本講義の演習課題

- 課題
  - 数値シミュレーション
  - 光源と光センサー
  - 光源の光のうちどれだけがセンサーに検出されるか。
  1. 光源とセンサーの位置を変えてみる。
  2. 鏡をおいてみる。どのように置けばどうなる。
  3. レンズをおいてみる。どのように置けば…
  4. 光源は直方体もしくは円筒の内部一様に分布している。その光源からの光を効率よく集める光学系を設計する。

プログラミング言語の歴史

- 手続き型言語(～70年代)
  - アセンブラ
  - BASIC
  - FORTRAN
  - •グローバル変数
  - •ローカル変数
  - •関数
- 構造化言語(～80年代)
  - C
  - Pascal
  - •文の構造化～文のグループ化
  - •変数の構造化～変数のグループ化
- オブジェクト指向言語(～90年代以降)
  - C++
  - Java
  - Python
  - •関数のグループ化～メンバー関数
  - •型のグループ化～クラス継承、テンプレート
- どう変わってきたかは講義の中で説明します

プログラミング言語発展の原動力

- 意味上の違いを文法上で区別できるようにする。
  - グループ化(関係するものを集める)
  - コンパイラ(計算機)が検査可能
- 同じ(似た)ことを二度書かない。
  - 共通性の発見
  - コードの再利用

C++の構文

- コメント
- 式と文
- 関数
- 構造体
- クラス

C++のコメント

- /*と*/に囲まれた部分。
  - 複数行にわたって良い。
  - 入れ子にはならないから注意
  - /*　/*　*/ ここはコメントの外 */
- //から行末まで。
  - 「行」という概念はC++ではあまり使われない。

作業1

- コメントのみのファイルを作りコンパイルしてみる
- comment.cppを作る。内容は欄外に
- g++ -c comment.cpp
- これでcomment.oが出来る。

```
// this is a C++ comment line
/*

this part is a comment block

*/
```

式

- 型
- 定数(リテラル)
- 変数
- 配列
- ポインター
- 参照
- 演算子

型

- 組み込み型(基本データ型)
  - 論理型
  - 文字型
  - 整数型
  - 浮動小数点数型
- ユーザ定義型
  - 列挙
  - 配列
  - 参照・ポインター
  - 構造体・クラス

精度

- 整数型
  - 符号付き整数と符号なし整数
  - char/short int/long int/long long int
- 浮動小数点数
  - IEEE
  - float/double/long double

