5. **Storage**

5.6. **Run time type information**

---

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing

---

**5. Storage**

5.1. Storage models
5.2. Arrays
5.3. Variables' life time
5.4. Value vs. reference semantics
5.5. Automatic memory management
5.6. **Run time type information**
The challenge of deep clone

“Algorithm” for Deep Clone:
- Start from current value.
- Traverse the network of values accessible from it.
- Duplicate this network

How should we “traverse” the network?

Network Traversal: breadth- (or depth-) first search

In Each Value we Visit:
- Mark the value as “visited”
- Proceed to all values it references
But, there is a catch...

The Challenge

Network Traversal: breadth- (or depth-) first search

In Each Value we Visit:
- Mark the value as “visited”
- Proceed to all values it references

When we reach a value, we do not what’s in it!

Character?  Reference?  Ends here?

0001001  0110001  00001110  11110100  01011111  ...  10100111  10010111
Memory = bits & bytes!

To understand the difficulty better, we need to take a second look at:

- Bits
- Bytes
- Values
- Types
- Memory representation of values
- The interpretation of memory representation
Example: different interpretations of a single byte

```
#include <stdlib.h>
#include <stdio.h>

main() {
    const void *p = malloc(1);
    *(unsigned char *)p = 0b1001011;
    printf("As integer: '%d' \n",*(char *)p);
    printf("As character: '%c' \n",*(char *)p);
}
```

As integer: '75'
As character: 'K'
Example: different interpretations of a 16 bits word

```c
#include <stdlib.h>
#include <stdio.h>

main() {
    const void *p = malloc(2);
    *(unsigned short *)p = 0
        b1101011100001001;
    printf("As signed integer: '%d'\n",
        *(signed short *)p);
    printf("As unsigned integer: '%d'\n",
        *(unsigned short *)p);
    printf("As an array: '(%d,%d)'\n",
        0[(char *)p],
        1[(char *)p]
    );
}
```

As signed integer: '-10487'
As unsigned integer: '55049'
As an array: '(9,-41)'
Example: different interpretations of 32 bits word

// A more civilized way to name integer values:
enum {
    // How many bits for index into pool:
    LG2_POOLSIZE = 14,
    // How many bits for storing car/cdr kind:
    KIND_SIZE = 2
};
enum kind { NIL, ATOM, STRING, INTEGER};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
};
The layout of a C structure

<table>
<thead>
<tr>
<th>1001</th>
<th>0110</th>
<th>0000</th>
<th>1110</th>
<th>1011</th>
<th>0001</th>
<th>0110</th>
<th>1001</th>
<th>0010</th>
<th>0011</th>
<th>1011</th>
<th>1000</th>
<th>1110</th>
<th>1000</th>
<th>0101</th>
<th>0010</th>
<th>0110</th>
<th>0001</th>
<th>0011</th>
<th>1110</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>0001</td>
<td>1110</td>
<td>0100</td>
<td>1011</td>
<td>0001</td>
<td>1110</td>
<td>0100</td>
<td>1011</td>
<td>0001</td>
<td>1110</td>
<td>0100</td>
<td>1011</td>
<td>0001</td>
<td>1110</td>
<td>0100</td>
<td>1011</td>
<td>0001</td>
<td>1110</td>
<td>0100</td>
<td></td>
</tr>
</tbody>
</table>

The same memory block could be interpreted in many different ways.
The layout of a C structure

The same memory block could be interpreted in many different ways. Here is a 16 bytes block,
The layout of a C structure

The same memory block could be interpreted in many different ways. Here is a 16 bytes block, which can be interpreted as `struct T0`,

```c
struct T0 {
    float x;
    struct T1 *p;
    char s[4];
    struct T2 *q;
};
```
The layout of a C structure

The same memory block could be interpreted in many different ways. Here is a 16 bytes block, which can be interpreted as struct T0, as struct T1,

```
struct T0 {
    float x;
    struct T1 *p;
    char s[4];
    struct T2 *q;
};
```

```
struct T1 {
    char s[4];
    float x;
    struct T2 *q;
    struct T0 *p;
};
```
The layout of a C structure

The same memory block could be interpreted in many different ways. Here is a 16 bytes block, which can be interpreted as `struct T0`, as `struct T1`, or, as `struct T2`.

```c
struct T0 {
    float x;
    struct T1 *p;
    char s[4];
    struct T2 *q;
};

struct T1 {
    char s[4];
    float x;
    struct T2 *q;
    struct T0 *p;
};

struct T2 {
    struct T0 *p;
    char s[4];
    struct T1 *q;
    float x;
};
```
Summary: the “meaning” of bits and bytes

A value is represented in memory as a *sequence* of bits and bytes.

