プログラミング言語 1
安全性と型
Programming Languages 1
Safety and Types

田浦

Table of Contents

プログラミング言語における「型」

- データ型 (data type) または単に型 (type) という
- ≈ データの「種類」
- C の例
  - プリミティブ型: 整数 (char, short, int, . . .)，浮動小数点数 (float, double, . . .), . . .
  - 複合型 (既存の型を組み合わせて作る型): ポインタ，配列，構造体，共用体，. . .

Types in Programming Languages

- "Data type" or more simply "types"
- $\approx$ *Kinds* of data
- Examples in C
  - Primitive types: integral types (char, short, int, ...), floating point number types (float, double, ...), ...
  - Compound (derived) types (types built from existing types): pointers, arrays, structures, unions, ...

型の役割

1. 抽象化，隠蔽
2. 性能向上
3. エラーの検出

Roles of Types

1. Abstraction
2. Performance improvement
3. Error detection

型の役割 (1) 抽象化，隠蔽

- 複数の値をまとめてひとつの値と見なせる (一つの変数で「持ち歩ける」) ようにする

```
1 typedef struct  {
2   float x; float y;
3 } vec2;
4 vec2 add(vec2 a, vec2 b);
```

- 型の「中身」をわかっていなくても使える

```
1 FILE * fp = fopen("foo.txt", "rb");
2 fread(buf, n, 1, fp);
```

⇒ 理解・再利用しやすいプログラム (部品)，変更しやすいプログラムの構造を作ることができる

Roles of Types (1) Abstractions

- make compound values look like a single value (a single variable can "carry" them)

```
1 typedef struct  {
2   float x; float y;
3 } vec2;
4 vec2 add(vec2 a, vec2 b);
```

- make it possible to use/pass a value without knowing its constituents

```
1 FILE * fp = fopen("foo.txt", "rb");
2 fread(buf, n, 1, fp);
```

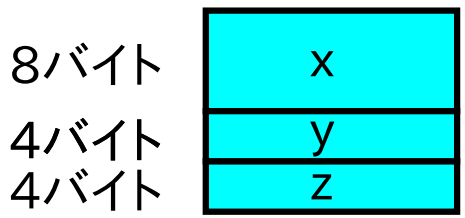
$\Rightarrow$ make programs (components) easier to understand and more reusable

型の役割 (2) 効率の向上

- コンパイル時に型がわかれば，ムダのないデータレイアウト，レジスタの利用，命令の生成が可能

```
1  typedef struct {
2    double x; int y; int z;
3  } DI;
4
5  double f(DI * di) {
6    return di->x + di->y + di->z;
7  }
```

⇒




```
1  cvtsi2sd 8(%rdi),%xmm0
2  addsd (%rdi),%xmm0
3  cvtsi2sd 12(%rdi),%xmm1
4  addsd    %xmm1,%xmm0
```

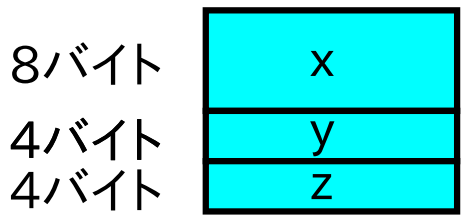
- x は di, y は di+8, z は di+12 バイトにある
- y, z は int なので double に変換する
- などは型の情報から推論する

Roles of Types (2) Performance Improvement

- knowing types of expressions at compile time lead to a more compact data layout, a better register usage and a more efficient sequence of instructions

```
1  typedef struct {
2    double x; int y; int z;
3  } DI;
4
5  double f(DI * di) {
6    return di->x + di->y + di->z;
7  }
```

⇒




```
1  cvtsi2sd 8(%rdi),%xmm0
2  addsd (%rdi),%xmm0
3  cvtsi2sd 12(%rdi),%xmm1
4  addsd    %xmm1,%xmm0
```

- the following is inferred from types:
- x is address designated di, y at (di+8), z at (di+12)
- y and z are integers, so need to be converted to double

型の役割 (3) エラーの検出

値の「取り違え」を防ぐ (「型エラー (type error)」)

- 変数代入時の「取り違え」

```
1 char * s;
2 s = 195;  /* char* <- 整数 */
3 putchar(s[0]);    /* 整数の [0] って? */
```

- 関数呼び出し時の「取り違え」

```
1 fprintf("x=%d\n", x); /* FILE*にchar*を渡すことに */
```

Roles of Types (3) Error Detection

prevent "misusage" of values (type error)