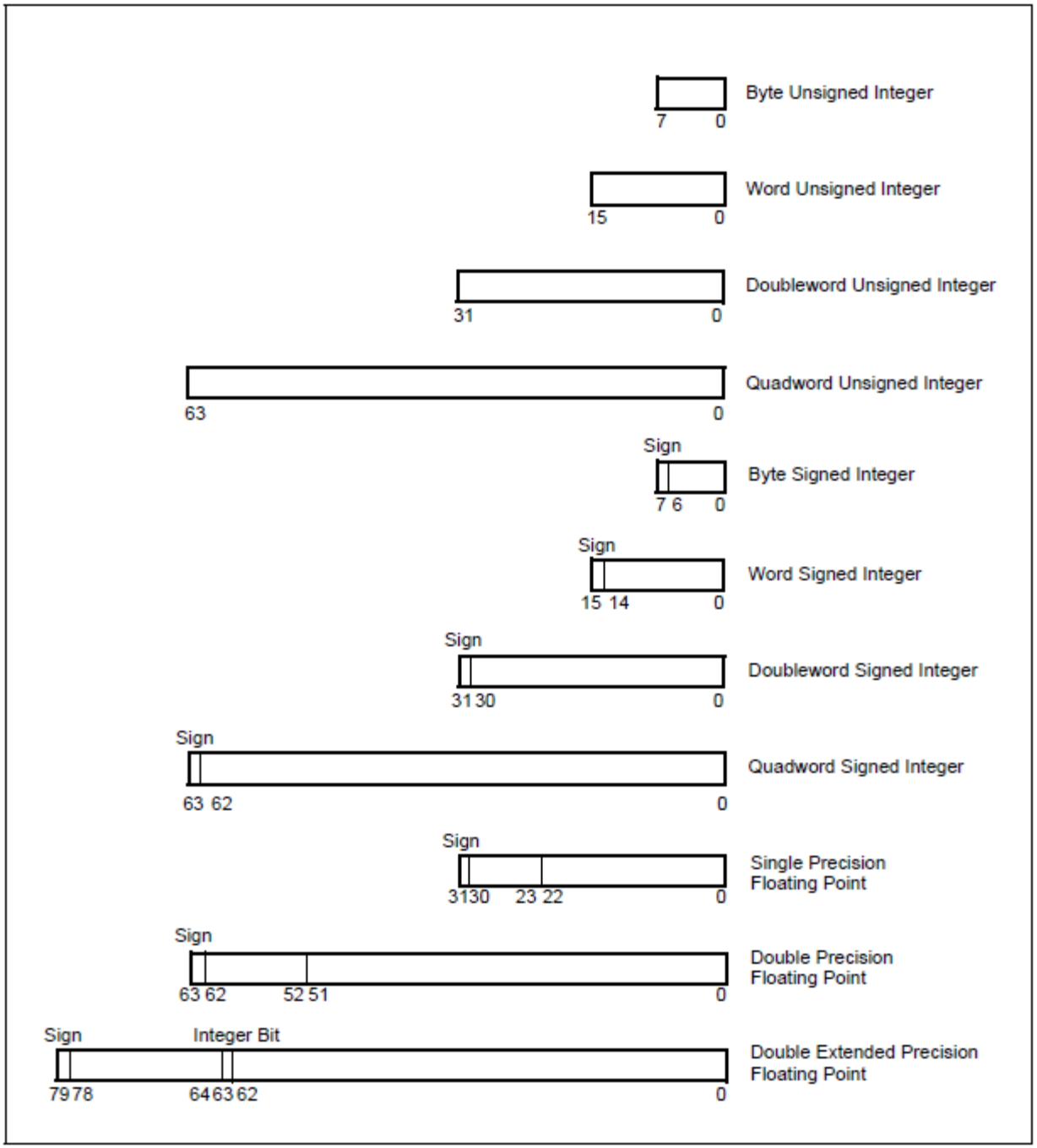


Figure 4-3. Numeric Data Types

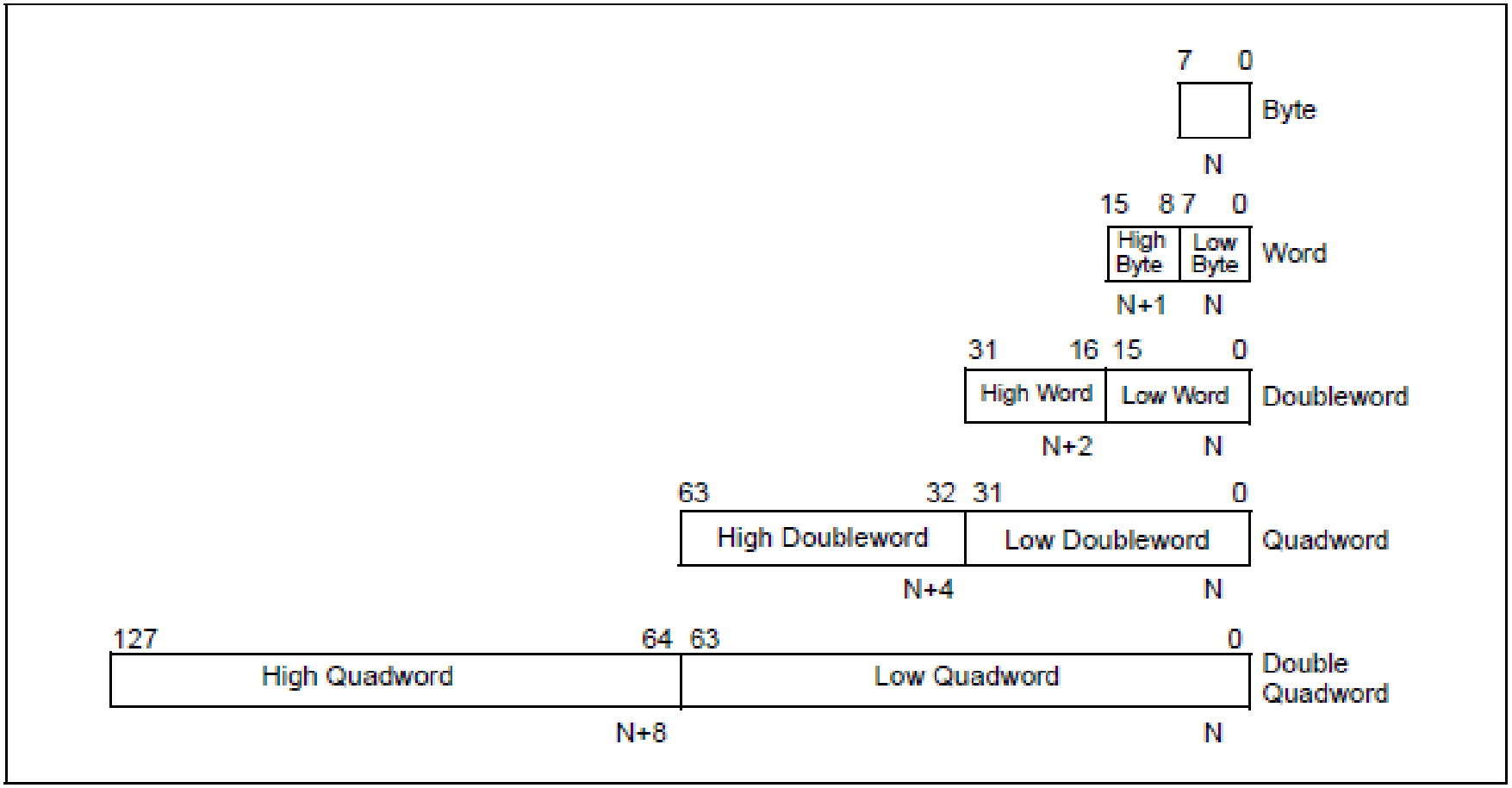
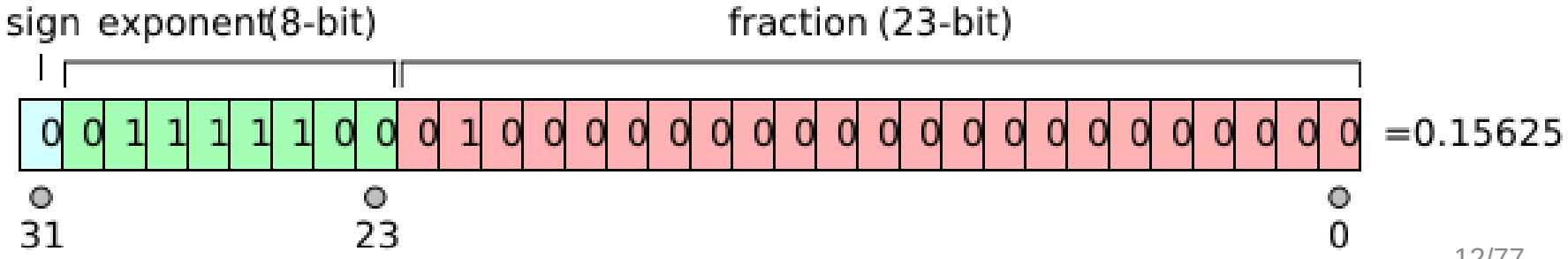
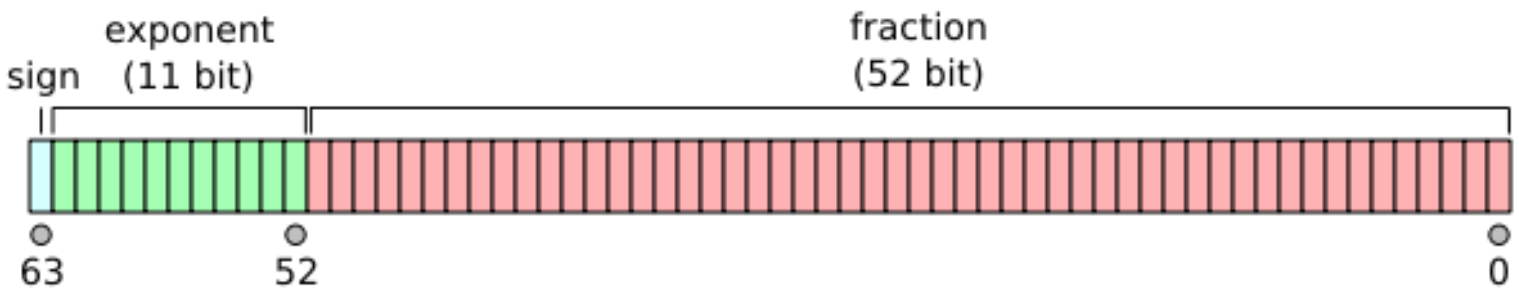


Figure 4-1. Fundamental Data Types

Table 4-5. Real and Floating-Point Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
IEEE Single-Precision Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 ← 1. (Implied)





1. The positive and negative numbers closest to zero (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the Fraction field) are

$$\pm 2^{-1074} \approx \pm 5 \times 10^{-324}$$

2. The positive and negative normalized numbers closest to zero (represented with the binary value 1 in the Exp field and 0 in the fraction field) are

$$\pm 2^{-1022} \approx \pm 2.2250738585072020 \times 10^{-308}$$

3. The finite positive and finite negative numbers furthest from zero (represented by the value with 2046 in the Exp field and all 1s in the fraction field) are

$$\pm((1-(1/2)^{53})2^{1024}) \approx \pm 1.7976931348623157 \times 10^{308}$$

定数

- 文字定数
  - ‘a’
  - ‘¥n’や’¥t’など制御文字も
- 整数定数
  - 123 (10進数)
  - 0123(8進数)
  - 0x123(16進数)
- 浮動小数点定数
  - 3.1415926
  - 4.56e-12

列挙型

- enum
- ひとまとまりの数字の集まり
  - enum { INITIAL, STANDBY, RUNNING };
  - それぞれ0,1,2の値が割り振られる。
- 値を指定することもできる。
  - enum { E_FILENOTFOUND = -1 };
- 型に名前をつけることもできる。
  - enum ERROR { E_INVALID_ARGUMENT=-2,…

