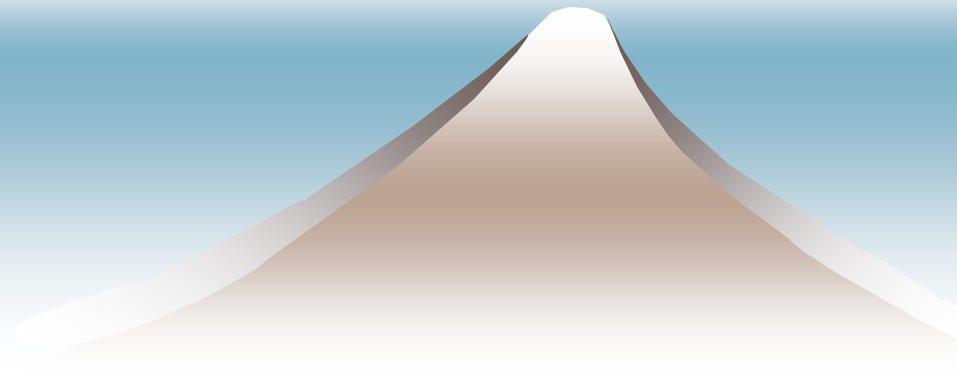
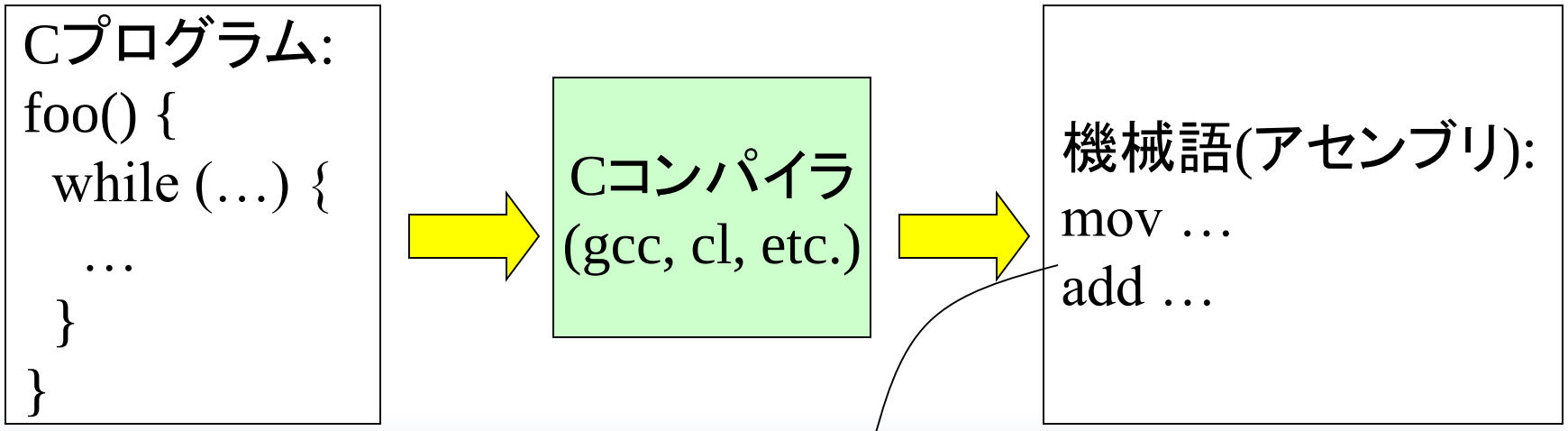
C言語と機械語

田浦



目標

- Cプログラムの実行を機械語レベルで「イメージ」できるようになる



OSの授業なのにどうして?

- OSやCPUは「言語中立」
  - OSはプログラムがどの言語で書かれていようと関係ない言葉で設計されている
    - たとえばシステムコール(API)の引数や返り値は, 整数や「アドレス」であってJavaの配列やPerlの文字列ではない
  - CPUもあくまで機械語しか知らない
    - たとえばスレッドがメモリを共有しているのはC言語の処理系がやっているのではなくCPU(とOS)が提供している

OSの授業なのにどうして?

- しかし説明の都合上「C言語」で説明することが多い
  - システムコールのAPI説明はCの関数で説明するのが慣習
  - スレッドが「メモリを共有」していることを, C言語の変数や配列が共有されていることとして説明するのが慣習
- 実際は機械語で説明すべきこと. だがそれは不必要に複雑になるのでCを使う

C言語を理解すること自体の御利益

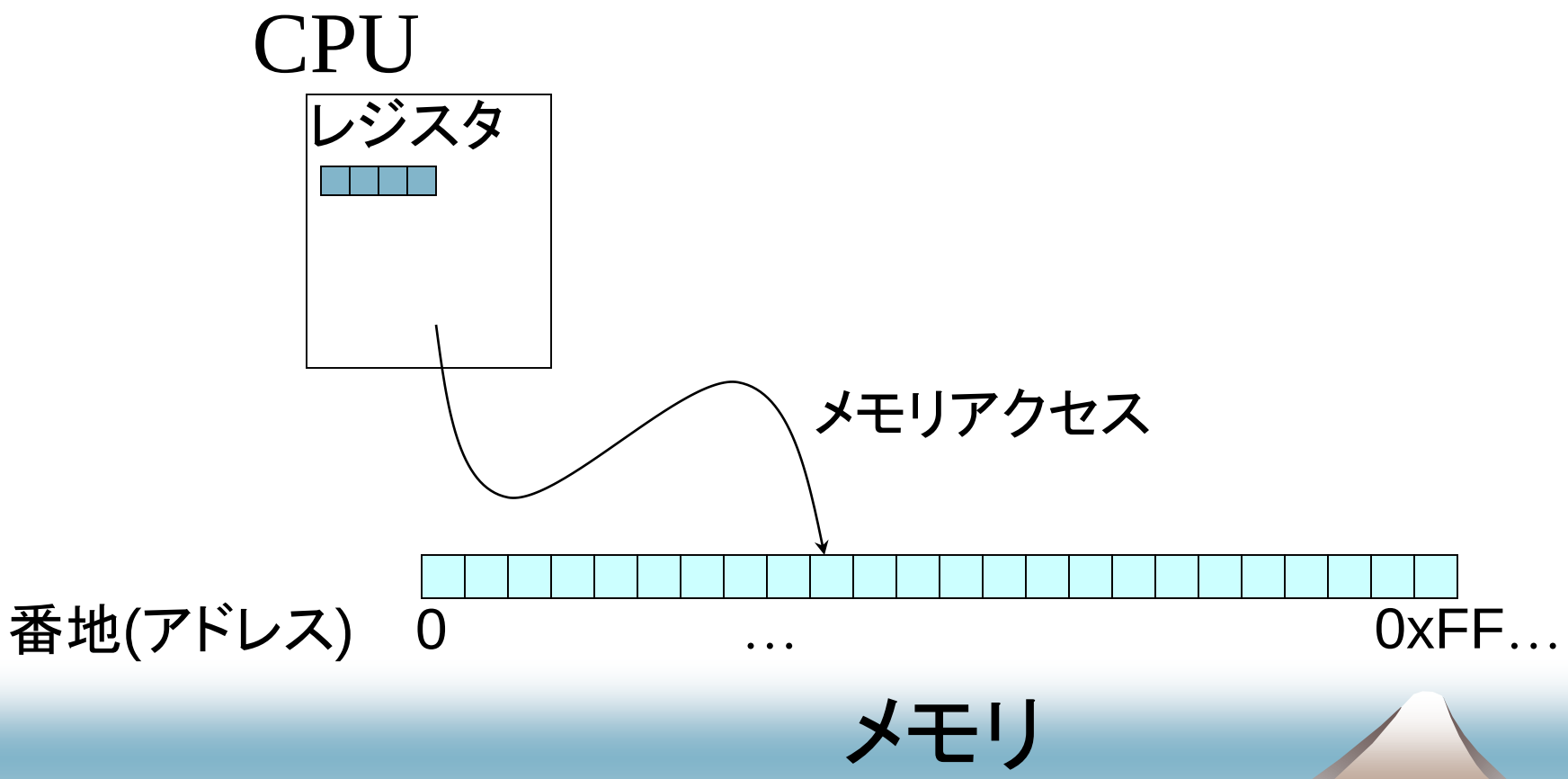
- 初級編:
  - ポインタってなーに? 関数ポインタ? なにそれ?
  - mallocってなに? いつ使うの?
  - Segmentation Faultってなに? いつおきるの?
- 中級編:
  - Cではポインタと配列は同じだって聞いたけど…
  - 関数呼び出しってどう実現されてるの?
- 上級編:
  - C/C++言語ならではの理解不能バグの理解

