Compiler Construction 2016/2017
Storage Allocation

Peter Thiemann

November 16, 2016

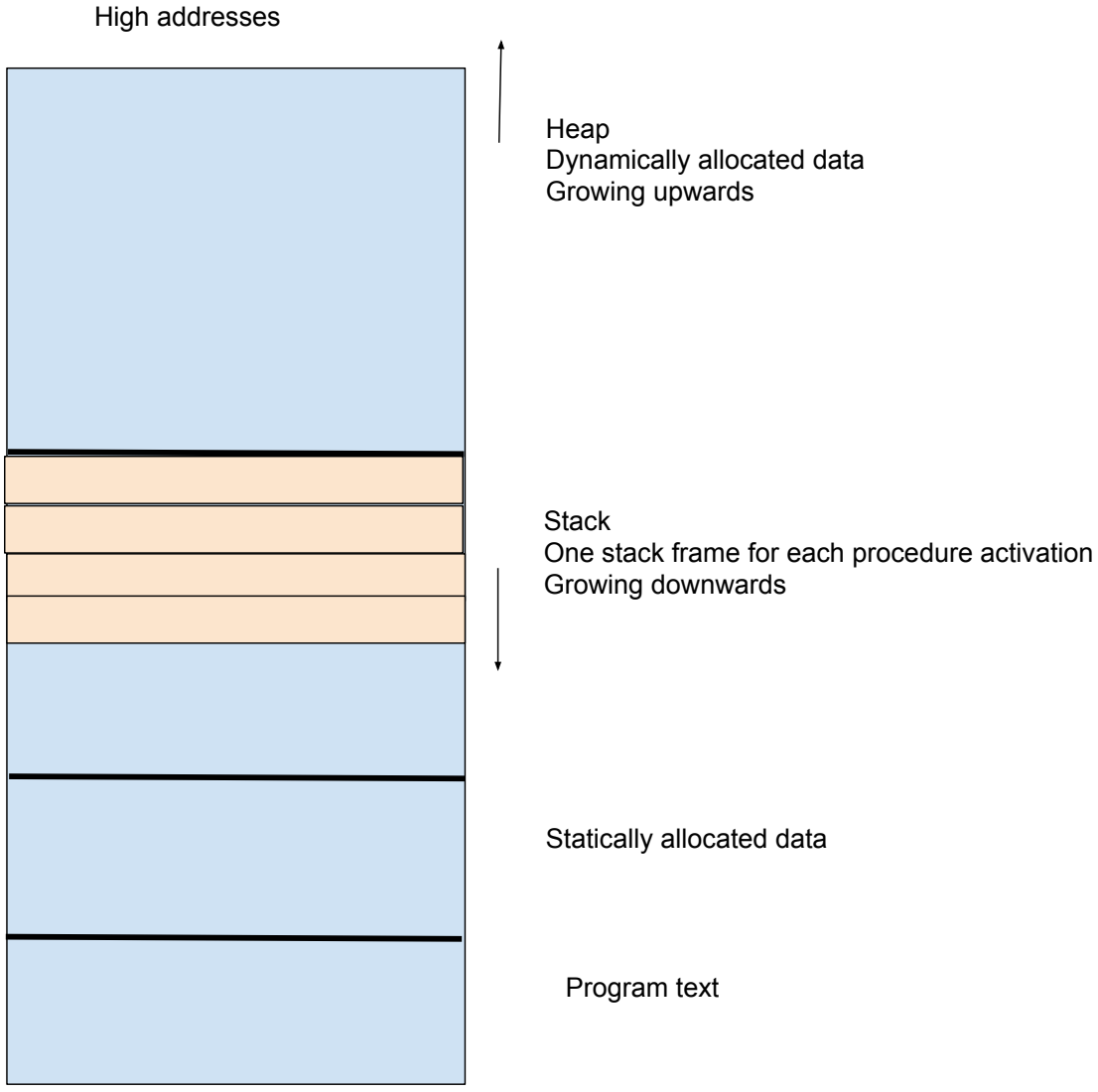
Outline

Storage Organization and Allocation

The compiler organizes data according to different aspects.

- extent
  - entire program run — static allocation
  - coupled to a procedure invocation — stack allocation
  - independent of program structure — heap allocation
- size (in bytes)
  - depending on type
  - architecture
- alignment
  - constraint on base address of a datatype
  - multiple of 2^n, for some n depending on size of data
  - architectural constraints
    unaligned access either slow or leads to memory fault