Components:

- Integers
- Floating point values
- Characters
- References
- Arrays
- Sets in bit mask representation.
- etc.

Deciphering a Value

- The values’ type is the *key*
- It gives meaning to the bit representation.

Information provided by type:

- Value’s length
- Partitioning into sections
- Appropriate way of interpreting each section
A step in a BFS/DFS tour

- Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

- Further, although we can examine the bits and bytes, we cannot know what their values mean!

- Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

- With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

- Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

- Further, although we can examine the bits and bytes, we cannot know what their values mean!

- Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

- With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second.
Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!
A step in a BFS/DFS tour

- Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

- Further, although we can examine the bits and bytes, we cannot know what their values mean!

- Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

- With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values' graph, and we follow a reference reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second...
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priori, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

- Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

- Further, although we can examine the bits and bytes, we cannot know what their values mean!

- Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

- With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
Suppose we are the midst of a BFS (or BFS) traversal in the values’ graph, and we follow a reference reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second one.
### A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values' graph, and we follow a reference, reaching a memory block. Unfortunately, a-priori we do not know how long the block is. Further, although we can examine the bits and bytes, we cannot know what their values mean!

**Supposing** that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

![16 bytes memory block](image)
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priori, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type \( T_0 \), then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

Suppose we are midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a priori, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priori, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type \( T_0 \), then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
A step in a BFS/DFS tour

Suppose we are the midst of a DFS (or BFS) traversal in the values’ graph, and we follow a reference, reaching a memory block. Unfortunately, a-priory, we do not know how long the block is.

Further, although we can examine the bits and bytes, we cannot know what their values mean!

Supposing that we know that the value is of type $T_0$, then, we know how long the memory block is, and that it has four words, of four bytes each, as well as the exact type of each of these words.

With this information, we can continue the traversal, along the first reference found in this memory block, and then, along the second such reference.
But, the visited block could be of any type!
But, the visited block could be of any type!
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

DFS/BFS

- \( T_0: \text{float} \)
- \( T_0: \text{struct T1 *} \)
- \( T_0: \text{char[4]} \)
- \( T_0: \text{struct T2 *} \)
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

DFS/BFS

```
T0: float  T0: struct T1 *  T0: char[4]  T0: struct T2 *
```

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```
But, the visited block could be of any type!

struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };

struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; }
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```c
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```
But, the visited block could be of any type!

```c
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

```c
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; }
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; }
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; }
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```

```
struct T2 { struct T0 *p; char s[4]; struct T1 *q; float x; };
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```

```
struct T2 { struct T0 *p; char s[4]; struct T1 *q; float x; };
```
But, the visited block could be of any type!