変数

- 宣言と定義
  - 宣言：名前を決める。いくつやっても良い。
  - 定義：実体を決める。一つしか許されない。
- 定義
  - 型(unsigned/signed, char/int/float/double)
  - サイズ(long, short)
  - 記憶域(extern, static, register, auto)
  - 定数か否か(const)
  - 初期値設定

変数の例

- unsigned char a;
- const char c = ‘C’;
- int i;
- const int n_entries = 10;
- long unsinged int u = 123;
- double f;
- const double pi = 3.1514926;
- extern int errno;
- static int error_code;

作業2

- 変数を定義してみる。
- variables.cppを作る。
- g++ -c variables.cpp

```
/*

      variables.cpp

*/

unsigned char a;
const char c = 'C';
int i;
const int n_entries = 10;
long unsigned int u = 123;
double f;
const double pi = 3.1415926;
extern int errno;
static int error_code;
```

配列

- 配列の定義
  - int iarray[10];
- 配列の初期化
  - int idir[4] = {1,2,3,4};
- 配列要素
  - int i3 = iarray[3];

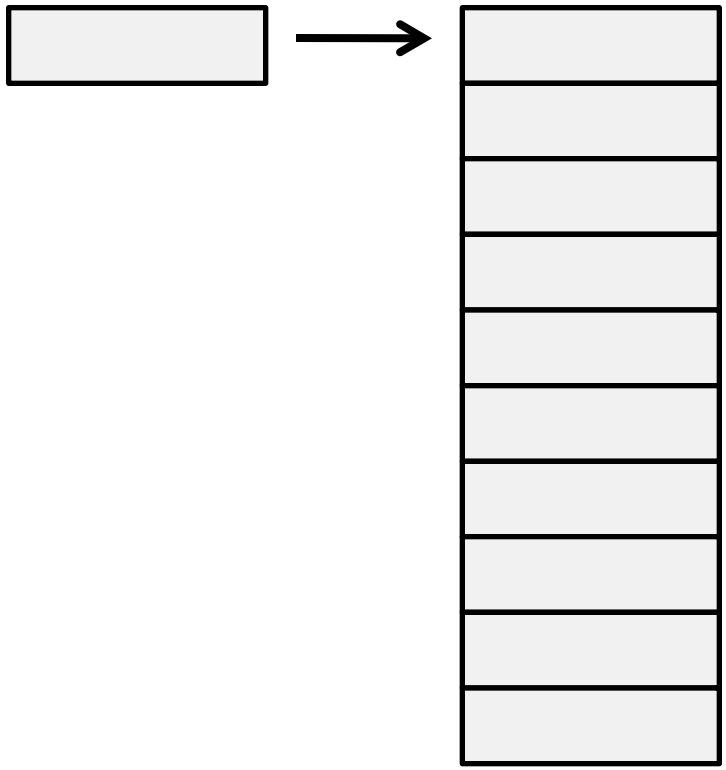
作業3

- 配列を定義してみる。
- arrays.cppを作る。
- g++ -c arrays.cpp

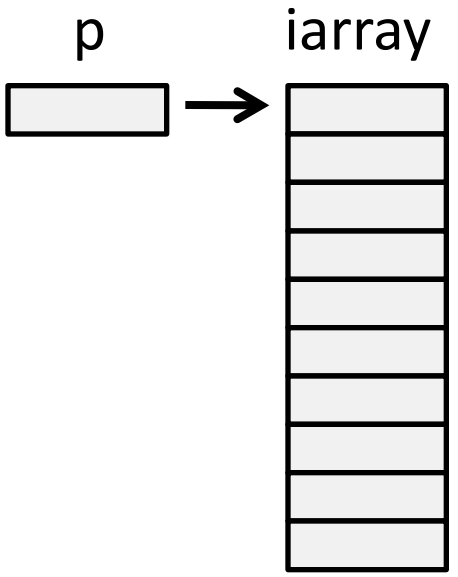
```
/*

      arrays.cpp

*/
int iarray[10];
int idir[4] = {1,2,3,4};
int i3 = idir[3];
```

ポインター

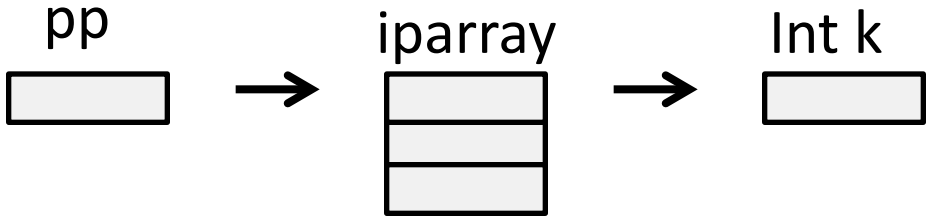
- ポインターの定義
  - ポインター型
  - ポインターの初期値
- ポインターの利用
  - 文字列リテラル
  - 文字型ポインター
  - 配列ポインター
- ポインター演算