中心的テーマ

全部説明しはじめたらコンパイラの授業になってしまうので…

- Cプログラムによるメモリの使われ方:
  - Cプログラムで用いる「変数，配列など」がどこに格納されており，どのようなCの式，文が，どのようにメモリをアクセスするか?
  - 関数呼び出しの仕組み
    - ローカル変数の格納場所(スタック)
    - 呼び出しの入れ子の実現

復習: CPU, メモリ

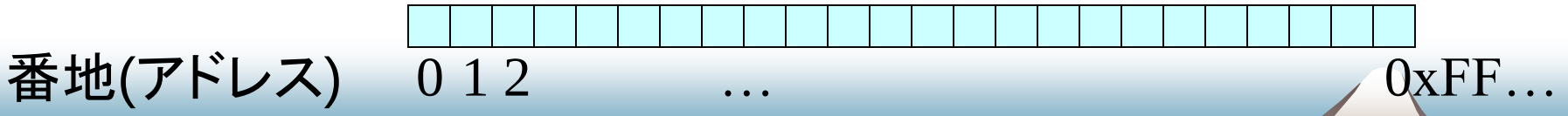


コンピュータとは

- コンピュータは，CPU内の少量の記憶領域(レジスタ; 数個～100個程度)と，CPU外の記憶領域(メインメモリ，主記憶)を用いて計算を行う機械である
- 主記憶には，プログラムが使うあらゆるデータが格納されている
  - ワープロで開いている文書，デスクトップの絵，etc.
  - C言語の変数，配列，etc.

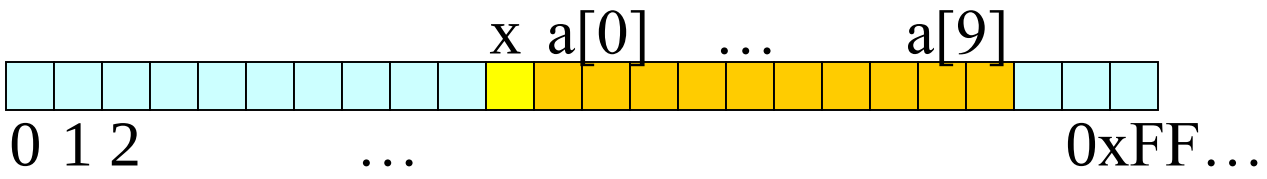
主記憶は単なるバイト列

- 主記憶へのアクセスの仕方:
  - 番地(アドレス)を指定して記憶「場所」を指定
  - 番地は単なる整数(0, 1, 2, …)
  - 「番地」として許される値は?
    - さしあたって,「任意の整数」と理解しておく
    - もちろん実際は「任意」ではない. 詳細後日



もちろんCプログラムもメモリをアクセスしている「だけ」

- int x; int a[10];
  foo() {
    a[3] = x;
  }



- 常にイメージしておくと良いこと:
  - C言語で用いている変数，配列，…ありとあらゆる「記憶域」がメモリのどこかに，プログラムが正しい限り重なりなく，格納されている

ポインタってなに?

- ポインタとは要するにアドレスのことである
- char * *p* = …;
  - *p*に格納されているもの: アドレス(整数)
  - **p* は*p*に格納されているアドレスをアクセスする
  - *p*[5] は「*p*に格納されているアドレス+5」番地をアクセスする
  - *p*[0] と**p*は同じもの

ものはためし.

変な番地(918番地)にアクセスするプログラムを書いてみよう

```
int main() {
  char * p = 918;
  return *p;
}
```

今起きたこと

- 「918番地」はアクセスすることが許されていない番地だった
  - OSが間違った・悪意のあるプログラムからシステムを保護するために働いた(メモリ保護): 詳細後日
- 一般的な用語: メモリ保護違反
  - Segmentation Fault = メモリ保護違反を表すUnix用語

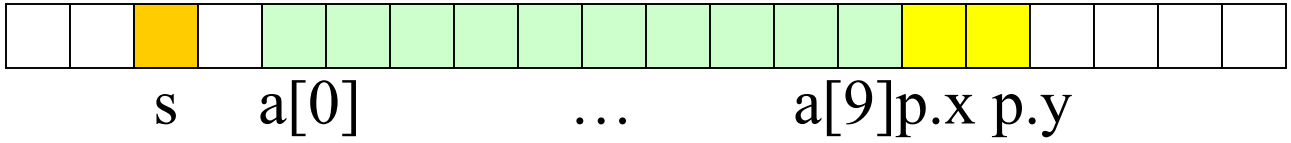
まめ知識

- 生成されたアセンブリを見る
  - gcc –S a.c (出力: a.s)
  - cl /FA a.c (出力: a.asm)

いろんな変数のアドレスを見る

- &*x* : *x*の値が格納されているアドレス
- int s;　　/* 単純な変数 */
  typedef struct point { int x; int y; } point;
  point p;　/* 構造体の変数 */
  int a[10]; /* 配列 */
  main() {
  　printf(“%d %d %d %d %d %d¥n”,
  　　&s, &p.x, &p.y, &a[0], &a[5], &main);
  }

で，所詮はすべてメモリの中…



&a[10] = &p.x !!