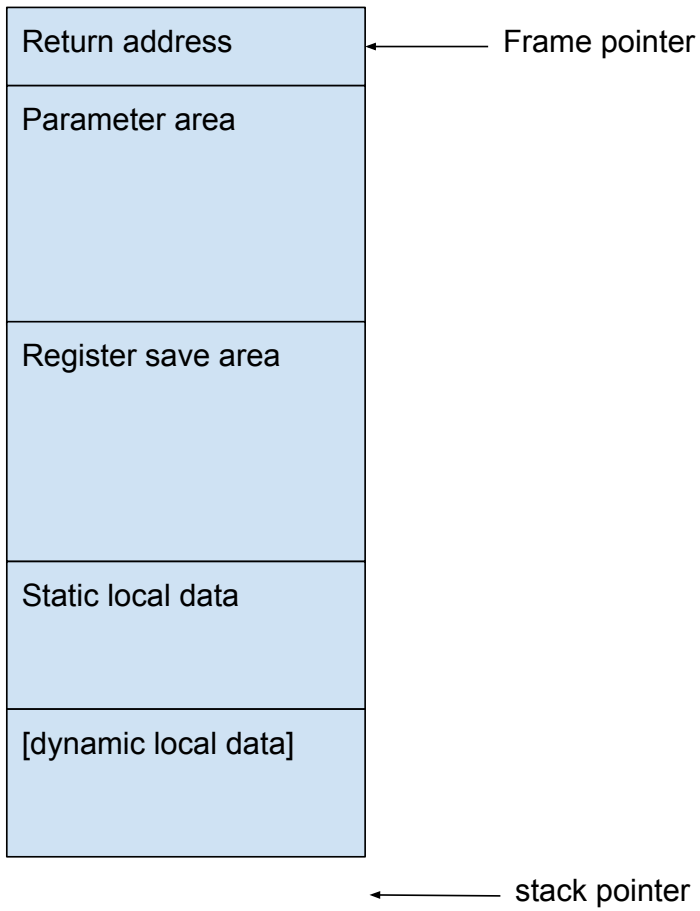
Memory map


High addresses
Heap
Dynamically allocated data
Growing upwards
Stack
One stack frame for each procedure activation
Growing downwards
Statically allocated data
Program text

Stack frames

- each procedure invocation allocates a frame that contains data local to this invocation
- as procedure invocations are properly nested, frames are allocated on a stack
  - procedure entry/call: push a new frame on the stack
  - procedure exit/return: pop its frame from the stack
- structure of stack frame is (partially) prescribed
  - by architecture
  - by run-time system
  - by API of operating system
- size of frame limited by addressing modes

Example stack frame layout



Intermediate machines

Stack machine

- intermediate results on the stack
- all operations operate on the stack
- uniform data size required
- example: JVM

Register machine

- intermediate results in (unlimited number of virtual) registers
- register operations (2-address or 3-address)
- register allocation
- data size may be different

Data allocation (JVM)

- word size 32 bits
- most data items represented in one word
  - one entry on the stack
  - one slot in local variables, object fields
- except `long` and `double`: two words
  - two adjacent entries on the stack
  - two adjacent slots in local variables and object fields
- object: reference to sequence of fields
- objects aligned at 8 bytes
- special objects: arrays
  - `byte[]` reference to contiguous sequence of bytes
  - `short[]` reference to contiguous sequence of 16 bit halfwords
  - etc . . .

Data allocation (C)

- each type has size and alignment constraint
- variables (global, local) and struct elements are packed
- Example: size 4 bytes, alignment 4

```
struct {
  char c;  // 1 byte, offset 0
  char d;  // 1 byte, offset 1
  short e; // 2 byte, offset 2
}
```

- **A struct is always aligned according to the largest alignment requirement of a member**
- stack alignment (Linux) 16 bytes

Data allocation (2)

Example: size 24 bytes, alignment 8

```
struct {
  char c;    // 1 byte, offset 0
             // compiler inserts 7 bytes padding
  double d; // 8 byte, offset 8
  char e;    // 1 byte, offset 16
             // 7 bytes padding
}
```

Wastes space; C compiler **not allowed** to reorder fields: better

```
struct {     // size 16 bytes
  char c;    // 1 byte, offset 0
  char e;    // 1 byte, offset 1
             // compiler inserts 6 bytes padding
  double d; // 8 byte, offset 8
}
```

Arrays in C

- alignment inherited from base type (type of elements)
- contiguous memory with adjacent base type values

```
struct {
  char a[3];  // 3 bytes, offset 0
  char b;     // 1 byte, offset 3
  short c[3]; // 3*2 bytes, offset 4
              // 2 bytes padding
  int d;      // 4 bytes, offset 12
}
```

Bitfields in C

Consider this (taken from Eric S. Raymond's http://www.catb.org/esr/structure-packing/)

```
struct foo6 {
    short s;
    char c;
    int flip:1;    // 1 bit
    int nybble:4; // 4 bits
    int septet:7; // 7 bits
};
```

However, bitfields must not cross the word boundaries of the underlying machine architecture.

Bitfields in C for 32-bit architecture

```
struct foo6 {
    short s;          // 2 bytes - 16 bits
    char c;           // 1 byte  -  8 bits
    int flip:1;       // 1 bit
    int nybble:4;     // 4 bits
    int __pad1:3;     // 3 bits  - padding to 32 bits
    int septet:7;     // 7 bits
    int __pad2:25;    // 25 bits - padding to 32 bits
};
```

Unions in C

- unions introduce aliasing
- ```
  union {
    int d;     // 4 bytes, offset 0
    char f;    // 1 byte,  offset 0
  }
  ```
- `d` and `f` share the same memory cells
- changing `d` changes `f` and vice versa
- size and alignment is maximum of components
- structs and unions can be nested arbitrarily
  ```

Dynamic memory (C)

- `malloc (n)` returns base address of free memory of size *n*, satisfying all alignment constraints (i.e., 16 byte)
- `sizeof type` number of bytes, computed at compile time
- programmer responsible for consistent use