ポインターの定義

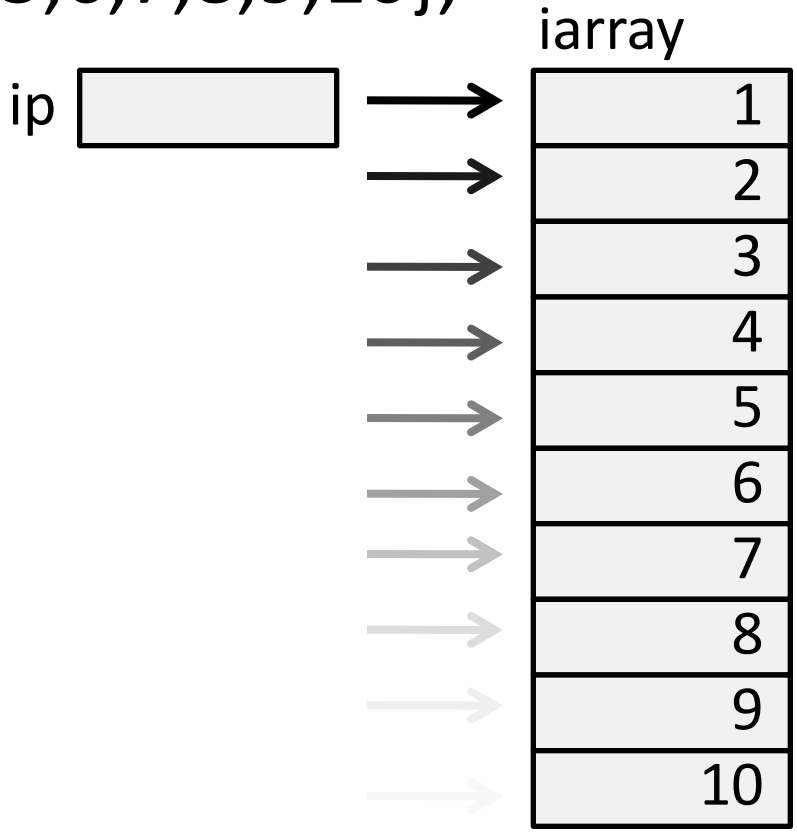


- ポインター型
  - int * p; //pは整数型へのポインター
  - int iarray[10];
  - p = &iarray[0]; //pは最初の配列要素を指す。
  - p=iarray;　　//これでも良い。iarrayは配列の参照。
- ポインターへのポインターも可能
  - int ** pp;　　//ppはポインターへのポインター
  - int * iparray[10];　　//ポインター配列
  - pp = iparray;



ポインターの利用

```
int     iarray[10]={1,2,3,4,5,6,7,8,9,10};
int     * ip;
ip = iarray;
int     sum;
for( int i = 0; i<10; i++ )
    sum += * ip ++;
```



作業4

- ポインターを使ってみる。
- pointers.cppを作る。
- g++ pointers.cpp
- ls
- 今回初めてa.outを作った。
- ./a.outで実行。
- 出力が書かれていないので何も起こらないけど。

```
/*

      pointers.cpp

*/
main( ) {

int iarray[10]={1,2,3,4,5,6,7,8,9,10};
int * ip = iarray;

int sum = 0;

for( int i = 0; i < 10; i ++ )
   sum += * ip ++;

}
```

ポインターの利用その2

```
#include    <iostream>
main( int argc, char ** argv ) {
   for( int i=0; i < argc; i ++ )
        std::cout << i << “: “ << * argv ++ <<
   std::endl;
}
```

作業5

- メインプログラムへの引数の受け渡し
- pointer2.cppを作る。
- g++ -o pointer2 pointer2.cpp
- ./pointer2
- ./pointer2 a 234

```
/*

      pointer2.cpp

*/
#include <iostream>

main( int argc, char ** argv ) {

   for( int i = 0; i < argc; i ++ )
      std::cout << i << ": " << * argv ++ << std::endl;

}
```

参照

- 参照の定義
  - int　　ibody;
  - int & iref = ibody;
- 参照の利用
  - 参照は必ず参照先を伴う。
    - ポインターは参照先を持たないときがある。
  - しかしだます方法はある。

作業6

- 参照を使ってみる
- reference.cppを作る。
- g++ -c reference.cpp
- エラーメッセージが表示される
- reference.cpp:8: error: ‘iref2’ declared as reference but not initialized
- 本当はiref3も正しくないのだが

```
/*

      reference.cpp

*/
int     ibody;
int &   iref = ibody;
int &   iref2;
int *   ip;
int &   iref3 = * ip;
```

const

- ユーザ定義定数
  - 整数ではenumが使える。
  - それ以外の型ではconstを使って定義する。
- ポインターとの組み合わせ
  - 定数ポインター
    - const int * p;　　// constはint*にかかる。pは可変
  - ポインター定数
    - int * const p;　　//constはpにかかる。初期化必要
- 参照との組み合わせ