C言語の理解不能バグの源泉

C言語においては「実はアドレス」なものがいっぱい

- int x;
  … &x …;　/* 変数のアドレス*/
- int a[10];
  … a … ;　/* 配列 */
- int * s = “abcdef”;
  … s … ;　/* 文字列 */
- int * q = malloc(100);
- … q … ;　/* メモリ割り当て関数の返り値 */
- In * p = 918;
  … p … ;　/* 実はアドレスは整数! */

C言語 vs. 他の言語

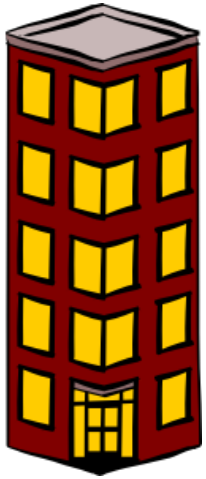
- どんな言語でも究極的には機械語(整数と浮動小数点数しかない世界)で動いている
  - いろいろなものをメモリに置き, そのアドレスで表している仕組み自体は同じ
- C言語の特徴はその「仕組み」を(ポインタという形で)包み隠さず見せているところ
  - 混乱のもとでもあり(コンピュータの仕組みを知ってしまえば)自然かつ単純なところでもある

(C言語についてよく言われること)
ポインタ=配列?

- ポインタ変数の定義と配列の定義は全く別物
  - int * p;　/* アドレス一個分の領域を確保.
    そこに勝手なアドレスを格納できる.
    Intを格納するための領域は確保されない */
    int a[10]; /* int 10個分の領域を確保.
    aはその先頭のアドレス */
- 要約:
  - 配列の定義(int a[10])は領域の確保を伴うが，ポインタ(int * p)の定義は伴わない
  - ポインタ変数(p)には勝手なアドレスを格納できるが，配列名(a)はあくまで確保された領域の先頭のアドレスであり，勝手なアドレスを代入できない

変数(一件分)
int x;

配列(多数件)
int x[10];



ポインタ
(住所だけ; 土地は??)
int * x;

東京都文京区
本郷 あさひ壮 3号室

ポインタに関するよくある間違い

- int foo {
  int a[10];
  int * p;
  a[0] = 10; /* OK */
  p[0] = 20; /* NG : どこをアクセス
  するかは運任せ! */
  }
- ポインタ変数の初期化忘れ

ポインタと配列が似ているところ

- 式の中で現れた場合，両者とも要するにアドレスのことである
- int a[10]; int * p = a;の元で以下は同じこと
  - … + a[0] + … と … + p[0] + …
  - … + a[3] + … と … + p[3] + …
  - … + (a + 5) + … と … + (p + 5) + …
- ややこしいことに以下の二つは違う
  - &a と &p

もう一歩理解を深める

- Cプログラム内で使われる記憶域が，どのように確保されるかをもう少し詳細に理解する

Cプログラムで現れる3種類の記憶領域

- 大域(global)/静的(static)変数・配列
  - 関数外に書かれた変数・配列
  - 関数中で，staticと書かれた変数・配列
- 局所(local)変数・配列
  - 関数定義の中に書かれた変数定義
- ヒープ
  - malloc, new (C++)などで確保される

大域/静的 変数(配列)

```
int x;              /* 大域変数 */
int a[10];          /* 大域配列 */
foo() {
  static int y;         /* 静的変数 */
  static int b[10]; /* 静的配列 */
  …;
}
```

大域/静的 変数(配列)の確保

- プログラム開始時に，各変数(配列)が，あるアドレスに割り当てられ，プログラム終了まで，その領域はその変数(配列)のために確保され続ける
  - 他の目的に使われない
- 「無限の寿命を持つ」

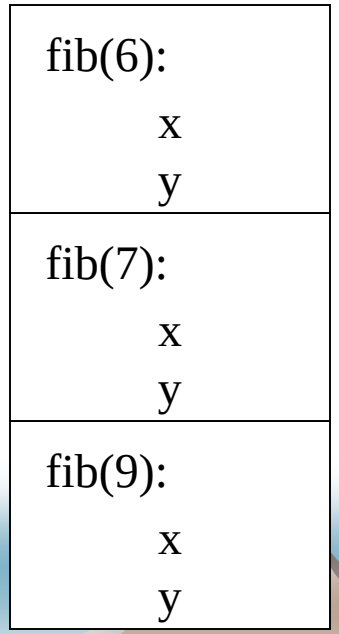

アドレス
時間

局所変数(配列)

- int fib(int n) {
  if (n < 2) return 1;
  else {
  int x = fib(n-1); /* x : 局所変数 */
  int y = fib(n-2); /* y : 局所変数 */
  return x + y;
  }
- 大域変数ほど話が簡単ではない
- fib(10)のxと，fib(9)のxは別の領域でないと×

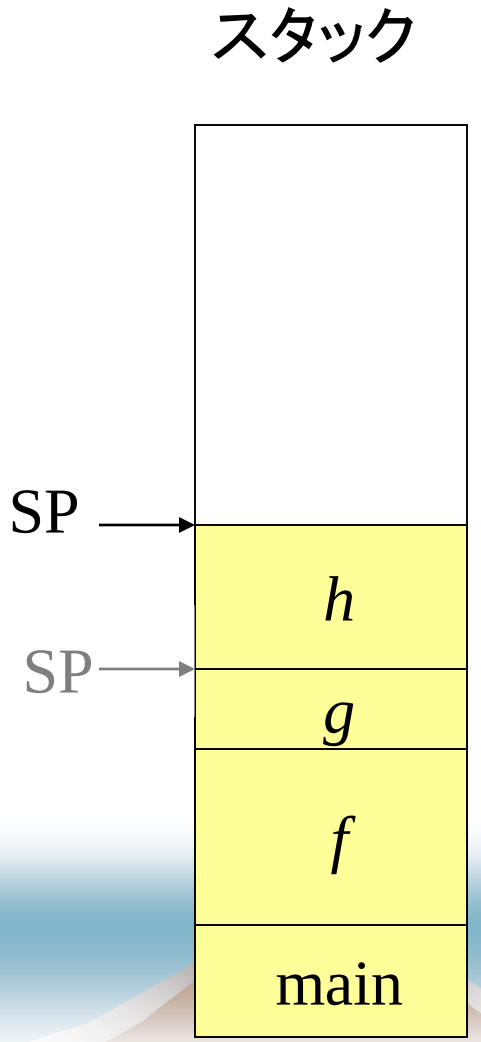
局所変数の確保

- 関数が呼び出されたときに，「その呼び出しの実行のための」領域を見つけて確保する必要がある
- そのためのデータ構造: スタック

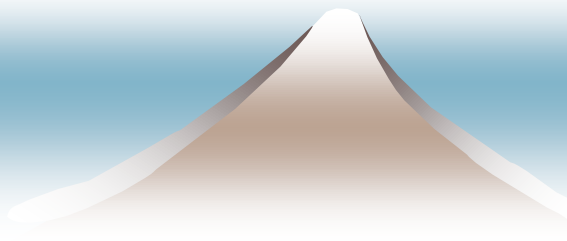


スタック

- 関数が実行を開始するとき
  - その関数が使う局所変数の大きさに応じて，空き領域からメモリを確保
- 終了(return)するとき
  - 開始時に確保した分だけメモリを開放
- 確保・開放の実際: SPをずらす
- 注: もちろん各スレッドがひとつのスタックを持っている