- assignments into wrong variables

```
1 char * s;
2 s = 195;  /* char* <- int */
3 putchar(s[0]);    /* what is ``[0]'' of an int? */
```

- passing wrong values to functions

```
1 fprintf("x=%d\n", x); /* passing char* to FILE* */
```

C 言語の安全でない仕様 (一部) 1/2

- 異なる型へのポインタ間の代入を許す

```
1  A * a = ...;
2  B * b = a;        /* さすがに不許可 */
3  B * b = (B *)a; /* 許可 明示的キャスト (何してるかわかってるはず) */
4  void * v = a;    /* 許可 void* への代入は許す */
5  b = v;           /* 許可 void* の代入は許す */
```

- 共用体

```
1  union {
2    char * p;
3    long x;
4  } u;
5  u.x = 10;
6  u.p[0] = ...;    /* 許可 */
```

Some Unsafe Features in the C Language Specification 1/2

- Allow assignments between different pointer types

```
1 A * a = ...;
2 B * b = a;       /* disallow */
3 B * b = (B *)a; /* allow: with explicit cast (you should know know what
                     you are doing) */
4 void * v = a;    /* allow: assignments into void* variables */
5 b = v;           /* allow: assignments of void* values */
```

- Unions

```
1 union {
2   char * p;
3   long x;
4 } u;
5 u.x = 10;
6 u.p[0] = ...;    /* allow */
```

C 言語の安全でない仕様 (一部) 2/2

- 変数の初期値が不定

```
1  A * a;              /* a の値は「不定」 */
2  a[0] = ...;         /* どこをアクセスするかは運任せ */
```

- return し忘れてもお咎め無し

```
1  char * f() { } /* return し忘れ */
2  int main() {
3    char * a = f(); /* OK. 一見まともだが,a の値は結局不定 */
4    a[0] = ...;
5  }
```

Some Unsafe Features in the C Language Specification 2/2

- uninitialized variables get unknown values

```
1  A * a;           /* a has an unpredictable value */
2  a[0] = ...;      /* who knows where it accesses */
```

- forgetting to return a value may not be catched

```
1  char * f() { } /* forget to return a value */
2  int main() {
3    char * a = f(); /* allowed. it looks benign but a still gets an
                        unpredictable value */
4    a[0] = ...;
5  }
```

型と安全性の関係を正しく理解する

誤解

1. ポインタ ← 非ポインタ を禁止すればいいじゃなイカ?
2. 「ポインタ型」をなくせばいいじゃなイカ?

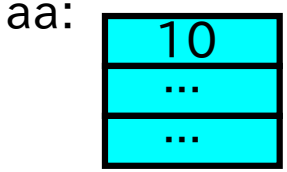
Understanding The Relationshp between Types and Safety

Misconceptions:

1. ban pointer $\leftarrow$ non-pointer assignments and we are done?
2. get rid of pointers altogether?

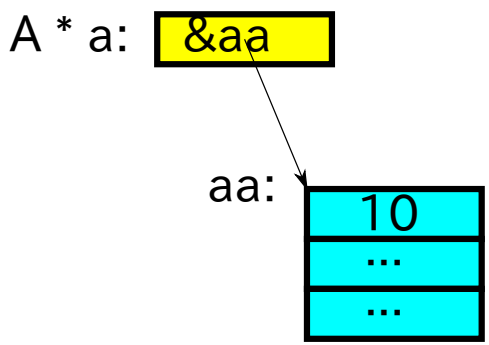
誤解: ポインタ←非ポインタ を禁止すればいいじゃないカ

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* 10番地にstore */
}
```



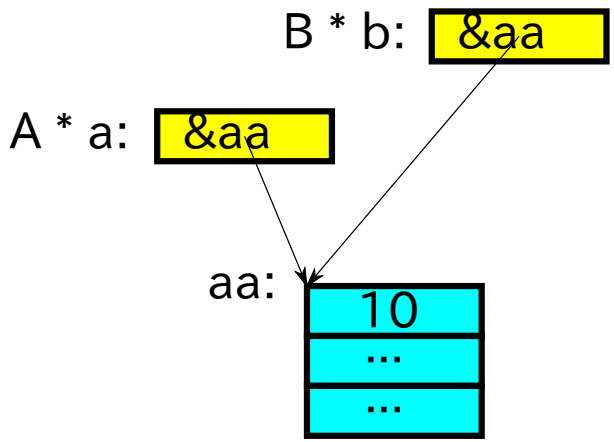
誤解: ポインタ←非ポインタ を禁止すればいいじゃないカ

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* 10番地にstore */
}
```



