Programmeertechnieken
Week 3

Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/pt2017/

Universiteit Leiden
The Netherlands

Vorige week

- Scripttalen (Python)
- Fasen in softwarecompilatie
- Shared objects
- Makefiles
- Build systems

Deze week

- Deze week staat in het teken van "low-level programming".
- We moeten ons te allen tijde beseffen dat we een machine aan het programmeren zijn.
  - Dit begrip helpt bij het programmeren.
  - Je zult het misschien niet geloven: maar met dit in het achterhoofd wordt het werken met pointers makkelijker!
- Daarnaast is dit basiskennis nodig bij low-level vakken als Computer Architectuur, Operating Systems, Compiler Constructie, ...

De machine

- Wat voor machine programmeren we nu precies?

Instructies

- De Central Processing Unit (CPU) voert instructies uit.
- Deze kunnen worden ingedeeld in verschillende klassen.
- De belangrijkste zijn:
  - Registeroperaties.
  - Load/store operaties op het RAM geheugen.

Instructies (2)

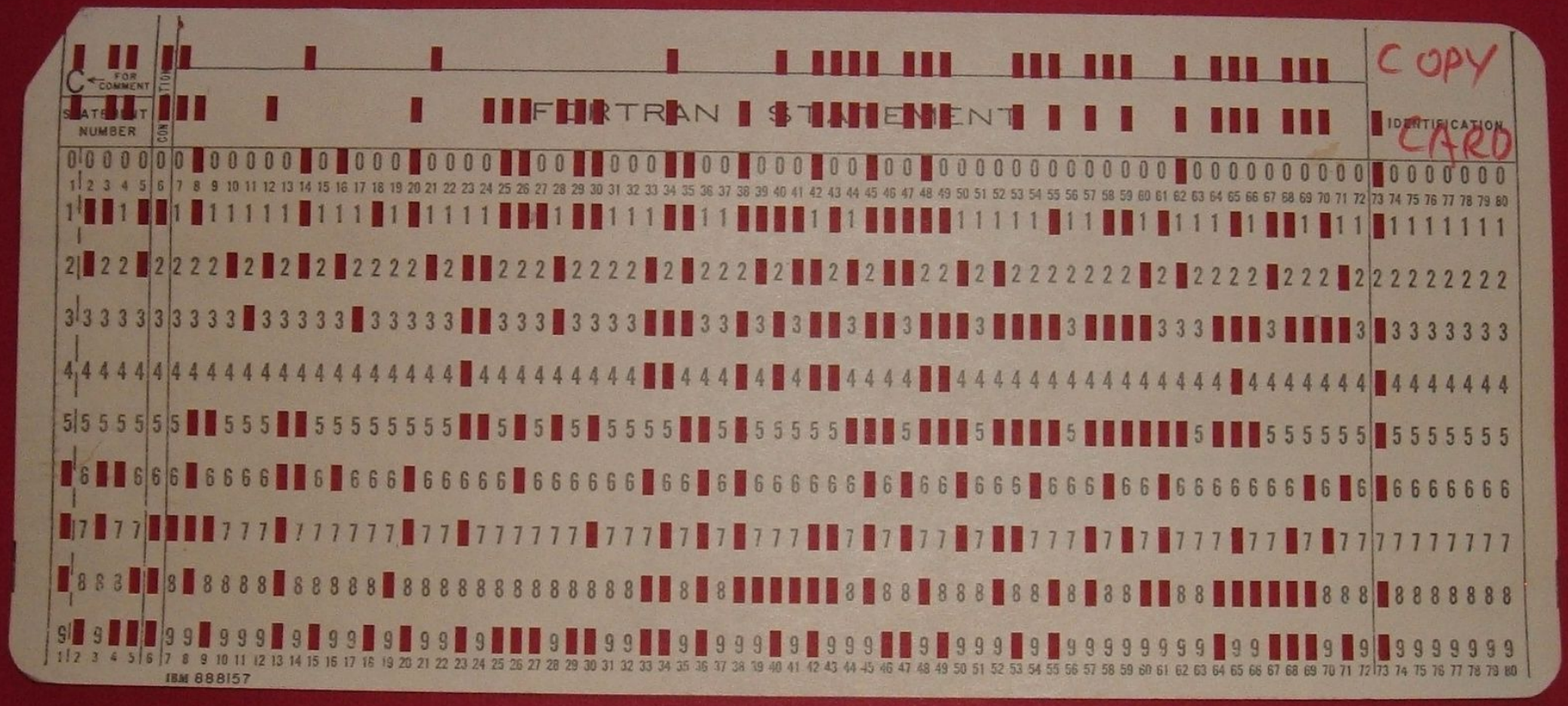
- CPU instructies worden geencodeerd in een binair formaat.
- Voorbeeld:

```
00000001111111000100010110000011
```

- Programma's zoals uitgevoerd door de CPU zijn dus sequenties van dit soort codes.
- Programma's op deze manier met de hand schrijven kost zeer veel tijd en zeer gevoelig voor fouten.

Ponskaarten

- Toch moest men vroeger wel …



Source: https://en.wikipedia.org/wiki/File:IBM1130CopyCard.agr.jpg

Instructies (3)

- *Assembly language:* CPU instructies geschreven in ASCII code in een door de mens leesbaar formaat.

```
addl    $0x1,-0x4(%rbp)
```