項

- 定数
- 変数
- 式
  - 演算子

演算子

- 単項演算子
  - +,-,~,&,*
  - ++,--,~,!
  - new,delete
- 二項演算子
  - +,-,*,/,%,<<,>>
  - =,+=,-=,*=,/=,%=
  - ==,!=,<,>,<=,>=
  - &,|,~,&&,||
- 右結合と左結合
  - a+b+c　まずa+bを評価し、次にその結果で+cを評価
  - a=b=c　まずb=cを評価し、次にその結果でa=を評価
- 優先順位
  - 自信がないときは()で優先順位を確定すること

これらも演算子
::スコープ解決
.,->メンバー選択
[]要素取りだし
()関数呼び出し

優先順位	演算	表記	結合
	スコープ解決	クラス名::メンバー	
18	スコープ解決	名前空間名::メンバー	
	大域解決	::名前	
	メンバー選択	オブジェクト.メンバー	
	メンバー選択	ポインター->メンバー	
	要素取りだし	ポインター[式]	
	関数呼び出し	表現(式の並び)	
	値の構築	型(式の並び)	
	後置インクリメント	変数++	
17	後置デクレメント	変数--	
	型識別	typeid(型)	
	実行時型識別	typeid(式)	
	実行時チェックつき型変換	dynamic_cast<型>(式)	
	コンパイル時チェックつき型変換	static_cast<型>(式)	
	チェック無し型変換	reinterpret_cast<型>(式)	
	const変換	const_cast<型>(式)	

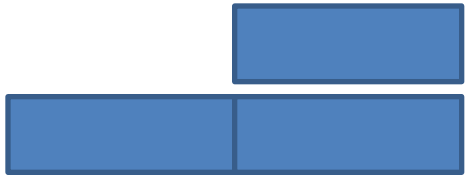
優先順位	演算	表記	結合
16	オブジェクトサイズ	sizeof 式	
	型のサイズ	sizeof(型)	
	前置インクリメント	++変数	
	前置デクレメント	--変数	
	1の補数(ビット単位反転)	^式	
	論理否定	!式	
	単項マイナス	-式	
	単項プラス	+式	
	アドレス取り出し	&変数	
	間接参照	*ポインター式	
	構築(領域確保)	new 型	
	構築(領域確保と初期設定)	new 型(式の並び)	
	構築(配置)	new (式の並び) 型	
	構築(配置と初期設定)	new (式の並び) 型(式の並び)	
	破棄(領域解放)	delete ポインター	
	配列の破棄	delete [] ポインター	

優先順位	演算	表記	結合
15	メンバー選択	オブジェクト.*メンバーへのポインター	
	メンバー選択	ポインター->*メンバーへのポインター	
14	乗算	式*式	
	除算	式/式	
	剰余	式%式	
13	加算	式+式	
	減算	式-式	
12	左シフト	式<<式	
	右シフト	式>>式	
11	未満	式<式	
	以下	式<=式	
	より大	式>式	
	以上	式>=式	
10	等しい	式==式	
	不等	式!=式	

優先順位	演算	表記	結合
9	ビット単位論理積	式&式	
8	ビット単位排他論理和	式^式	
7	ビット単位論理和	式|式	
6	論理積	式&&式	
5	論理和	式||式	
4	条件式	式?式:式	
3	代入	変数=式	右
	演算して代入	変数*=式、変数/=式、変数%=式、変数+=式、 変数-=式、変数<<=式、変数>>=式、 変数&=式、変数|=式、変数^=式	右
2	例外投擲	throw 式	
1	並び	式,式	

型変換

- プロモーション(高い精度への拡張)
  - short int is;
  - long int il;
  - il+is
- 暗黙の型変換
  - il=is
- 明示的な型変換(キャスト)
  - il=long int(is)



ポインター演算

```
int iarray[10];
int * p = iarray;
int sum = 0;
for( int i = 0; i < 10; i++ )
    sum += * p ++;
```

文

- 単文　項;
- 複文　{　単文もしくは複文　}
- 流れ制御命令文
  - 条件分岐
  - 繰り返し

流れ制御

- if文
- while文
- for文
- do文
- break文
- continue文
- switch文

if文

- if(項) 文

```
if( ivalue > imax ) imax = ivalue;
if( iref > maxref ) {
  std::cout << “Error: invalid index” << std::endl;
  return ERROR_INVALID_INDEX;
}
```