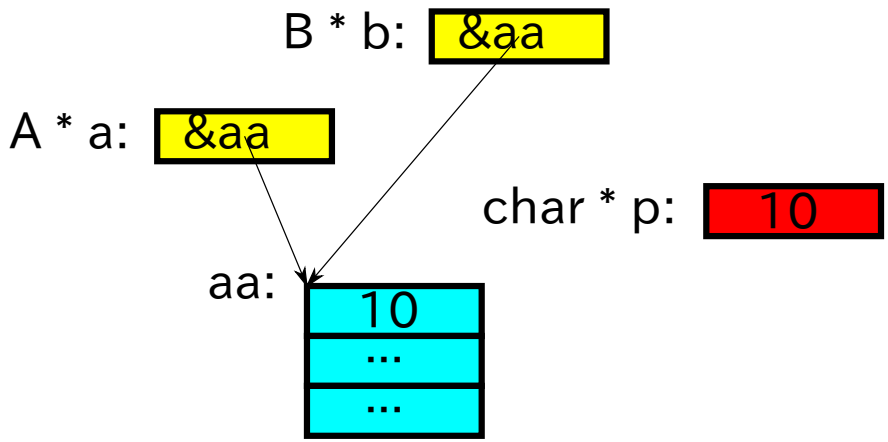
誤解: ポインタ←非ポインタ を禁止すればいいじゃないカ

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* 10番地にstore */
}
```



誤解: ポインタ←非ポインタ を禁止すればいいじゃないカ

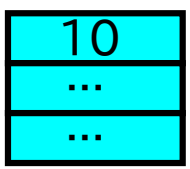
```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* 10番地にstore */
}
```



Misconception: ban pointer ← non-pointer assignments and we are done

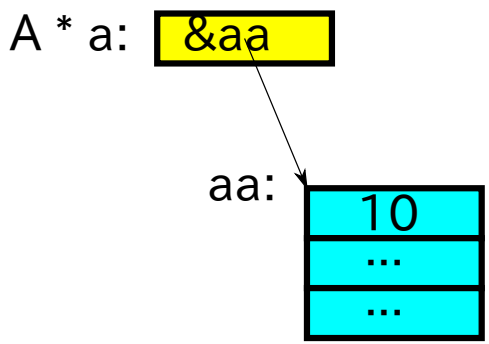
```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* you store into address
          10 */
}
```

aa:



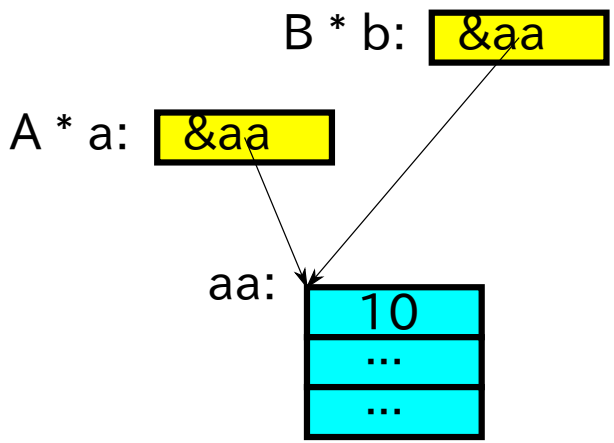
Misconception: ban pointer ← non-pointer assignments and we are done

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* you store into address
          10 */
}
```



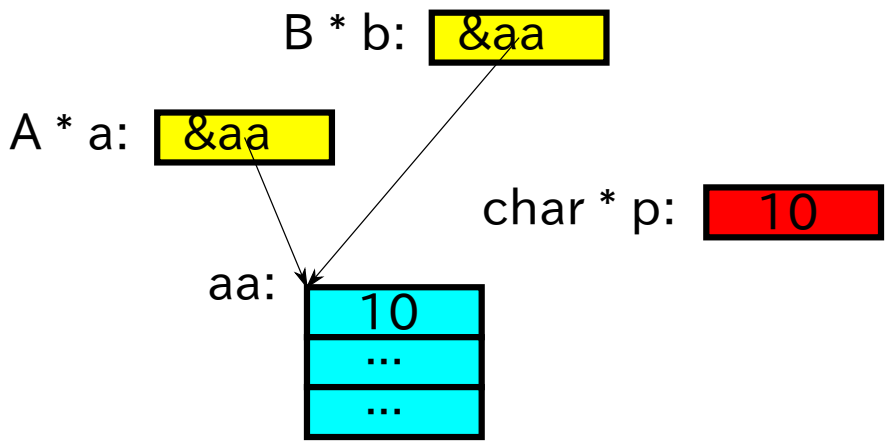
Misconception: ban pointer ← non-pointer assignments and we are done