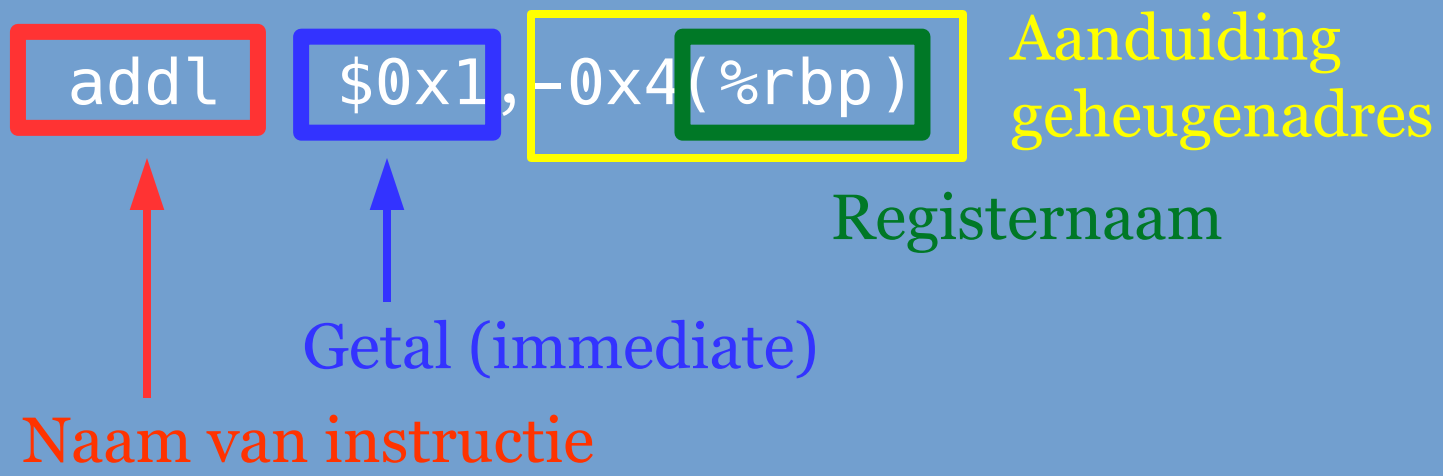
vertaalt naar

```
00000001111111000100010110000011
```

- Een "assembler" vertaalt de door de mens leesbare vorm naar de binaire machinecode.

Instructies (4)

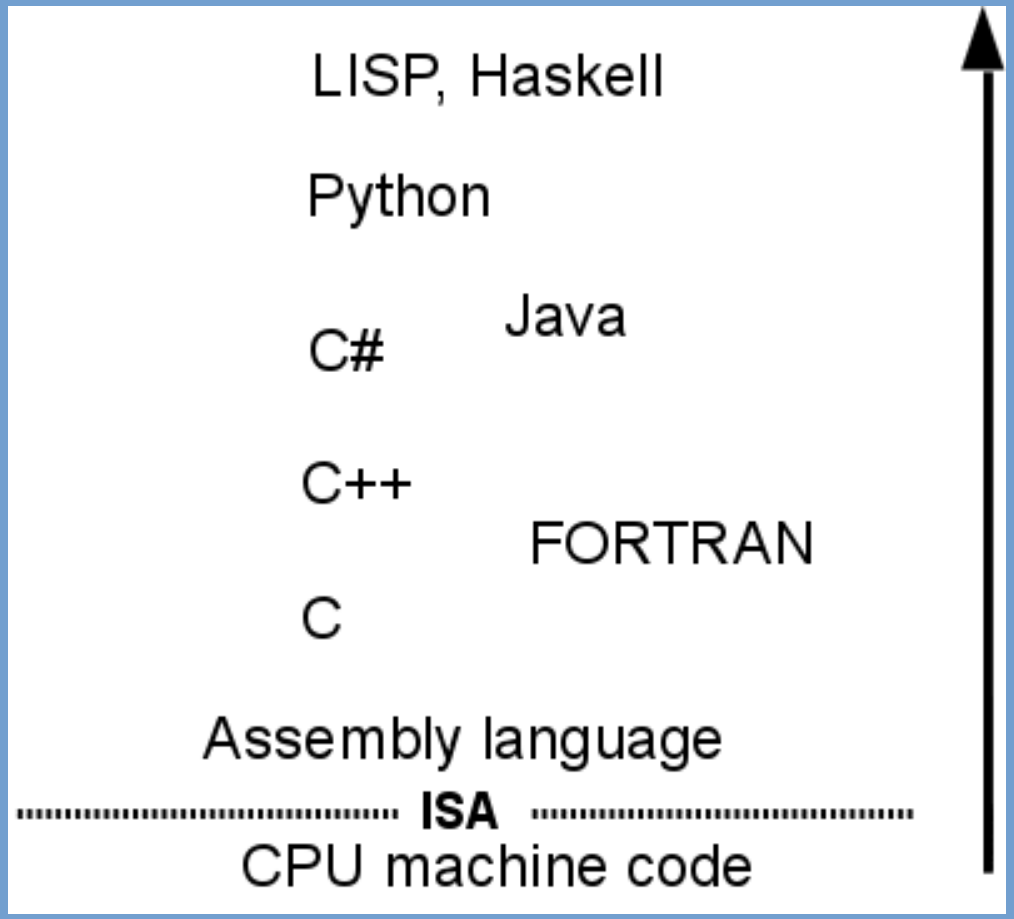
- Wat zien we in de instructie terug:



Low - high level languages

- Het schrijven van programma's in assembly is nog steeds een hoop werk.
- Er zijn daardoor programmeertalen bedacht, die op een hoger niveau staan en de programmeur werk besparen. "Higher-level languages".
  - Voorbeelden: FORTRAN, C.
- Met een "compiler" vertalen we de high-level taal naar assembly, de assembler vertaalt dat weer naar machinecode.
- Laag boven laag: abstractielagen. Komt overal in de informatica terug.

Low - high level languages (2)



C / C++

- Hoewel C tegenwoordig als low-level wordt gezien, noemen sommige mensen C een high-level taal.
  - (Soms wordt C ook wel een "macro assembly language" genoemd, omdat het redelijk dicht op assembly niveau zit.)
- C wordt veel voor "low-level programming" gebruikt, waarbij er hardware moet worden geprogrammeerd. Besturingssystemen, device drivers...
- C++ werd high-level geacht, maar door de komst Java/C# verschuift het meer naar low-level.