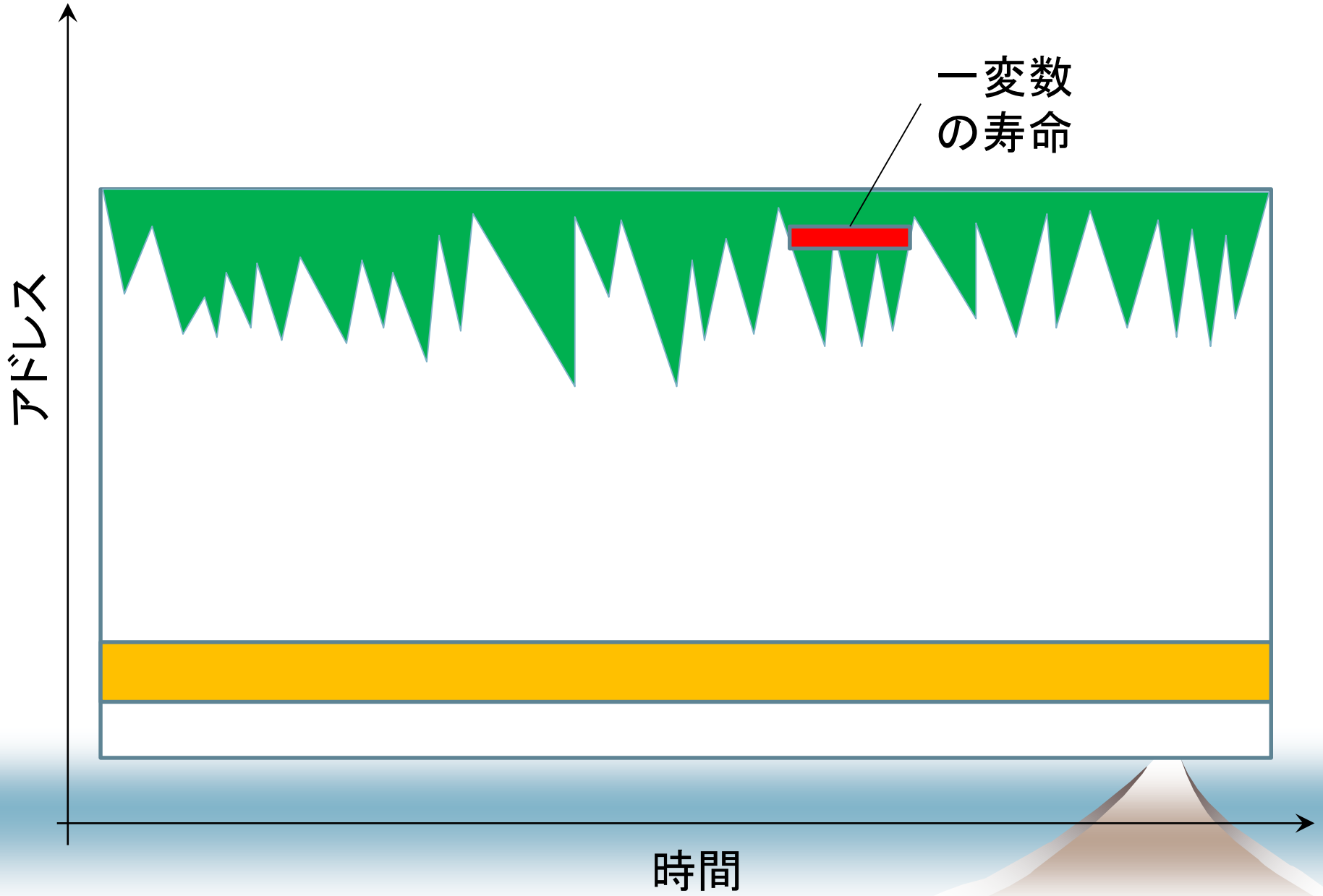


局所変数のアドレス観察

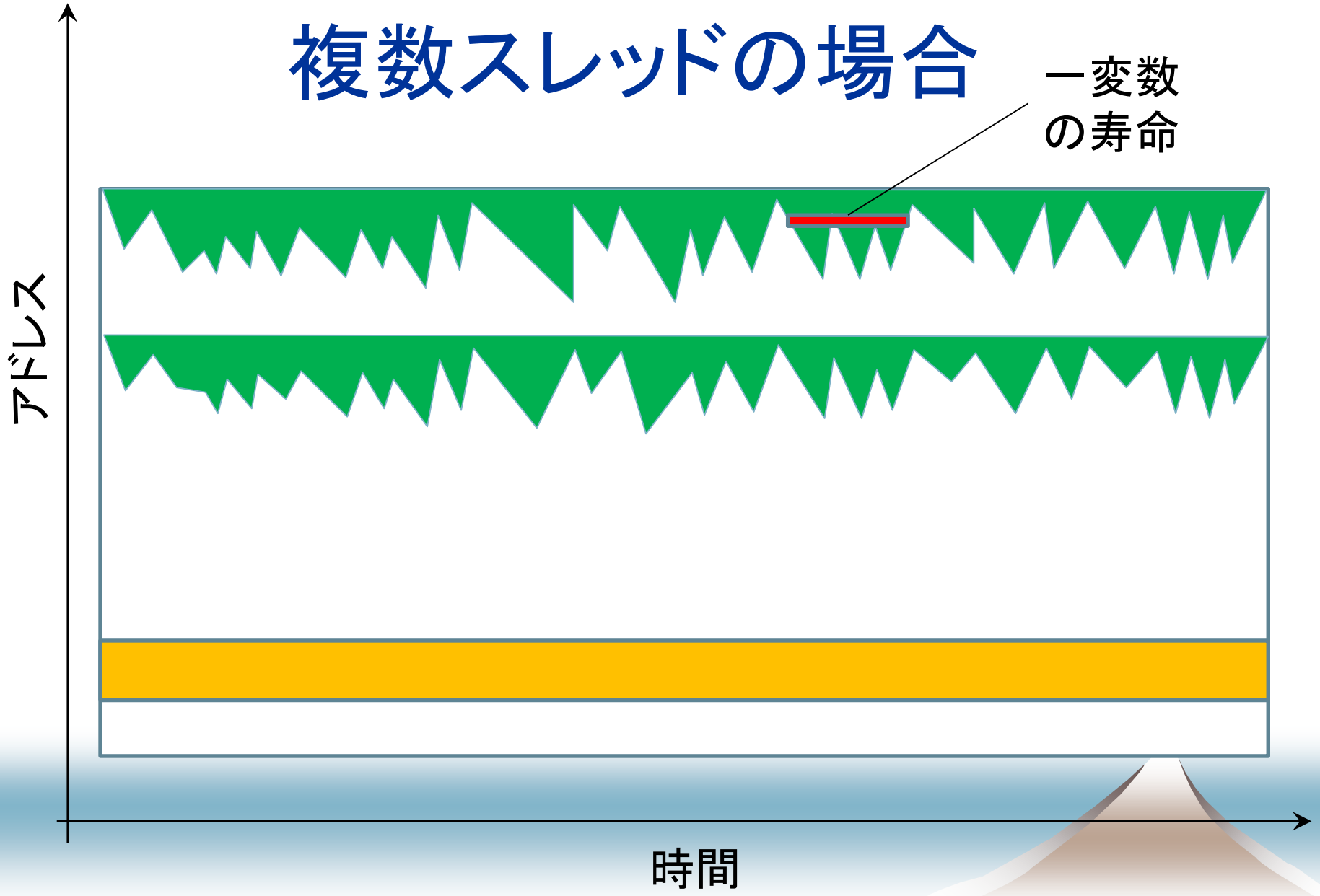


まめ知識

- ほとんどのOS/CPUで，スタックは大きい番地から小さい番地へ向かって「伸びる」
  - 確保: SP −= size;
  - 開放: SP += size;
- 「伸ばす」ためにはその方が都合が良かった(複数スレッドがあればどの道問題だが)


一変数
の寿命
アドレス
時間

複数スレッドの場合



局所変数・配列関係の悲しい間違い

- 局所変数・配列は，それを確保した関数呼び出しが終了すると開放される