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* you store into address
          10 */
}
```



Misconception: ban pointer ← non-pointer assignments and we are done

```
typedef struct {
  long x;
  ...
} A;
typedef struct {
  char * p;
  ...
} B;
int main() {
  A aa = { 10, ... };
  A * a = &aa;
  /* B* <- A* */
  B * b = a;
  char * p = b->p;
  p[0] = 20; /* you store into address
          10 */
}
```



この例が示すもの

- 「A へのポインタ ← B へのポインタ」を許すだけでも，結局ポインタ型に非ポインタ (10) が代入されることになる
- ⇒ 異なる型の間の代入を許すだけで，(必ずではないが) 多くの場合，巡り巡って変なアドレスをアクセスするプログラムになる
- 注: 先の例の，{ 10，...} を変えるだけで，どこにでもアクセスしうるプログラムになる
- Segmentation Fault ではすまない (他のデータを破壊する) 場合もある

What This Example Tells You

- allowing "pointer to A ← pointer to B" effectively allows you to "pointer ← int"
- ⇒ once allow assignments between different pointer types, you allow programs to access invalid addresses
- Remark: just change { `10, ...`} part in the example and it can access *any* address
- you will get Segmentation Fault or worse (corrupt your data)

誤解: 「ポインタ型」をなくせばいいじゃないカ

- たしかに,「C から」ポインタ型をなくせば (配列の添字溢れ以外の) 危ないエラーは起きない
- それは実用的な言語ではないし，C 以外の「ポインタ型を持たない」言語とも似ていない
- 以下のようなデータは (それを「ポインタ型」と呼ぶかによらず)，ポインタなしには実現できない
  - 文字列，配列，オブジェクト，etc.
  - これらはおよそどんな言語にもある
  - ポインタ型とそうでない型の区別があるかは問題ではない

Misconception: Get Rid of Pointers!

- Sure, if you remove pointers from C, none of the above errors will ever happen
- But it's neither a practical language nor like any other "pointer-free" languages
- Following data cannot be realized without internally using pointers (whether the language calls it "pointers" or not),
  - strings, arrays, objects, etc.
  - virtually all languages have them
  - it doesn't matter whether the language has explicit "pointer" types

実現のためにポインタが必要なデータ

- 配列 (a[i] のように定数でない添字でアクセス可能)
- 更新可能なデータ

```
1 a.x = b.x = 0;
2 Foo c = (random() ? a : b);
3 c.x = 1;
4 System.out.println(a.x);
```

- 再帰的なデータ構造 (Node の中の Node はポインタなしでは困難)

```
1 class Node {
2   Node left;
3   Node right;
4 };
```

標語的には，C 以外の多くの言語では,「すべてがポインタ」

Data types that require pointers

- arrays (allows data to be accessed through non-constant index, as in `a[`i`]`, which requires address arithmetic)
- mutable data

```
1 a.x = b.x = 0;
2 Foo c = (random() ? a : b);
3 c.x = 1;
4 System.out.println(a.x);
```

- recursive data types ("Node insdide Node" is difficult to implement without making them pointers)

```
1 class Node {
2   Node left;
3   Node right;
4 };
```

In other languages, "everything is a pointer"

「ポインタ型」がなくても危険

Java の例:

```
1 class Foo { long p; };
2 class Bar { String s; };
3 Bar b = new Foo();  /* Bar <- Foo */
4 b.s[0];
```

- 変数 b には, ポインタが保持されている
- 3 行目の代入 (異なる型の間での代入) を許せば，C と同様の危険が生ずる

Unsafety arises without "pointers"

Example in a (hypothetical) Java-like language:

```
1  class Foo { long p; };
2  class Bar { String s; };
3  Bar b = new Foo();  /* Bar <- Foo. let's say we allow this */
4  b.s[0];
```

- variable b holds a pointer
- if you allow the assignment at line 3, it is as unsafe as C

では安全な言語をどう作るか?

- その前に「安全な言語」って何?
- 「この言語で書けばセキュリティホールのあるプログラムは決してできませんよ」は素晴らしいが，目標がやや野心的 (無理っぽ) すぎる
- ここでの「安全」の定義はもっと控えめなもの (memory safety)
  - ≈ わけのわからない挙動をしない
    - ★ 例: データはそのデータを変更することでしか書き換わらない
    - ★ C の場合: `a[i] = x` を実行したら変数 `y` が壊され得る
  - ≈ 早い話,「変な場所をアクセスしない」
- 以降「安全」はこの控えめな意味で使う

So how you make languages safe?

- what are "safe languages" anyway?
- it would be ideal but too ambitious to have a language that does not allow any programs with security holes
- we must be satisfied with a more modest definition of "safety" (memory safety)
  - ≈ never exhibits a "strange" (≈ undefined) behavior
    - ex: data will never be "corrupted"; data will be modified only through "valid" pointers to it
    - in C: it is possible that `a[i] = x` can currupt another, totally independent varible, `y`
  - ≈ you never access "invalid/strange locations"
- we always use "safety" in this modest sense

安全な言語の要件

「ポインタの dereference」の妥当性を保証する

- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実行前・コンパイル時) 型検査; static type check
- 自動メモリ管理
  - すでに解放 (正確には再利用) されている領域へアクセスしない $\Rightarrow$ これからもアクセスされる可能性がある領域を再利用しない

以下は「型安全」の部分に限って議論する

Requirements for Safe Languages

Ensure the validity of "pointer derefernces"

- ensure arrays are not accessed with invalid indexes (indexes that are out of bounds)
- type safety (e.g., ensure the program does not confuse data types; it never access data of type X as a different type Y)
  - Dynamic (runtime) type checks
  - Static (pre-execution or compile-time) type checks
- automatic memory management
  - ensure the program does not access the region that are already released (reused for other purposes, to be precise) $\Rightarrow$ do not wrongly reuse the region that will be used in future

The rest of the slides will focus on the "type safety"

Contents

型エラーに対するアプローチによる言語の分類

	実行前型検査	実行時型検査	実例
危険な言語	抜け穴だらけ	しない	C, C++, Fortran, ...
動的型付き言語 (†)	しない	する	Python, Ruby, JavaScript, ...
静的型付き言語 (††)	(ほぼ) 完全	(ほぼ) 不要	Java, Eiffel, ML, OCaml, ...

- ∗ 危険
- † dynamically typed languages. インタプリタ型言語はほぼ必然的にこの方法になる
- †† statically typed languages. ここに分類された言語には，実行時型検査が本当に不要 (実行前検査を通れば実行時に型エラーが起きないもの), ところどころ実行時型検査を行うもの，言語仕様の間違いにより実は危険なもの，などがある

Different Approaches to Detecting Type Errors

	Compile-Time Checks	Runtime Chcecks	I
Unsafe	done albeit with loopholes	never done	C
Dynamically typed (†)	never done	done	P J
Statically typed (††)	(mostly) complete	(mostly) unnecessary	J M

- † many interpreting or scripting languages fall in this category
- †† statically typed languages vary in the level of type safety guarantees they provide; some guarantee safety without any runtime checks; others guarantee safety with the help of runtime checks; others simply fail to provide type safety (despite static type checks)

Contents

動的 (実行時) 型検査

- 危険な操作に対して意味のある「実行時エラー」を起こす.
  例 (Python):

```
>>> def f():
...     s = 20000
...     return s[0]
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f
TypeError: 'int' object has no attribute '__getitem__'
>>> def g():
...     s = "abc"
...     return s.x
>>> g()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in g
AttributeError: 'str' object has no attribute 'x'
```

- f, g の「定義」自身はエラーではないことに注意 (実行して初めてエラーになる)

Dynamic (Runtime) Type Checks

- Raise a (meaningful) error at runtime, upon an unsafe operation. Ex (Python):