C / C++ (2)

- Voorbeeld, `c = (b + 4) * a` vertaald naar x86_64 assembly:

```
movl    $0xea,-0xc(%rbp)  # initialiseer a met 234.

movl    $0x22,-0x8(%rbp)  # initialiseer b met 34.

mov     -0x8(%rbp),%eax   # laad b in %eax
add     $0x4,%eax         # tel 4 op bij %eax
imul    -0xc(%rbp),%eax   # vermenigvuldig %eax met waarde a.
mov     %eax,-0x4(%rbp)   # zet resultaat in geheugen.
```

C / C++ (3)

- Om een goed programmeur te worden, moeten we begrijpen wat voor machine we aan het programmeren zijn.
- Relatie code - machine nooit uit het oog verliezen.
- Het is dus belangrijk om de relatie tussen C en assembly goed te begrijpen.
- Bestuderen assembly code komt verder terug bij Computer Architectuur.

Variabelen

- Wat is een variabele?
- In feite een constructie in programmeertalen om een waarde op te slaan.
- Assembly language kent geen variabelen, alleen registers, geheugenadressen.
  - We bepalen hier zelf waar en hoe we de variabele opslaan.
- In C doet de compiler dat voor ons:
  - Met een declaratie wordt in principe een geheugenlocatie gereserveerd.
  - Daarnaast worden load/store instructies ingevoegd waar nodig.

Variabelen (2)

- Voor een variabele wordt dus een geheugenlocatie gereserveerd.
- Uit het type van de variabele volgt de lengte van de variabele en het formaat.
  - `sizeof(int)` geeft je de lengte van een int variabele in bytes.
- `int`, 4 bytes en signed, getal -135 opgeslagen als `11111111111111111111111101111001`
- `double`, 8 bytes en floating-point, getal 3.14159 opgeslagen als `1000000000001001001000011111100111111000000011011100001100110110`

  (eigenlijk `3.1415899999999999`, afronding!)

"&" operator

- Omdat er voor een variabele een geheugenlocatie wordt gereserveerd, kunnen we het adres van die locatie opvragen.
- "&" operator.

```
int a;
printf ("addr: %p\n", &a);
```

Pointers

- Een "pointer" is een variabele die een geheugenadres bevat.
- Pointers worden gebruikt om te wijzen naar data op een bepaalde plek in het geheugen.
- Een pointervariabele heeft een vaste grootte, meestal 4 of 8 bytes, afhankelijk van het platform.
  - (Hier komt het verschil tussen 32-bit en 64-bit systemen vandaan).

Pointers (2)

- We kunnen het resultaat van `&a` in een pointer opslaan: `int *b = &a;`.
- `b` bevat nu het geheugenadres van de variabele `a`.
- Wat kunnen we nu met deze pointer? Hoe kunnen we het geheugen benaderen?

Pointers (3)

- *-operator. Ook wel de "dereference" operator genoemd.
- Als * voor de naam van een pointer wordt gezet, vraag je de compiler het geheugenadres in die pointer te benaderen.
- Dus om een load of store instructie naar dat adres uit te voeren.

```
int a = 13;
int *b = &a;
int c = *b;  // 13

*b = 27;
/* a is nu 27, c nog steeds 13 */
```

Pointers (4)

- Opgepast!! Het geheugenadres moet wel valide zijn.
- Verkeerd adres: gedrag is niet bepaald. Programma kan crashen.

```
int *a = (int *)0xabcdef;
int *b = (int *)0x0;

*a;
*b;
```

Intermezzo: casting

- `float a = 3;`
  - De C compiler zal zelf de integer 3 omzetten naar een floating-point.
- We mogen dat ook expliciet maken, de compiler voegt een typeconversie toe:
  - `float a = (float)3;`
- Soms kan of wil de compiler een waarde niet converteren:
  - Bijvoorbeeld een integer naar een pointer.
  - Of het type van de pointer veranderen.
- We gebruiken dan een cast om aan te geven dat het in orde is de data te herinterpreteren. Op onze verantwoordelijkheid en eigen risico!

Pointers (4)

- Pointers spelen een belangrijke rol in C. We kunnen gaan rekenen met deze geheugenadressen.
- Nodig voor allerlei "low-level" zaken zoals schrijven besturingssystemen en device drivers.
- Allerlei "low-level" manipulatie mogelijk in assembly is door pointers ook mogelijk in C.
- Merk op dat vele echte "high-level" programmeertalen dergelijke manipulatie niet toestaan.

Rekenen met pointers

- Er kan worden gerekend met pointers door een waarde bij een pointer op te tellen.
- Let op! Er gebeurt iets interessants!

```
int *a = (int *)0x100;
int *b = a + 1;

// Wat is de waarde van b?
```

Rekenen met pointers (2)

- Wat gebeurt er? De waarde die moet worden opgeteld wordt eerst vermenigvuldigd met de grootte van het element waarnaar de pointer wijst.

```
b = a + 1 * sizeof(int)
```

- Dus eigenlijk: met "+ 1" verplaatsen we de pointer naar de locatie van het volgende element van hetzelfde type.

Relatie met arrays

- Een array wordt in het geheugen opgeslagen door alle elementen van de array achter elkaar te zetten.
- Hoe berekenen we het geheugenadres van een individueel element `n`?

```
addr = begin + n * sizeof(type)
```

- Dit hebben we eerder gezien! Wat doet?

```
int A[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
*(A + 3)
```

Relatie met arrays (2)

- Blijf wel altijd opletten.

```
int A[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
*(A + 3)    // vierde element in A
            //  (+ 3 * sizeof(int))
*(A + 11)   // twaalfde (?) element in A
            //  (+ 11 * sizeof(int))
*(&A + 3)   // ???
            // (+ 3 * sizeof(A))
```

- Er is *geen* beveiliging tegen het lezen voorbij het einde van de array!

Rekenen met pointers (3)

- Ik wil niet dat er met `sizeof(type)` wordt vermenigvuldigd. Wat dan?
- In dat geval is het gebruikelijk om te rekenen met `uintptr_t` en dan te casten naar het pointer type.
- LET OP! Een `unsigned int` *hoeft niet* even groot te zijn als een pointer!!!

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8 };
uintptr_t test = (uintptr_t)&A + sizeof(int) * 3;
int *b = (int *)test;
```

void pointers

- Hoe zit dat met `void *`?
- Een pointer, dus de variabele bevat een geheugenadres.
- Echter, het type waarnaar we wijzen is niet bekend.
- Een cast is nodig voordat we kunnen dereferencen.