  - 「寿命は関数終了まで」
  - 実際に起こること：将来呼び出された関数の局所変数として使われ，まず間違いなく塗りつぶされる

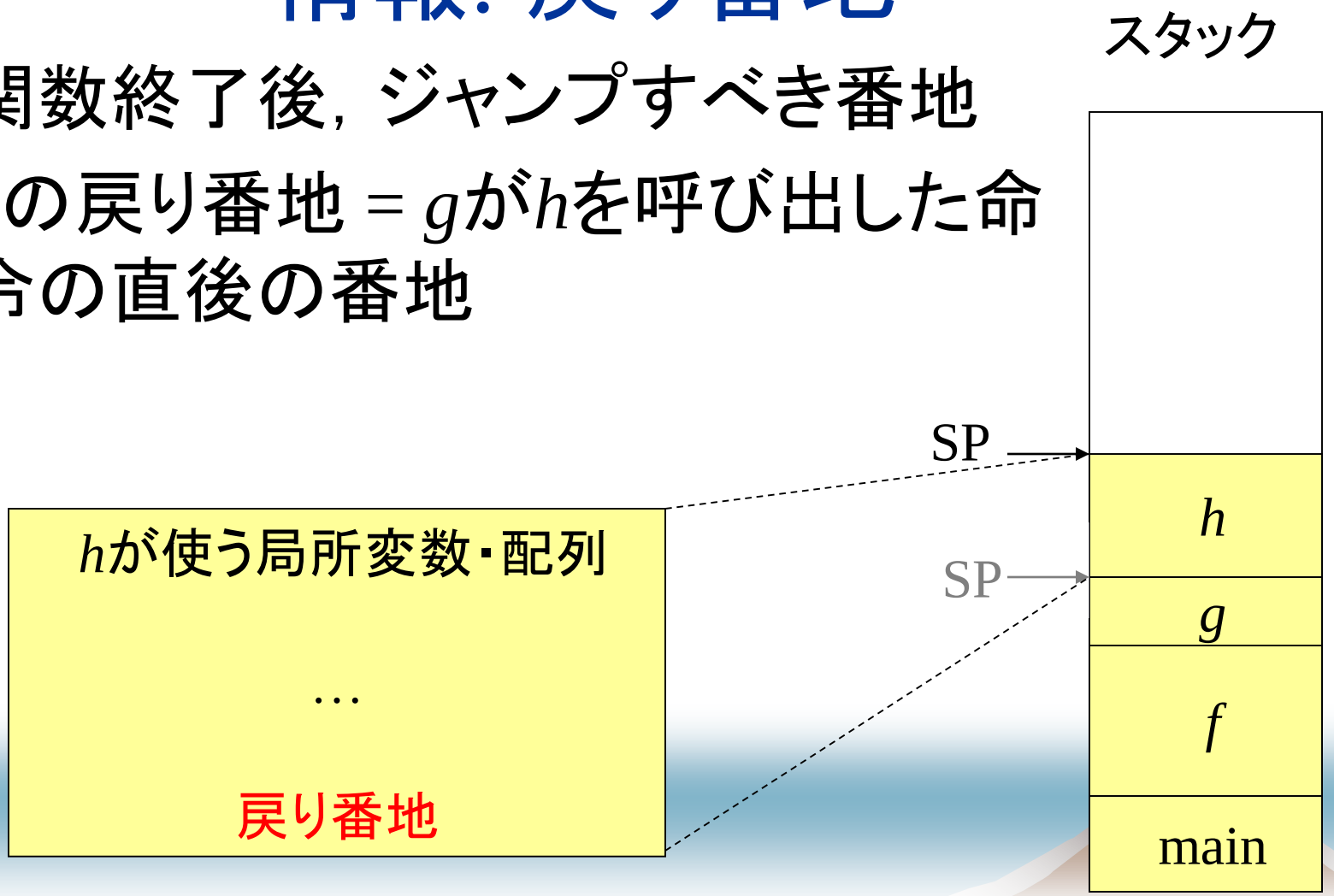
```
int* foo() {
  int a[10];
  return a;
}
```

素朴な疑問: スタックって無限に伸びるの?

- もちろんNO
- スタックの使いすぎ(スタックオーバーフロー)に注意
  - 巨大な局所配列
  - 深すぎる関数呼び出しの入れ子
- スタックの大きさ
  - スレッド生成時に指定できる
  - mainスレッドは?