```
>>> def f():
...     s = 20000
...     return s[0]
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f
TypeError: 'int' object has no attribute '__getitem__'
>>> def g():
...     s = "abc"
...     return s.x
>>> g()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in g
AttributeError: 'str' object has no attribute 'x'
```

- note that the "definition" of `f`, `g` succees without errors (an error occurs only when you execute them)

動的 (実行時) 型検査に必要なもの

演算の妥当性を「実行時に」判定できるような，データの「共通表現」

- 演算: フィールド参照 (`.x`)，添字での参照 (`[i]`)，プリミティブ演算 (足し算など)

そのために:

- 「即値 (1 ワードで表現できる値．整数など)」と,「box 値 (ポインタ+メモリ上のデータ)」の区別ができる
- box 値のデータからデータ型 (配列, レコード，文字列，etc.), レコードであればどんなフィールドがあるかなどが判定できる

How to implement Dynamic (Runtime) Type Checks

a "common data format" with which the system can judge the validity of operations at runtime

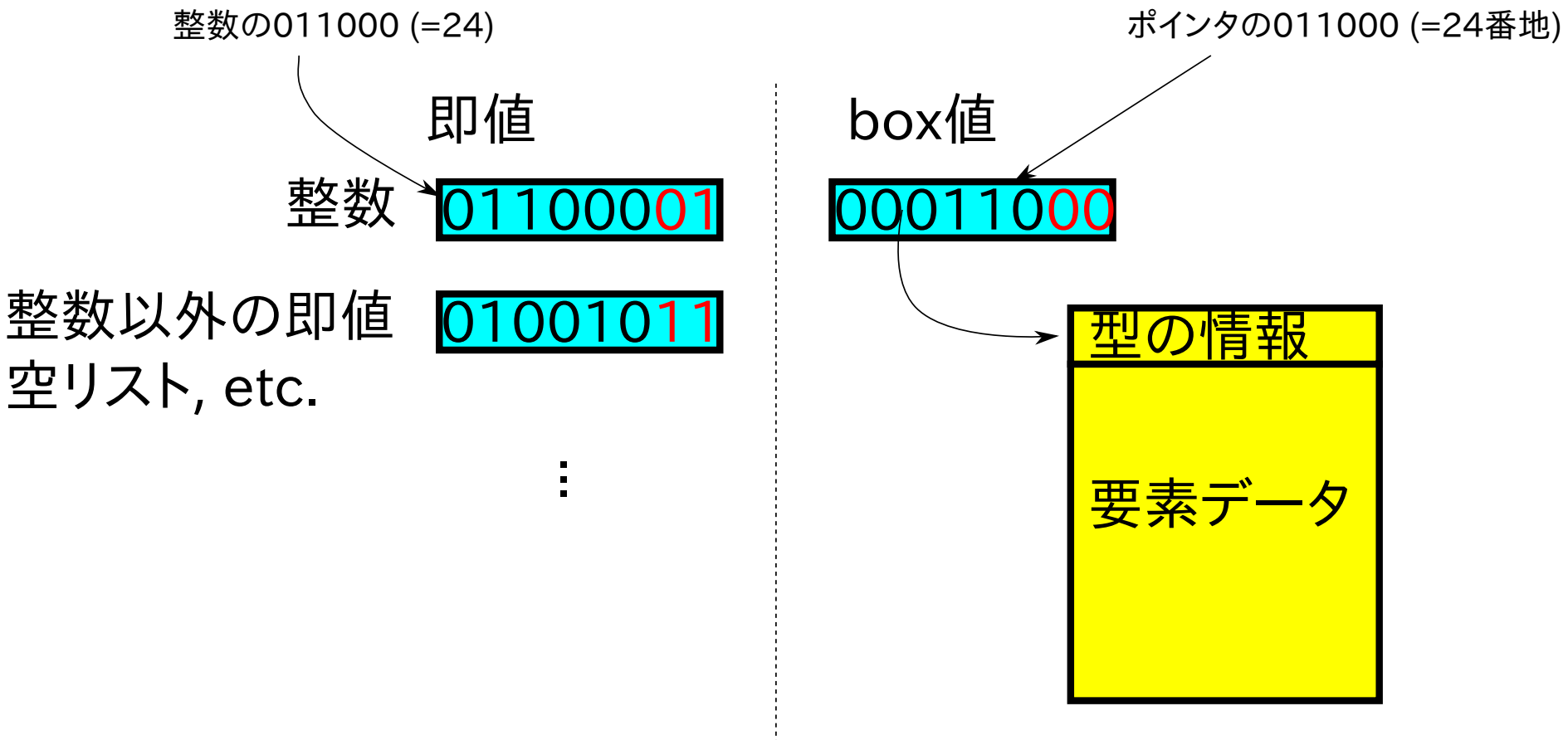
- operations: field access (`.x`), index access (`[i]`), arithmetics (addition etc.)

to this end, we need to

- distinguish "an immediate value (a value reresented in a single machine word (an integer etc.)" from "a boxed value (a represented as pointer to values put in memory)"
- know the type of any boxed value (arrays, records, strings, etc.) from its representation at runtime

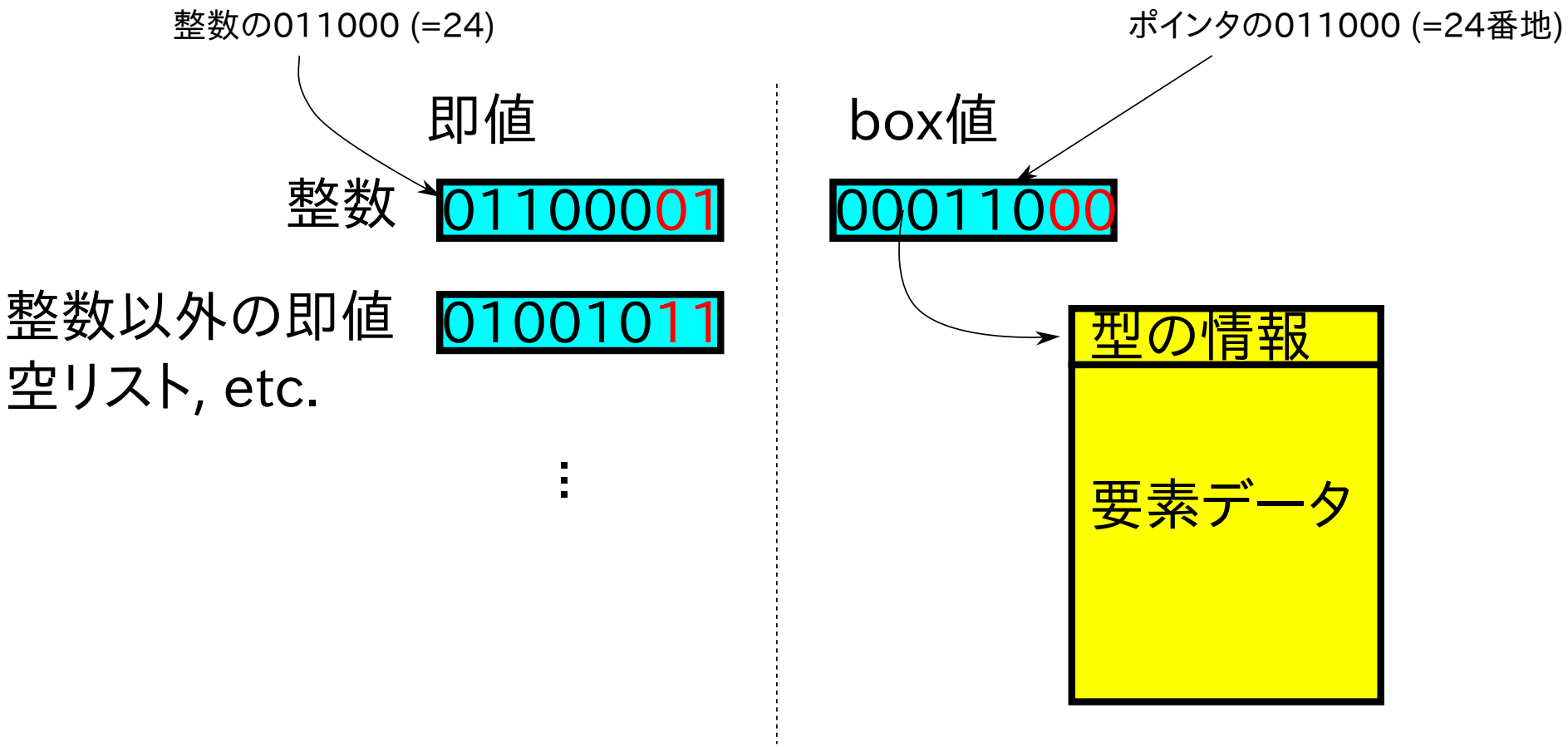
データの共通表現の例

- 即値 (下 2 bit $\neq$ 00)
  - 整数
  - その他のよく出る即値 (空文字列，空リスト，etc.)
- box 値 (下 2 bit = 00)



A Common Data Representation

- immdiate value (the lowest 2 bits $\neq$ 00)
  - integers
  - other common values (empty strings, empty lists, etc.)
- boxed values (the lowest 2 bits = 00)



動的 (実行時) 型検査の特徴

- エラーは実行途中で判明する．⇒ エラー原因の追求は (安全でない言語に比較して) 行い易いが，エラーを事前に防ぐという観点からはほとんど意味がない
- 効率の良い実装が困難．特に，
  - 浮動小数点: 倍精度浮動小数点数 (double; 64 bit) を表そうと思うと，最低でも 65 bit 必要 → 「即値」にしづらい → 演算のたびにメモリアクセス，割り当て