- if(項) 文1 else 文2

```
if( ivalue > 0 ) { … }
else if( ivalue < 0 ) {…}
else {…}
```

switch文

- switch(項) {caseラベル1:文…caseラベルn:文…}
- caseラベル
- defaultラベル
- breakによる脱出

if(i==STANDBAY) {…)
else if(i==RUNNING) {…}
else {…}
と書くよりわかりやすい。

```
switch(i) {
case STANDBY:
    …
    break;   //このbreakがないと次を続けて実行する。
case RUNNING:
    …
    break;
default:
    std::cerr << “Invalid state = “ << i << std::endl;
    break;
}
```

for文

- for(初期化文;判定項;更新文) 実行文

```
int i;
for(i=0;i<10;i++) {
    iarray[i]=i;
}
```

- 初期化文には変数定義も書ける。

```
for( int i=0; i<10; i++ ) {…}
```

- この場合iのスコープは

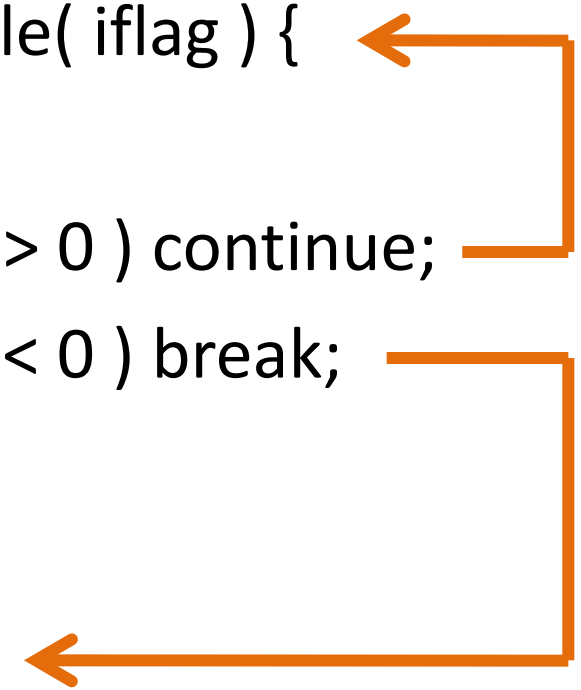
while文

- while(項) 文1

```
while( iref >0 ) {
    array[iref]=array[iref-1];
    iref--;
}
```

- breakとcontinue

```
while( iflag ) {
…
if( i > 0 ) continue;
if( i < 0 ) break;
….
}
```



作業8

- 流れ制御を確認
- flowcontrol.cppを書く。
- g++ -o flowcontrol flowcontrol.cpp
- ./flowcontrol
- ./flowcontrol 2 3 4
- ./flowcontrol 2 0 4
- ./flowcontrol 2 -1 4
- 何が起こるか見てみよう。

```
/*

      flowcontrol.cpp

*/
#include <iostream>
#include <stdlib.h>

int main( int argc, char ** argv ) {

int ivalue = 0;

   argv ++;

   for( int i = 1; i < argc; i ++ ) {

   int ivalue = atoi( * argv ++ );
      if( ivalue < 0 ) continue;
      if( ivalue == 0 ) break;

      while( ivalue > 0 ) {

         std::cout << ivalue -- << " ";
      }
      std::cout << std::endl;
   }

}
```

関数

- 関数の定義
- 関数の値(戻り値)
  - 任意の型の値を返す。ユーザ定義型を含む。
  - 必要のないときはvoid型
- 仮引数リスト
- ローカル変数
- 関数の呼び出し

関数の定義

```
int fsum( int n, int *p ) {
//関数名fsum,戻り値int,仮引数n,p
    int isum = 0;    //ローカル変数isum,i
    for( int i = 0; i < n; i ++ ) {
        isum += *p ++;
    }
    return isum;    //戻り値int型
}
```

ローカル変数

- 関数定義内部で定義された変数
- 変数定義文で生成される。
  - autoが仮定されている。
  - 動的に割り当てられる。
- 関数から制御が抜けるときに消滅する。
- 維持したいときはstatic宣言をする。

作業9

- function.cppを作る
- g++ -o function function.cpp
- ./function 10

```
/*

      function.cpp

*/
#include <iostream>
#include <stdlib.h>

int fsum( int n, int * p ) {

   int isum = 0;

   for( int i = 0; i < n; i ++ ) {
      isum += * p ++;
   }

   return isum;

}
```

```
int main( int argc, char ** argv ) {

   int iarray[ 10 ];
   int istart = 0;
   if( argc > 1 ) {
      argv ++;
      istart = atoi( * argv );
   }

   for( int i = 0; i < 10; i ++ ) {
      iarray[ i ] = istart + i;
   }

   std::cout << "Sum of " << istart << " to "
      << istart + 10 << " equals " << fsum( 10, iarray )
      << std::endl;


}
```

関数の戻り値とローカル変数

- ポインターを返す関数

```
int * func1( ) {
    int iarray[10];
    …
    return iarray;
}
```