スタックに格納されている重要な情報: 戻り番地

- 関数終了後，ジャンプすべき番地
- *h*の戻り番地 = *g*が*h*を呼び出した命令の直後の番地



バッファオーバーラン

- 戻り番地の破壊
- 関数呼出し後，制御が「あさっての方向」へ
  - 数々のsecurity holeの源泉
- 「あさっての方向」ならまだ良いが…

バッファオーバラン対策

- (OS) スタックのアドレスを実行ごとにランダムに変える
- (C言語処理系)
  - 関数実行開始時: 返り番地付近に特定の値（カナリア語）を入れておく
  - 関数から戻る直前: カナリア語が踏みつぶされていたらoverrunされたとみなす

ヒープ

- 任意の時点で確保，任意の時点で開放できる領域

	確保	開放
大域変数・配列 静的変数・配列	プログラム開始時	されない(プログラム終了時)
局所変数・配列	関数開始時	関数終了時
ヒープ	任意(malloc, new)	任意(free, delete)

API

- C:

  T * p = (T*)malloc(size); /* sizeバイト確保 */

  …

  free(p); /* 開放 */

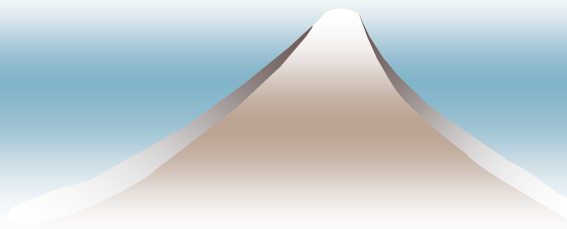
- C++:

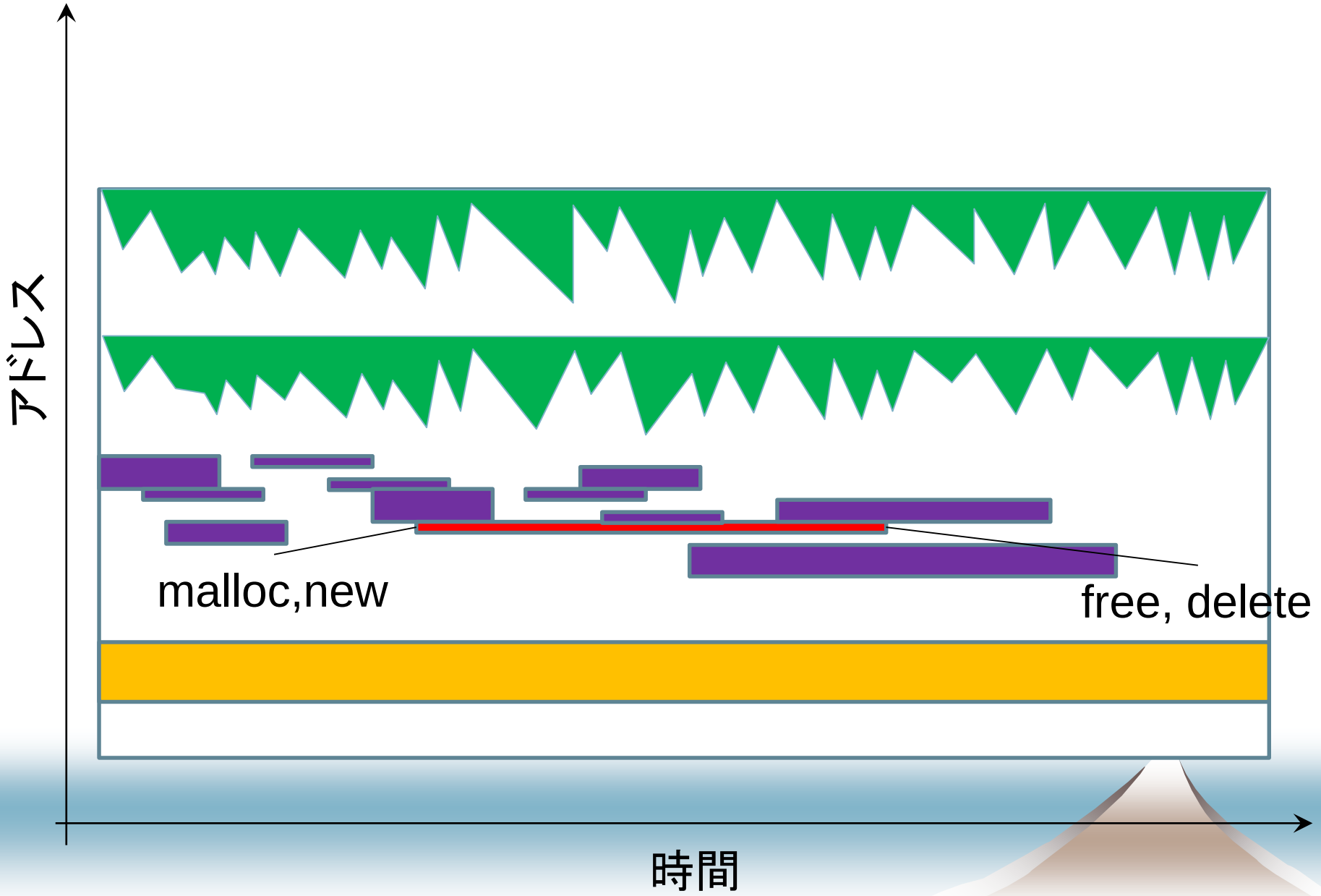
  T * p = new T;

  …

  delete p;

mallocのアドレス観察




アドレス
malloc,new
free, delete
時間

malloc関係の間違い(1)

- p = malloc(…);
  …
  free(p);　　/*早すぎるfree */
  …
  q = malloc(…); /* その後，運悪く同じ番地が他のデータに割り当てられる */
  p = 10;　　/ 意図しないデータを破壊 */

malloc関係の間違い(2)

- freeのし忘れ(メモリリーク)
- char * p = (char *)malloc(…);
  …; *p = …; ...; … = *p;
  /* 本当はこの辺で使い終わっているとする */
  …;
  char * q = malloc(…);
  …;
  /* しかし永遠に再利用されない */

簡単な応用問題

- 2次元の点をあらわす構造体:
  typedef struct Point { int x, y } Point;
- 新しい点を作る関数
  Point * mk_point(int x, int y)
  を書け

間違い1

- Point * mk_point(int x, int y) {
  Point * p;
  p->x = x; p->y = y;
  return p;
  }

間違い2

- Point * mk_point(int x, int y) {
  Point p[1];
  p->x = x; p->y = y;
  return p;
  }
- Point * mk_point(int x, int y) {
  Point p;
  p.x = x; p.y = y;
  return &p;
  }