```
struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };
```

```
struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };
```

```
struct T2 { struct T0 *p; char s[4]; struct T1 *q; float x; };
```
But, the visited block could be of any type!

struct T0 { float x; struct T1 *p; char s[4]; struct T2 *q; };

struct T1 { char s[4]; float x; struct T2 *q; struct T0 *p; };

struct T2 { struct T0 *p; char s[4]; struct T1 *q; float x; };
## Interpreting values

### Static typing

The compiler knows the “deciphering key”, and it generates code based on this information.

### Dynamic typing

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

RTTI field may **contain all the type information**

more commonly, RTTI is a reference to a “type descriptor” shared by all values of the type.

<table>
<thead>
<tr>
<th>RTTI</th>
<th>( v_1 \in T_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1015</td>
<td>&quot;gaga&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTTI</th>
<th>( v_2 \in T_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2324.05</td>
<td>&quot;barr&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTTI</th>
<th>( v_3 \in T_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3432.2</td>
<td>&quot;vazz&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTTI</th>
<th>( v_4 \in T_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4540</td>
<td>&quot;gaga&quot;</td>
</tr>
</tbody>
</table>
Interpreting values

**Static typing**

The compiler knows the “deciphering key”, and it generates code based on this information.

**Dynamic typing**

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

*RTTI field may contain all the type information*

more commonly, *RTTI is a reference to a “type descriptor” shared by all values of the type.*
Interpreting values

**Static typing**

The compiler knows the “deciphering key”, and it generates code based on this information.

**Dynamic typing**

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

RTTI field may contain all the type information

more commonly, RTTI is a reference to a “type descriptor” shared by all values of the type.

\[ v_1 \in T_0 \]

\[ RTTI \]

\[ v_2 \in T_0 \]

\[ RTTI \]

\[ v_3 \in T_0 \]

\[ RTTI \]

\[ v_4 \in T_0 \]

\[ RTTI \]
Interpreting values

**Static typing**

The compiler knows the “deciphering key”, and it generates code based on this information.

**Dynamic typing**

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

**RTTI field may contain all the type information**

more commonly, **RTTI is a reference to a “type descriptor” shared by all values of the type.**
Interpreting values

### Static typing
The compiler knows the “deciphering key”, and it generates code based on this information.

### Dynamic typing
A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

*RTTI field may contain all the type information*

more commonly,  
**RTTI is a reference**  
to a “type descriptor” shared by all values of the type.

---

Floating Point Number  
Reference (to type $T_1$)  
4 Characters  
Reference (to type $T_2$)  

RTTI for type $T_0$
Interpreting values

**Static typing**

The compiler knows the “deciphering key”, and it generates code based on this information.

**Dynamic typing**

A “deciphering key” is attached to each value; the run-time system decodes the key.

The “deciphering key” in nothing but the type!

RTTI field may contain all the type information

more commonly, RTTI is a reference to a “type descriptor” shared by all values of the type.
Designing an algorithm for traversing values

Can we use static type information?

*No!!!*

- The network of objects typically contains values of very many distinct types
- The traversal algorithm should know
  - the type of each visited value,
  - the types of each of the values it references
- It is impractical to generate a different traversal algorithm for each input program as per the different that occur in it.
RTTI is the answer!

**Definition (Run-Time Type Information)**

Run-time type information *(or RTTI for short)* is a tag attached to each value, which specifies its type.

**Application of RTTI in different kinds of PLs:**

**Statically Typed**
- Deep cloning,
- Garbage collection, *and*,
- Serialization.

**Dynamically Typed**
- Deep Cloning,
- Garbage Collection,
- Serialization, *and*,
- Run time type checks.
C, C++, & RTTI

- As a result of the “no hidden cost” language principle, C does not and cannot have RTTI.
- As a result, C cannot have general purpose GC, serialization, cloning or any deep operations.
- Due to the “C-compatibility at almost all costs” language principle, C++ does not and cannot have RTTI.
- As a result, C++ cannot have general purpose GC, serialization, cloning or any deep operations.
- C++ has a limited form of RTTI for the implementation of virtual functions.
- More on these mysterious “vptr” and “vtbl” in our OOP course.
Use of RTTI in the implementation of different PLs

- Consider a variable `today` which references an object with (say) three fields: `year`, `month`, `year`.
- How is `today.day=35` being implemented?

| prog. lang. | C | Java | JavaScript |
|-------------|----------------|------------------------|
| syntax      | `today->day=35` | `today.day=35` | `today.day=35` |
| static typing | ✓ | ✓ | ✗ |
| dynamic typing | ✗ | ✓ | ✓ |
| RTTI | ✗ | ✓ | ✓ |
| type punning | ✓ | ✗ | ✗ |

**Implementation**

- 1. dereference `today`
- 2. advance by `off(day)`
- 3. update field

- 1. dereference `today`
- 2. ignore RTTI
- 3. advance by `off(day)`
- 4. update field

- 1. dereference `today`
- 2. examine RTTI
- 3. determine `off(day)`
- 4. advance by `off(day)`
- 5. update field
Comments on use of RTTI in PLs

- When and how is \texttt{off(day)}, the function determining the field offset, determined?

- In statically typed languages:
  - at compile time
  - from the static type of \texttt{today}.

- In statically typed languages:
  - at runtime
  - from the RTTI of “\texttt{*today}”

- In \texttt{C}, the \textbf{actual} type of \texttt{*today} could be anything (due to type punning).

- In \texttt{Java}, the \textbf{actual} type of the object that \texttt{today} refers to, can be any class that \textbf{extends} of \texttt{Date}.