```
int my_callback(int a, void *user_data)
{
  int *my_data = (int *)user_data;
  printf("%d\n", *my_data);
}
```

Dubbele pointers

- Wat is `int **` ?
- Een pointer naar (een pointer naar (een int)).

```
int a = 13;
int *b = &a;
int **c = &b;

printf("%d\n", **c);
```

- Quiz! `c + 1`, hoeveel wordt er bij `c` opgeteld?

Dubbele pointers (2)

- Heeft dit nut? Ja, bijvoorbeeld als functieargument. De functie kan de waarde van de pointer zetten.

```
void set_pointer(int **a)
{
  *a = (int *)0x100;
}

int *a = NULL;
set_pointer(&a);
```

Driedubbele pointers

- Driedubbele pointers komen ook soms voor in code.
- Vierdubbele en hoger bijna nooit.

Structures

- Structures worden vaak gebruikt om variabelen te groeperen.
- De compiler berekent de byte-offset waar elk veld terecht komt.

```
struct MyRecord
{
    unsigned int sid;
    char name[64];
    uint8_t age;
};
```

- Er is een `offsetof(structure, member)` macro om te bepalen op welke offset een veld wordt geplaatst.

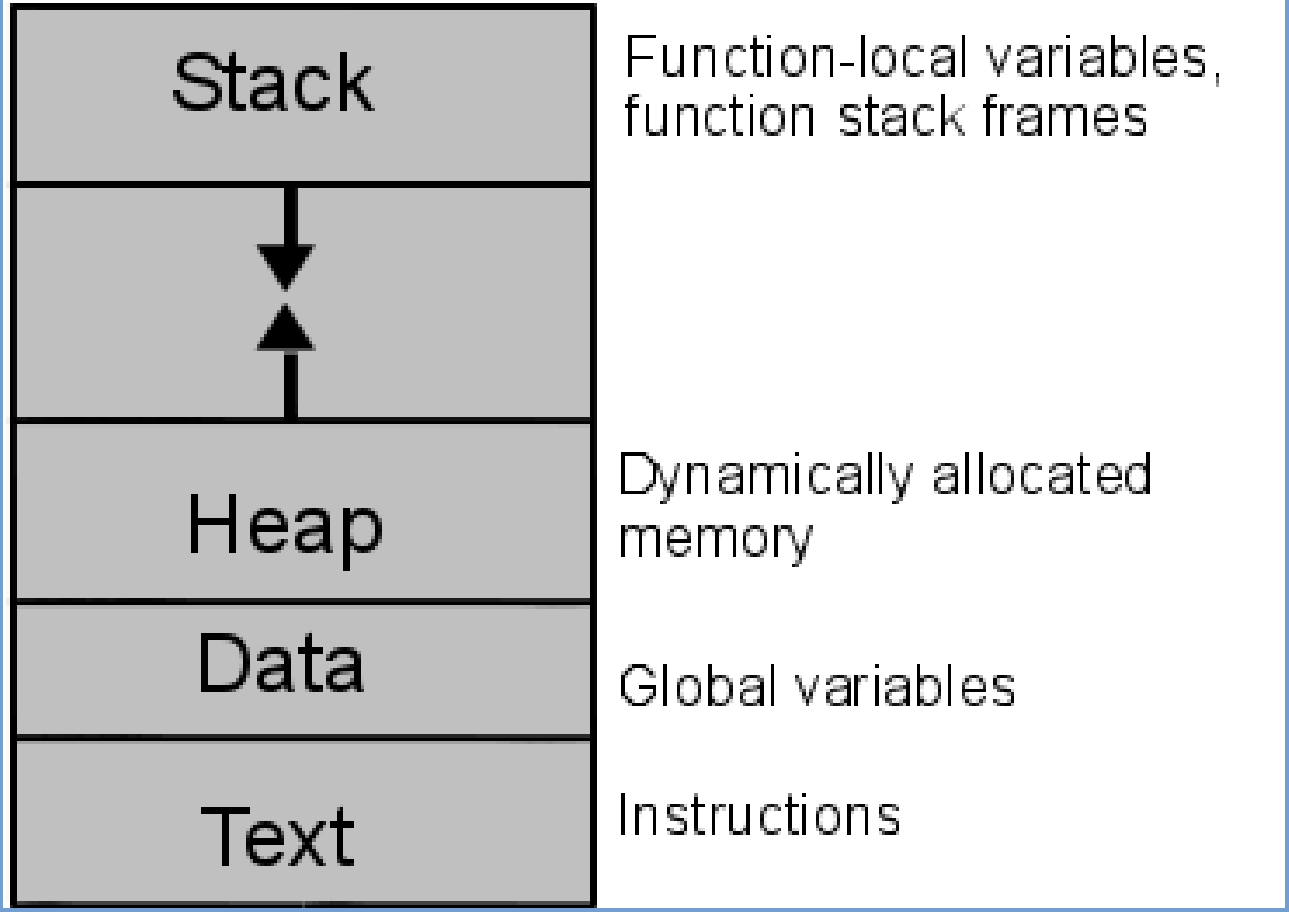
Structures (2)

- De operator -> is eigenlijk een afkorting voor het gebruik van de dereference operator gevolgd door de ".".

```
struct MyRecord r;
puts(r.name);

struct MyRecord *t;
t = (struct MyRecord *)malloc(sizeof(struct MyRecord));
puts((*t).name);
puts(t->name);
```

Programma's in het geheugen



Functie-aanroepen

- Alle functies staan in het geheugen op een bepaald adres.

0x401cd0 <exit>

0x4173c0 <write>

0x401164 <main>

0x4021c0 <_IO_puts>

0x402120 <_IO_printf>

0x41fb70 <abort>

Functie-aanroepen (2)

- Functie aanroep: we willen een andere functie uitvoeren en daarna terugspringen naar waar we waren gebleven.