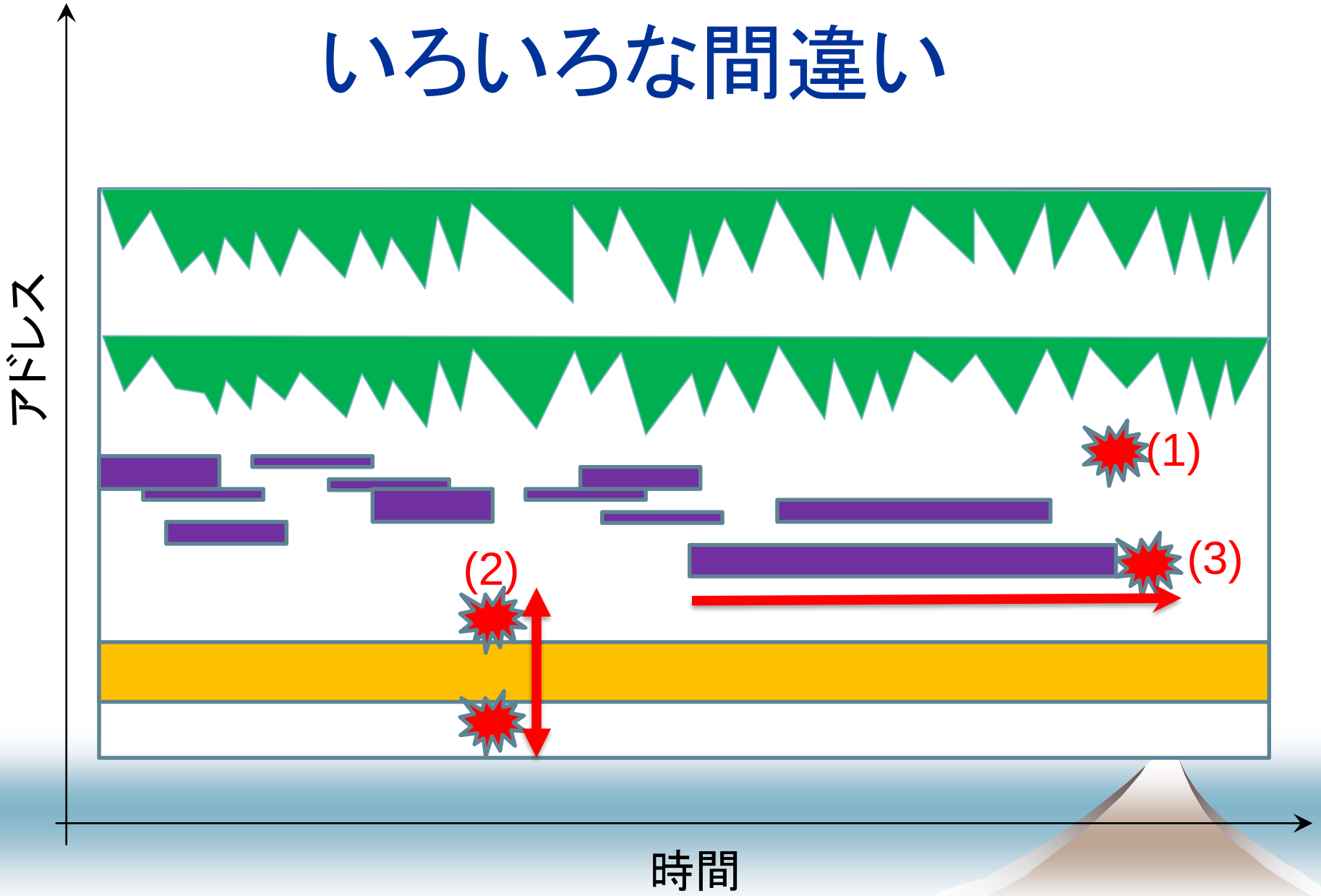
間違い3

- Point p;
  Point * mk_point(int x, int y) {
    p.x = x; p.y = y;
    return &p;
  }

正解

- Point * mk_point(int x, int y) {
  Point * p = (Point *)malloc(sizeof(Point));
  p->x = x; p->y = y;
  return p;
  }

いろいろな間違い



間違いの簡単な分類(1)
ポインタの捏造

- 割り当てられた領域と全く無関係な場所をアクセス
  - char * p = 918; ...; *p = …; /* 918番地って… */
  - char * p; …; … = *p; …　　/* 初期化忘れ */

間違いの簡単な分類(2)
領域外アクセス(out of bounds)

- 割り当てた領域をはみ出してアクセスする
  - char a[100]; … a[100] = …; …
  - char * a = (char *)malloc(10); ... a[10] = …;
  - char * a = (char *)malloc(10);
    gets(a); /* 何バイト書かれるか不明 */
  - typedef struct point { double x; double y; } * point_t;
    point_t p = (point_t)malloc(sizeof (point_t));
    p->x = …; p->y = …;
    /* よくある間違い */
  - char * a = (char *)malloc(10);
    strcpy(a, “0123456789”); /* わかりますか? */

間違いの簡単な分類(3)
寿命後のアクセス(premature free)

- 領域の寿命を超えてアクセス(早すぎる解放)
  - char * mk_data() { char a[1]; return a; }
    int main() { char * p = mk_data(); *p = …; }
  - char * p = (char *)malloc(1);
    …; free(p); …;
    *p = …;
  - (double delete)
    char * p = (char *)malloc(1);
    …; free(p); …;
    free(p);

間違いの簡単な分類(4)
メモリ(memory leak)

- もう使われない領域を再利用しない
  - 遅すぎる解放(または決して解放しない)
  - メモリリーク (freeすべき場所でfreeしない)
- プログラム終了まで，大した量のメモリを使わなければ問題ない
- 長時間の実行の後，突如問題になる(実行が急に遅くなるなど)

「安全な」言語

- 「安全」≈メモリをただしく使う≈前述のようなエラーを起こさないか, 検出する
  - メモリ安全(memory safe), ポインタ安全(pointer safe)などとも言う
- 「安全」といっても最低保証くらいの安全
  - 安全な言語で書いたプログラムは決して脆弱性を持たず, 個人情報も決して漏洩しないという意味ではない

「最低保証」くらいの安全

- 厳密な定義は難しいが,
  - 配列・構造体・文字列etc.として割り当てた場所以外を読み書きすることはできない(勝手な番地, 配列の終端を越えた場所, etc.)
  - 配列・構造体・文字列etc.として割り当てた場所を読み書きするには, その配列・構造体・文字列etc. への参照を得なければならない
    - Aへの参照を得ずしてAにアクセスできない
- C/C++はそうではなかった

「安全」の保証: 基本方針

- (1)ポインタ捏造～：
  - 実行時の null ポインタ検査
  - 実行時または実行前の型検査
- (2)領域外～：型検査+実行時の添え字範囲検査
- (3)寿命後～：自動ゴミ集め(garbage collection)
- (4)リーク～：自動ゴミ集め(garbage collection)
- 以下は主に(1)ポインタ捏造を防ぐアプローチについて

ポインタ捏造をおこさない言語にするには?