  - レコードの要素アクセス: 定数オフセットのポインタ参照では無理．ハッシュ表などの動的な探索が必要
- (特にインタプリタ実行の) 実現が容易
- 型エラーをおこさないプログラムが，静的な検査で違反となることがない (理解しやすい)

Dynamic (Runtime) Checks: Pros and Cons

- detect errors only at runtime $\Rightarrow$ it certainly makes diagnosis easier than unsafe languages, but does little to prevent errors from happening in the first place
- makes efficient implementation difficult. in particular,
  - floating point arithmetic: double-precision numbers (double; 64 bit) would require at least 65 bits $\rightarrow$ not amenable to immediate values $\rightarrow$ memory allocation + access for each operation
  - field access of a record: cannot be implemented as a memory access with a constant offset. instead requires dynamically searching a record for the field
- simplifies implementation (especially of interpreters)
- never prohibit programs that in fact do not cause type errors ($\rightarrow$ is easy to understand)

Contents

静的型検査

- 実行前に型エラーを検出する
- 理想: 実行前の型検査をパスすれば，決して実行時に型エラーは起きない
  - C はもちろんそうではないし，Java も実はそうではない．実はこれを保証できている言語は少ない
  - そのつもりで設計された言語が後にそうではないとわかったこともある (Eiffel)
  - 何が難しいのか? 異なる型の間の代入をすべて禁止すればいいだけでは?

Static Type Checks

- Detect type error prior to execution
- Idea: if you pass the pre-execution check, the program "never" encounters a type error at runtime
  - C does not have this property; nor Java. only a handful of languages has this property
  - some languages (Eiffel) were designed to achieve this goal but later turned out not to satisfy it
  - it is difficult? doesn't it amount to forbidding assignments between different types?

Contents

異なる型の間の代入をすべて禁止すればいいだけじゃなイカ

- 「安全」だけが目標ならそのとおり (例: Pascal)
- 問題: プログラムの再利用性・一般性が損なわれる
  - 特定の型用に書かれた定義 (関数・データ構造) は，たとえやっていることが文面上同じでも，他の型に適用できない