```
int main(void)
{
    …
    …
    puts("hello world");
    …
    …
    return 0;
}
```

```
int puts(const char *s)
{
    …
    …
    …
    return n;
}
```

Functie-aanroepen (3)

- Hoe zit dat in assembly? Er is vaak een aparte instructie voor:
  - "call"
  - "jsr" - jump subroutine
  - "bal" - branch and link

```
0000000000401164 <main>:
  401164:       push   %rbp
  401165:       mov    %rsp,%rbp
  401168:       sub    $0x50,%rsp
  40116c:       movl   $0xd,-0x50(%rbp)
...
  4011e8:       mov    $0x496064,%edi
  4011ed:       callq  4021c0 <_IO_puts>
  4011f2:       mov    $0x496070,%eax
  4011f7:       mov    -0x10(%rbp),%edx
  4011fa:       mov    %edx,%esi
  4011fc:       mov    %rax,%rdi
  4011ff:       mov    $0x0,%eax
  401204:       callq  402120 <_IO_printf>
```

```
00000000004021c0 <_IO_puts>:
  4021c0:       push   %r12
  4021c2:       mov    %rdi,%r12
  4021c5:       push   %rbp
  4021c6:       push   %rbx
  4021c7:       callq  40c2c0 <strlen>
...
  4022d2:       mov    %ebp,%eax
  4022d4:       pop    %rbp
  4022d5:       pop    %r12
  4022d7:       retq
```

Functie-aanroepen (4)

- Hoe worden de waarden voor argumenten van de functie doorgegeven?
- Meerdere mogelijkheden:
  - Via CPU registers.
  - Door deze op de stack te plaasten.
  - Combinatie van beiden.
- De exacte methode hangt af van de CPU architectuur. Soms worden er per architectuur nog meerdere manieren gebruikt.
- Dergelijke methoden worden "Calling Conventions" genoemd.

Belang van ABI

- We zien nu dat om een functie te kunnen aanroepen de argumenten op de juiste plek moeten zijn gezet.
  - We moeten dus weten welke calling convention we moeten gebruiken.
  - We moeten ook het exacte aantal en type argumeten weten!
- Alle modules die met elkaar moeten samenwerken (shared object, executable) moeten dezelfde calling convention gebruiken.
- De manier waarop functie-argumenten moeten worden gecommuniceerd maakt deel uit van de ABI, "Application Binary Interface". Vergelijk API.

Belang van ABI (2)

- (1) We maken een shared object met een functie

```
int telop(int a, int b)
```

- (2) We maken een programma dat deze functie gebruikt.
- (3) We passen het shared object aan:

```
int telop(int a, int b, int c).
```

- We gebeurt er nu met het testprogramma?

Belang van ABI (3)

- Niet gedefinieerd! Zou kunnen crashen.
- Het testprogramma was nog gecompileerd voor de oude "function signature".
- De API is veranderd en ABI ook => probleem!
- ABI compatibiliteit is verbroken en als een gevolg moet alle software die het shared object gebruikt opnieuw worden gecompileerd!
  - Erger nog: alle source code moet worden aangepast om het extra argument mee te geven! (API compatibiliteit verbroken).

Belang van ABI (4)

- Ben je de enige gebruiker van het shared object? Dan is dit niet zo'n groot probleem.
- Als het shared object ook door andere mensen wordt gebruikt: boze gebruikers!
- Library maintenance: ABI compatibiliteit moet behouden blijven.
- Herinner je bijvoorbeeld de bijna 60 libraries waar the GIMP een dependency op heeft.

Belang van ABI (5)

- Als we zouden hebben toegevoegd

  ```
  int telop3(int a, int b, int c)
  ```

  dan hebben we deze problemen niet.
- De oude functie blijft bestaan, dus bestaande software blijft functioneren.
- We spreken wel van een "API addition".
- Ook belangrijk: het aanpassen van structures breekt ook ABI compatibiliteit! (waarom?)

Lokale variabelen

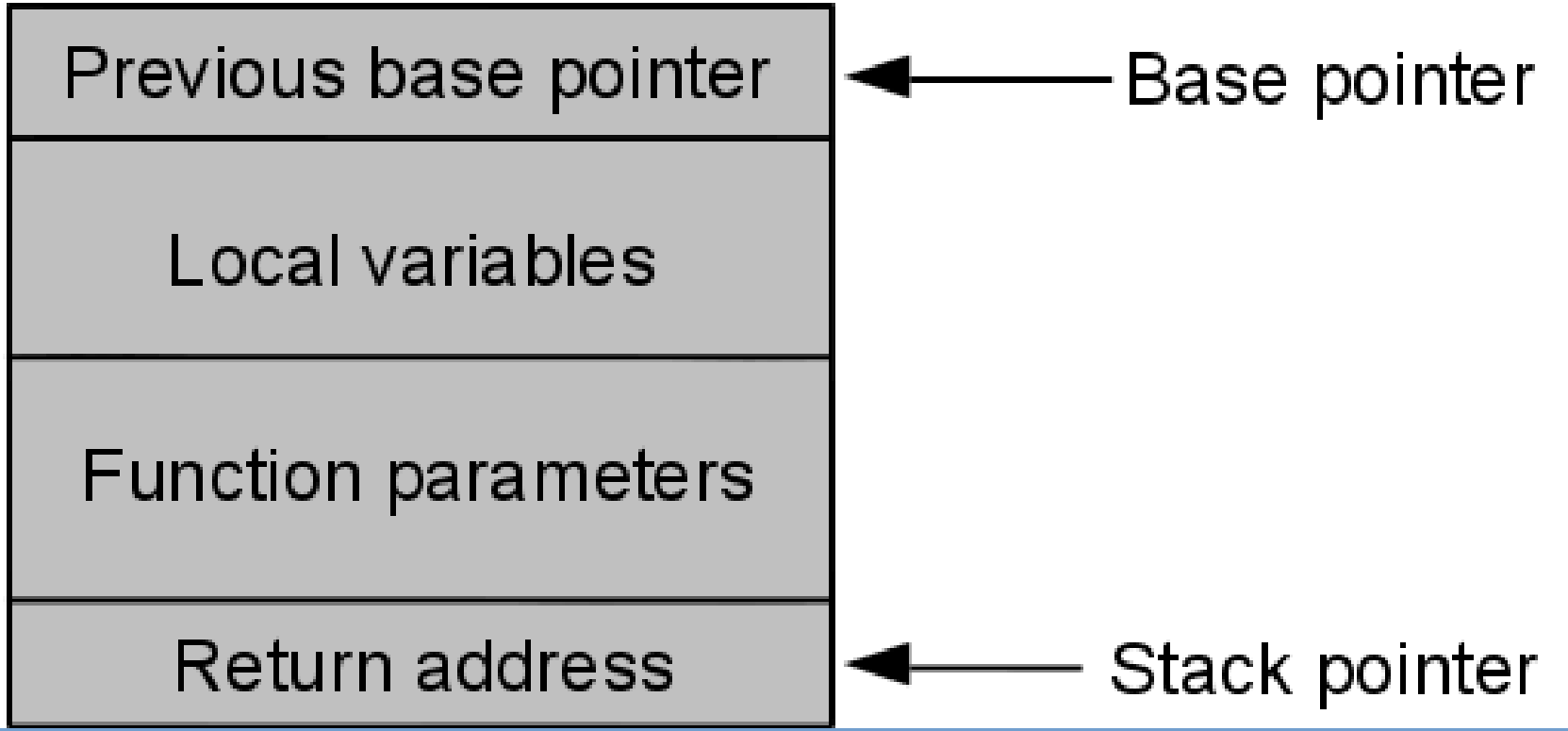
- Terug naar onze functie-aanroepen.
- Functies hebben lokale variabelen, deze worden opgeslagen op de stack.
- Een functie heeft een "proloog" en "epiloog" om deze opslag aan te maken en weg te gooien.