- 一番単純な違反 (ポインタに整数を代入)
  - char * p = 918;
    p[0] = 10;
- もう少し複雑な違反 (ポインタ=整数を禁止すればいいというものではない)
  - struct Foo { int p; };
    struct Bar { char * p; };
    Bar * b = …;
    Foo * f = …;
    f = b; f->p = 918; b->p[0] = 10;

- Foo * f = …;
  void * p = f;
  Bar * b = p;
  union Uoo { char * p; int x; };
  Uoo * u = …;
  u->x = 918;
  u->p[0] = …; /* おそらく918番地に書く */

誤解:「ポインタ型」がなければ安全?

- およそどんな言語でも以下は共通
  - データ(配列, 文字列, オブジェクト etc.)を表わすのにメモリを確保し,
  - データを, 確保されたアドレスを通じて保持し,
  - データの中身(配列要素, オブジェクトのフィールド etc.)を読み書きするのに, そのアドレスを元にメモリを読み書きしている
- これが安全に実行されるには何らかの「仕組み」が必要

「ポインタ型」がなくても危険なものは危険

- 「ポインタ型」がなくても実行の方式がCと根本的に違うわけではない
- Foo o = new Foo();

  …

  o.f = 10;

あいているメモリ領域確保

oは確保された領域のアドレスを保持

oに保持されている値をアドレスとしてfを取り出すアドレスを計算し，そこをアクセス

cf. Python…
o = [1, 2, 3]
…
o[1]

ポインタ捏造をさせないための二つの基本アプローチ

- 「動的に(実行時に)」保証する:
  - 多くのスクリプト言語，Lisp, Schemeなど
  - 動的な型検査を行う
- 「静的に(実行前に)」保証する:
  - Java, ML, Haskellなど
  - 静的な型付けを行う

動的な保証

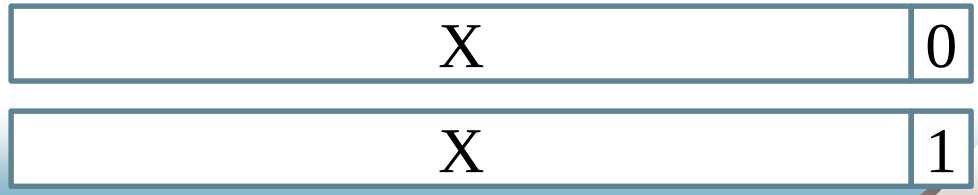
- Foo o = new Foo();
  if (…) { o = 918; }
  o.f = 10;
- 実例:
  - Python, Ruby, Lisp, Scheme, Prolog, JavaScript, …

動的な型検査
実行時にoがfというフィールドを持つデータでなければエラー

動的な保証(動的型検査)の仕組み

- 最低の基本:
  - X番地に割り当てられたデータ(アドレスX)と
  - 整数のX

  が実行時に区別できるようにするデータのタグ付け
- 例
  - X番地:
  - 整数のX



動的型検査

- より一般には, 実行時に「そのデータの型(種類)」が分かるようにデータを表現しておく
- あらゆる操作が「正しい型」のデータに対して行われているかどうかを実行時に検査する
  - 行われていなければ実行時に「型エラー」がおきる

静的な保証

- Foo o = new Foo();
  if (…) { o = 918; }
  o.f = 10;

静的な型安全保証
そもそもこういう代入をさせない
実行前(コンパイル時)に検出,「禁止」

- Foo f = new Foo();
  Bar b = new Bar();
  f = b; /* これも同様にエラー */
- 実例: Java, ML, etc.

静的型検査(静的な型付け)

- (理想的には)目的:
  - 実行時に型エラーが起きないことを実行前に保証する
  - 型エラーを起こす可能性があるプログラムはそもそも実行させない

静的型付けの基本的仕組み

- 最も単純な静的型付け:
  - すべての変数には一種類の型の値しか代入できない
  - 配列や構造体の要素も同様
  - ⇒ プログラム中に現れるすべての式の型が静的に(実行前に)一通りに決まる
  - 代入文(や引数渡し)は, 両者の型が一致しなければ実行前にエラーとする

例

- class Foo { int x; }
  class Bar { Foo f; }
- Bar b = …;
  if (…) { b = 918; }　/* Bar = int ⇒ エラー */
  b = new Foo(); /* Bar = Foo ⇒ エラー */
  b.f = new Foo(); /* Foo = Foo ⇒ OK */
- Bar[] a = …;
  a[i].f.x = 13; /* int = int ⇒ OK */

動的保証 vs. 静的保証

- 動的 …
  - 遅い
  - 間違ったプログラムに対する「安全網」にはなるが, そもそもの間違い防止にはならない
- 静的 …
  - 理想的には実行時の型検査不要
  - ある部分の間違い(型の間違い)を自動的に発見
  - 一見理想的(Cなみに速い, だが安全) 欠点は??

単純な静的型付け言語の弱点

- 柔軟性の欠如: 制約が厳しすぎる(実行すればエラーを起こさないプログラムも静的な型エラーとなることがある)
- 再利用性の欠如: 静的な型エラーを起こさないようにすると, 無駄な重複の多いプログラムになってしまう
- 「すべての式の型が静的に(実行前に)一通りに決まる」という制約から生ずる

例

- Fooの配列とBarの配列を整列する関数は例えアルゴリズムが同じでも別々に書かなければならない
  - Sort(Foo[] a) { … }
  - Sort(Bar[] b) { … }
- 同様の例: 可変長配列, ハッシュ表, etc.
- 「型によらず共通なアルゴリズム」を一つのコードで書く方法がない
  - ⇒もう少し複雑な静的型システム

柔軟性と安全性の両立

- 多相型
  - オブジェクト指向言語における継承・部分型
  - 型変数による多相, テンプレート

継承・部分型

- class Point2D { int x; int y; };
  class Point3D extends Point2D { int z; }
  /* 許してもOKなのは? */
  Point2D a = new Point3D(); /* 2D = 3D */
  Point3D b = new Point2D(); /* 3D = 2D */
  a.x + a.y
  b.x + b.y + b.z

部分型による多相

- Pont2D の変数 p に入っているのは今や実際には, Point2Dおよびそれを継承(extends)したクラスのオブジェクト
  - なんであれ p.x, p.y を持つことが保証される
  - p.z は静的型エラー(たとえ偶然 Point3D であっても実行前にエラーとなる)

Quiz

- Point2D[] p = …; /* Point2Dの配列 */
  Point3D[] q = …; /* Point3Dの配列 */
  /* 許してもOKなのはどっち? */
  Point2D[] a = q; /* Point2D[] = Point3D[] */
  Point3D[] b = p; /* Point3D[] = Point2D[] */