```
1 /* 配列 a から x を見つける */
2 int index(int * a, int n, int x) {
3   for (i = 0; i < n; i++) {
4     if (a[i] == x) return i;
5   }
6   return n;
7 }
```

  - このコードは，a の要素に関する制約は,「==で比べられること」だけであるにも関わらず，int 以外には適用できなくなる．

Forbit All Assignments between Diffenrent Types and Aren't We Done?

- Yes, if "safety" is the only concern (ex: Pascal)
- Issue: it hampers program reusability and generality
  - a definition (of functions or data structure) can never be reused for other data types even if it generically works for them

```
1 /* find x from array a */
2 int index(int * a, int n, int x) {
3   for (i = 0; i < n; i++) {
4     if (a[i] == x) return i;
5   }
6   return n;
7 }
```

  - this code would "generically" work on any array a, as long as its elements can be compared with ==. yet this can be applied only to array of int.

単純な静的型では不十分な場面

- 汎用的なデータ構造 (コンテナ)
  - 配列，リスト，キュー，スタック，グラフ，etc.

```
1 typedef struct { /* 可変長配列 (≈STL vector) */
2   element * a;
3   int n;
4   int sz;
5 } var_array;
```

- 汎用的なアルゴリズム
  - 整列，探索，グラフアルゴリズム，etc.

```
1 void sort(element * a, int n);
```

  - メモリコピー，メモリ管理

どの場合も,「プログラムの文面上ひとつの変数や式が，実行時には複数の型を持ちうる (多相性; polymorphism ポリモルフィズム)」ことが必要

Cases in which simple types are too inconvenient

- generic data structures (containers)
  - array, list, queue, stack, graph, etc.

```
1 typedef struct { /* variable array (≈STL vector) */
2   element * a;
3   int n;
4   int sz;
5 } var_array;
```

- generic algorithms
  - sort, search, graph algorithms, etc.

```
1 void sort(element * a, int n);
```

  - memory copy and memory management

Any of them requires polymorphism, a property that a variable or an expression in the program can have multiple types at runtime

多相性に対するアプローチ

- ジェネリックなポインタ型 (void*) + キャスト
  - C
  - 安全ではない
  - 型に関わる言語仕様は単純さを維持できる
- パラメトリック多相 (parametric polymorphism)
  - 関数型言語では基本 (→ 詳しくは OCaml の演習で)
  - 多くのオブジェクト指向言語でも採用
- 部分型多相 (subtype polymorphism, subtyping)
  - オブジェクト指向言語では基本 (→ 詳しくは Pyhon 演習後に)

Approaches toward Poloymorphism

- Generic pointers (void*) + cast
  - C
  - not safe
  - can keep the language simple
- parametric polymorphism
  - spread in functional programming languages ($\rightarrow$ OCaml exercise)
  - nowadays common in object oriented languages too
- subtype polymorphism, subtyping
  - foundation of object oriented languages ($\rightarrow$ details after Python exercise)