```
0000000000401164 <main>:
  401164:       55                      push   %rbp
  401165:       48 89 e5                mov    %rsp,%rbp
  401168:       48 83 ec 50             sub    $0x50,%rsp
  40116c:       c7 45 b0 0d 00 00 00    movl   $0xd,-0x50(%rbp)
  401173:       c7 45 b4 15 00 00 00    movl   $0x15,-0x4c(%rbp)
  40117a:       c7 45 b8 44 00 00 00    movl   $0x44,-0x48(%rbp)
  401181:       c7 45 bc 3d 00 00 00    movl   $0x3d,-0x44(%rbp)
  401188:       c7 45 c0 16 00 00 00    movl   $0x16,-0x40(%rbp)
  40118f:       c7 45 c4 59 00 00 00    movl   $0x59,-0x3c(%rbp)
  401196:       c7 45 c8 18 00 00 00    movl   $0x18,-0x38(%rbp)
...
  401244:       c9                      leaveq
  401245:       c3                      retq
```

Stack frames

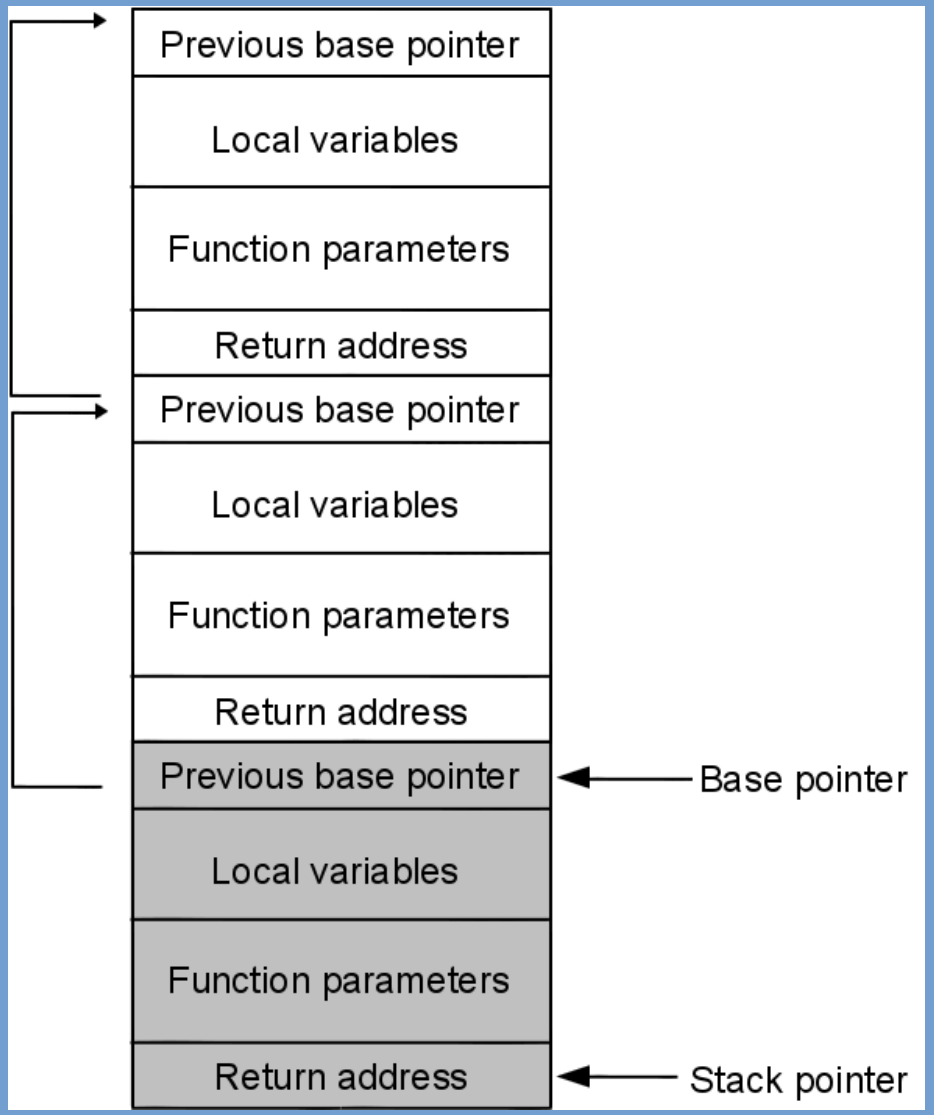
- Alle data die behoort tot een functie-aanroep wordt geslagen in een "stack frame" op de stack.
- Denk aan:
  - Lokale variabelen.
  - Adres waar we moeten terugspringen.
  - Waarden van registers die we in deze functie overschrijven.
- Onthoud: Als een functie is afgelopen, moeten we alles terugzetten in de toestand waarin deze functie was aangeroepen. Op deze manier kan de aanroeper verder waar deze was gebleven.

Stack frames (2)



Stack frames (3)

- Functies roepen vaak weer functies aan. Steeds weer wordt een stack frame toegevoegd.



Bit fiddling

- Iets anders dat nodig is bij het low-level programmeren, is het kunnen wijzigen van individuele bits in een binaire waarde.
  - Value packing.
  - Manipulatie van geheugenadressen (afronden voor alignment).
- Hiervoor kunnen we de "bitwise operators" gebruiken:
  - &, |: bitwise AND, OR.
  - ^: bitwise XOR.
  - ~: (unair) bit inversion.
  - <<, >>: shift left, right.

Bit fiddling (2)

- Uitlezen individuele bits.
  - `(value & 0x1)`
  - `(value & 32) == 32`
  - `(value & (1 << 5)) == (1 << 5)`

