wird nicht empfohlen. Es ist vorzuziehen, bei ihrem Aufruf, ihre Referenz/Adresse der Funktion zu übergeben.

Achtung:

Die Verwendung von globalen Variablen führt dazu, dass der Code nicht "re-entrant" bleibt, und kann in Anwendungen, die Threads aufrufen, zu grossen Schwierigkeiten führen.

Der Aufruf einer Funktion kann auf zwei verschieden Arten erfolgen, entweder als Befehl oder als Operand eines Ausdrucks.

z. B.

1. Befehl `fnct (10);`
2. Operand `int i = fnct (20);`

Der Funktionsaufruf erfolgt in 4 Etappen:

1. Die Aufrufparameter werden ausgewertet
2. Für jeden Parameter wird das Ergebnis seiner Auswertung in einem Register oder auf dem Stapel (stack) abgelegt
3. Aufruf der Funktion
4. Wiederherstellung des Stapels (stack)

Achtung:

Die Reihenfolge der Parameterauswertung ist nicht definiert und kann zwischen den Compilern oder ihren Versionen zu unterschiedlichem Verhalten führen.

```
extern int fnct (int a, int b, int c);
int i=0; int j=3; int k=0;
k = fnct (i, ++i, j);          → a kann sehr gut auch 0 anstatt 1
                                 enthalten!!!
k = fnct (i, i+1, j); i++;     → a das gute Verhalten.
```

C implementiert das Konzept der Rekursivität, d. h. den Aufruf einer Funktion durch sich selbst, was den Softwareentwicklern erlaubt, rekursive Algorithmen zu programmieren.

z. B.

```
int power (int n, int exp) {
  if (exp > 0) return n*power(n, exp-1);
  return 1;
}
```

Achtung:

Die Verwendung rekursiver Algorithmen kann eine bedeutende Auswirkung auf die Leistungsfähigkeit der Anwendung und die Benutzung des Stapels (stack) haben.

Wiedereintrittsfähiger Code – Bibliotheken

- **In der Informatik bezeichnet die Wiedereintrittsfähigkeit die Eigenschaft einer Funktion, gleichzeitig von mehreren Aufgaben benutzt werden zu können. Der Wiedereintritt (re-entry) erlaubt, Duplikationen von Funktionen im Arbeitsspeicher zu vermeiden, die gleichzeitig durch mehrere Anwendungen benutzt werden.**

- **Eine Funktion ist unter den folgenden Bedingungen wiedereintrittsfähig:**
  - Sie darf keine statischen (globalen), nicht konstanten Daten verwenden
  - Sie darf keine statischen (globalen), nicht konstanten Adressen zurückgeben
  - Sie darf nur Daten verarbeiten, die durch das aufrufende Programm geliefert werden
  - Sie darf ihren eigenen Code nicht verändern
  - Sie darf keine nicht wiedereintrittsfähigen Unterprogramme aufrufen