```
 01111011
 00100000
 ---------  &
 00100000
```

Bit fiddling (3)

- Zetten van een bit.
  - `value |= (1 << 5)`
- Bit op nul zetten.
  - `value |= 0` heeft weinig effect …
  - `value &= ~(1 << 5)`

```
 01001011
 00100000
 --------  |
 01101011

 01111011
 11011111
 --------  &
 01011011
```

Bit fiddling (4)

- Een groep van bits isoleren (bitmasker / bit masking).
  - `0xf = 0b1111`
  - `value & 0xf`
  - `value & (0xf << 4)`

```
 01111011
 00001111
 --------  &
 00001011

 01111011
 11110000
 --------  &
 01110000
```

Bit fiddling (5)

- Nog een voorbeeld hoeveel controle we over geheugenlayout hebben.

```
uint64_t values = 0x0;
uint8_t *bytes = (uint8_t *)&values;

bytes[0] = 0xee;

uint8_t tmp = values & 0xff;
```

- Wat staat er in `tmp` ?

Bit fiddling (6)

```
uint64_t values = 0x0;
uint8_t *bytes = (uint8_t *)&values;

bytes[0] = 0xee;

uint8_t tmp = values & 0xff;
```

- Wat staat er in `tmp` (op een x86_64 machine) ?
- Little vs. big endian.
  - Volgorde waarin bytes in een word worden opgeslagen.
- We zitten echt “low-level”, het gedrag hangt af van de CPU!

Little. vs Big Endian

- Little endian: least significant byte **eerst**.
  - values: 0xff, 0x0, 0x0, ..., 0x0
  - bytes: b[0], b[1], b[2], ..., b[7]
- Big endian: most significant byte **eerst**
  - values: 0x0, 0x0, 0x0, ..., 0xff
  - bytes: b[0], b[1], b[2], ..., b[7]
- Dus op een big-endian machine geeft “values & 0xff” de waarde 0x0!

Declaraties

- Gaat hier een alarmbel af?

```
int *a, b;
```

Declaraties (2)

- Gaat hier een alarmbel af?

```
int *a, b;
```

- U mag kiezen:
  - a is een int*, b is een int*.
  - a is een int*, b is een int.
  - Compiler error.

Declaraties (2)

- Gaat hier een alarmbel af?

```
int *a, b;
```

- U mag kiezen:
  - a is een int*, b is een int*.
  - **a is een int*, b is een int.**
  - Compiler error.

Arrays & pointers

```
int *A[16];
vs.
int (*A)[16];
```

- Is er een verschil? Zo ja, wat?

Arrays & pointers (2)

- `int *A[16]`: een array van 16 pointers naar integers.
- `int (*A)[16]`: een pointer naar een array van 16 integers.

Arrays & pointers (3)

```
int A[16] = { 0, };

A + 1    // int *
&A       // int (*)[16]
&A + 1   // int (*)[16]
&A[0]    // int *
```

Arrays & pointers (4)

```
int B[16][16] = { 0, };

&B        // int (*)[16][16]
&B + 1    // (schuift 1 2-d array op)
B + 1     // int (*)[16]
          // (schuift 1 rij op)
&B[0]     // int (*)[16]
&B[0][0]  // int *
B[0]      // int [16]
B[0] + 1  // int *
```

Huiswerk 2

- Om te oefenen met pointer arithmetic en bitwise operators: een aantal opgaven in Huiswerk 2.
- Mag weer in tweetallen.
- Deadline: vrijdag 3 maart.

Geheugenallocatie

- We zagen eerder al de "heap" waar ruimte is voor dynamisch gealloceerd geheugen.
- Deze allocaties worden gemaakt met `new` in C++, `malloc()` in C.
- Returnwaarde: een pointer naar het begin van het gereserveerde blok geheugen.
- Je moet zelf het geheugen vrijgeven als je klaar bent! Anders blijft de ruimte gereserveerd staan in de heap.
  - `delete` in C++, `free()` in C.

Geheugenallocatie (2)

- `void *malloc(size_t size);`
- `malloc` heeft 1 argument: hoeveel bytes we willen alloceren.
- Berekenen met behulp van `sizeof`.

```
double *A = (double *)malloc(sizeof(double) *10*10);
...
free(A);
```

Geheugenallocatie (3)

- Stel we alloceren geheugen en overschrijven de pointer:

```
int *a = (int *)malloc(sizeof(int) * 16);
a = 0xc0ffee;
```

- We hebben nu geen pointer meer naar het voor ons gereserveerde geheugen!
- We kunnen het dus ook niet meer vrijgeven.
- We spreken van een "memory leak".
- Een probleem voor programma's die langdurig actief zijn!

Geheugenallocatie (4)

- `malloc()` regelt de allocatie van het gevraagde geheugen.
  - Het voert "system calls" uit als nodig.
  - Het beheert ruimte op de heap.
  - Enz.
- Je bent vrij om `malloc()` te vervangen met een zelfgeschreven geheugenallocator.
- Dit wordt soms gedaan om de allocator te optimaliseren voor een bepaald scenario.
  - Bijvoorbeeld heel vaak objecten van dezelfde grootte alloceren.

Memory errors

- Al deze flexibiliteit in het werken met het geheugen is natuurlijk prachtig.
- Maar er kunnen ook een hoop dingen fout gaan als zaken niet goed worden gedaan…
- Leidt niet alleen tot crashes, maar ook tot security problemen!
- Een kort overzicht.

Memory errors (2)

- Aanroep `free()` met ongeldig geheugenadres.
  - 'Random' overschreven pointer.
  - Geheugen dat niet met `malloc()` was gealloceerd.
- "Double free". `free()` wordt meer dan 1 keer aangeroepen voor een bepaald geheugenadres.
- "Out of bounds" geheugentoegang overschrijft "boekhouddata" van `malloc()`, hierdoor kunnen ook `free()` calls mislukken.

Memory errors (3)

- "Out of bounds" geheugentoegang leidt vaak tot problemen:
  - Onverwachte data wordt gelezen. Interessant gedrag, maar geen crash.
  - Andere data wordt overschreven! Kan leiden tot crash.
  - Vooral het geval als een gedeelte van de stack wordt overschreven! "Smashed stack".

Memory errors (4)

- Veroorzaakt door:
  - Fouten in pointer arithmetic.
  - Schrijven voorbij einde array (of voor het begin). Off-by-one errors ...
  - Werken met een "busted pointer" die toch naar een geldig stuk geheugen wijst.
  - Schrijven naar geheugen dat al is vrijgegeven.

Memory errors (5)

- Nog een veelvoorkomend probleem: het lezen van niet-geinitialiseerde data.
- Leidt niet direct tot een crash, dus soms moeilijk te vinden.
- Initialiseer altijd variabelen!
- `malloc()` initialiseert het geheugen *NIET*! Doe dit altijd zelf!
  - Bijv. `memset()`.
  - Of gebruik `calloc()`.
  - Andere routines: ga altijd na in de documentatie of geheugen wordt geinitialiseerd!

Memory errors (6)

- Hoe zit dat met new in C++?
  - Allocatie arrays: geen initialisatie.
  - Allocatie objecten: constructor wordt aangeroepen en een goede constructor initialiseert de data.

Memory errors (7)

- Bij het werken met strings in C moet ook worden opgepast.
- Strings moeten "null terminated" zijn.
- Zelf opletten dat de string in de gealloceerde array past bij het uitvoeren van operaties. De array groeit *niet* vanzelf mee.

Memory errors (8)

```
char buffer[16];
sprintf(buffer, "dsfkjhasdfkjhasdfkjhasdf");

// beter:
snprintf(buffer, 16,
         "dsfkjhasdfkjhasdfkjhasdf");
```

Memory errors (9)

- Wat kan hier foutgaan?

```
char dst[16];
strcpy(dst, src);
```

Memory errors (10)

```
char dst[16];
// Kan nog steeds foutgaan!
strncpy(dst, src, 16);

// Nog beter:
char dst[16];
strncpy(dst, src, 15);
dst[15] = 0;
```

Memory errors (11)

- Wat kan hier fout gaan?

```
char dst[20] = "TEST1";
strncpy(dst + 5, src, 20);
dst[20] = 0;
```

Memory errors (12)

```
char dst[20] = "TEST1";
strncpy(dst + 5, src, 20 - 5 - 1);
dst[20] = 0;
```

Memory errors (13)

```
char *a = (char *)malloc (strlen(b));
strcpy(a, b);
```

Memory errors (14)

- `strlen( )` telt de 0 aan het einde niet mee ...

```
char *a = (char *)malloc (strlen(b) + 1);
strcpy(a, b);
```

- Of gebruik `strdup( )`.

Memory errors (15)

- Pas dus heel erg goed op met het gebruik van C strings.
- Bij twijfel: altijd documentatie raadplegen.
- Als je C++ gebruikt: geef de voorkeur aan `std::string`!

Debugging

- We hebben nu aardig wat bugs gezien.
- Gelukkig zijn er tools om ons te helpen met het vinden en oplossen van dit soort problemen.
- Bug fixing, debugging.

Debuggers

- Met een "debugger" kunnen we een programma stopzetten en er stap voor stap doorheen lopen.
- We kunnen ook alle variabelen uitlezen op elk moment in het programma.
- Ook kunnen we zien welke functies elkaar hebben aangeroepen: een overzicht van de stack frames (een "backtrace").
- Er zijn text-based en grafische debuggers.

Debuggers (2)

- Op Linux: text-based debugger "gdb".
- Met commando "print" kun je variabelen uitlezen, maar je kunt er ook mee rekenen.
  - Handig: gdb kent alle regels voor pointer arithmetic!
- Met "bt" krijg je een backtrace te zien.
- Stappen door het programma: "step", "next", "continue", ...
- Probeer tijdens het werkcollege deze en volgende week eens met "gdb" te oefenen: beste manier om het onder de knie te krijgen.
  - Oefenmateriaal is beschikbaar: zie de website.

Debuggers (3)

- Veel "memory errors" zijn met gdb nog steeds lastig op te sporen.
- Vooral als een "out of bound" schrijfactie stiekem de waarde van een variabele verandert.
- Soms een langdurig zoekproces ...

Debuggers (4)

- Er bestaat een tool voor het opsporen van "memory errors": ***valgrind***.
- Draait in ieder geval op Linux.
- Valgrind interpreteert het programma en onthoudt alle geheugenallocaties.
- Het controleert alle lees en schrijfacties naar het geheugen.
- Als het een fout vindt, laat valgrind dat weten.

Debuggers (5)

```
==10365== Invalid write of size 1
==10365==    at 0x4C2BFFC: strcpy (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10365==    by 0x400670: main (bla.c:12)
==10365==  Address 0x7ff001000 is not stack'd, malloc'd or (recently)
free'd
==10365==
```

Debuggers (6)

- Hoe draaien?

```
valgrind ./mijnprogramma
```

Volgende week

- Advanced C++ Programming.