- 参照を返す関数

```
int & func2( ) {
    int ivalue;
    …
    return ivalue;
}
```

作業10

- returnvalue.cppを書く
- g++ -c returnvalue.cpp
- 警告メッセージを確認

returnvalue.cpp: In function ‘int* func2(int)’:
returnvalue.cpp:13: warning: address of local variable ‘ivalue’ returned
returnvalue.cpp: In function ‘int& func4(int)’:
returnvalue.cpp:23: warning: reference to local variable ‘ivalue’ returned

```
/*

      returnvalue.cpp

*/

int func1( int iarg ) {
   int ivalue = iarg * iarg;
   return ivalue;
}

int * func2( int iarg ) {
   int ivalue = iarg * iarg;
   return & ivalue;
}

int * func3( int iarg ) {
   static int ivalue = iarg * iarg;
   return & ivalue;
}

int & func4( int iarg ) {
   int ivalue = iarg * iarg;
   return ivalue;
}

int & func5( int iarg ) {
   static int ivalue = iarg * iarg;
   return ivalue;
}
```

構造体

- 構造体の定義
- メンバーデータ
- 構造体オブジェクト
- メンバーデータへのアクセス

構造体の定義

```
struct vect {        //構造体vectの定義
    double x;        //第一のメンバーデータx
    double y;        //第二のメンバーデータy
    double z;        //第三のメンバーデータz
};                   //構造体vectの定義の終わり
```

構造体の利用

```
struct vect p;
p.x = 0.0;
p.y = 1.0;
p.z = 2.0;
struct vect * pp = &p;
double x = pp -> x;  // x = (*pp).x;と同じ
double fsize( struct vect * pp ) {
    return sqrt( pp->x * pp->x + pp->y * pp->y + pp->z*pp->z );
}
double fsize( struct vect & p ) {
    return sqrt( p.x * p.x + p.y * p.y + p.z * p.z );
}
```

作業11

- struct1.cppを書く
- g++ -o struct1 struct1.cpp
- ./struct1

```
/*

      struct1.cpp

*/

#include <iostream>

struct vector {
double x;
double y;
double z;
};

int main( ) {
   struct vector v;
   v.x = 1.0;
   v.y = 2.0;
   v.z = 3.0;

   struct vector * pv;
   pv = & v;

   std::cout << "x = " << pv -> x
      << " y = " << pv -> y
      << " z = " << pv -> z
      << std::endl;
}
```

構造体の利用(続き)

```
void fnorm( struct vect & p ) {
    double s = fsize( p );
    if ( s == 0.0 ) return;
    p.x /= s;
    p.y /= s;
    p.z /= s;
}
```

作業12

- struct2.cppを書く
- g++ -o struct2 struct2.cpp
- ./struct2 1 2 3

```
/*
      struct2.cpp
*/
#include <iostream>
#include <stdlib.h>
#include <math.h>

struct vector {
double x;
double y;
double z;
};

double size_vector( struct vector & v ) {
   return sqrt( v.x * v.x + v.y * v.y + v.z * v.z );
}
bool norm_vector( struct vector & v ) {
   double size = size_vector( v );
   if( size == 0.0 ) return false;
   v.x /= size;
   v.y /= size;
   v.z /= size;
   return true;
}
void print_vector( struct vector & v ) {
   std::cout << "( " << v.x << ", "
      << v.y << ", " << v.z << " )";
}
```

```
int main( int argc, char ** argv ) {
   struct vector v;
   v.x = 0.0;
   v.y = 0.0;
   v.z = 0.0;
   if( argc > 1 ) {
      argv ++;
      v.x = double( atoi( * argv ) );
   }
   if( argc > 2 ) {
      argv ++;
      v.y = double( atoi( * argv ) );
   }
   if( argc > 3 ) {
      argv ++;
      v.z = double( atoi( * argv ) );
   }
   std::cout << "v = ";
   print_vector( v );
   std::cout << " size = " << size_vector( v );
   norm_vector( v );
   std::cout << " normalized = ";
   print_vector( v );
   std::cout << std::endl;
}
```

クラス

- クラスの定義
- メンバーデータ
- メンバー関数
- スコープ

- 構造体からクラスへの発展
  - 引数に構造体へのポインターを持つ関数群
  - それぞれの関数が構造体の文脈上の役割を表す。
  - C++では構造体もクラスの一形態と考える。

クラスの定義

```
class vect {
public:
    vect( );
    vect( double sx, double sy, double sz );
    double size( );
    void norm( );
private:
    double x;
    double y;
    double z;
};
```

作業13

```
/*
      class_intro.cpp

*/
#include <iostream>

class Vector {

public:
   Vector( );
   Vector( double px, double py, double pz );
   void set( double px, double py, double pz );
   double getX( ) { return x; }
   double getY( ) { return y; }
   double getZ( ) { return z; }
private:
   double x;
   double y;
   double z;

};
```

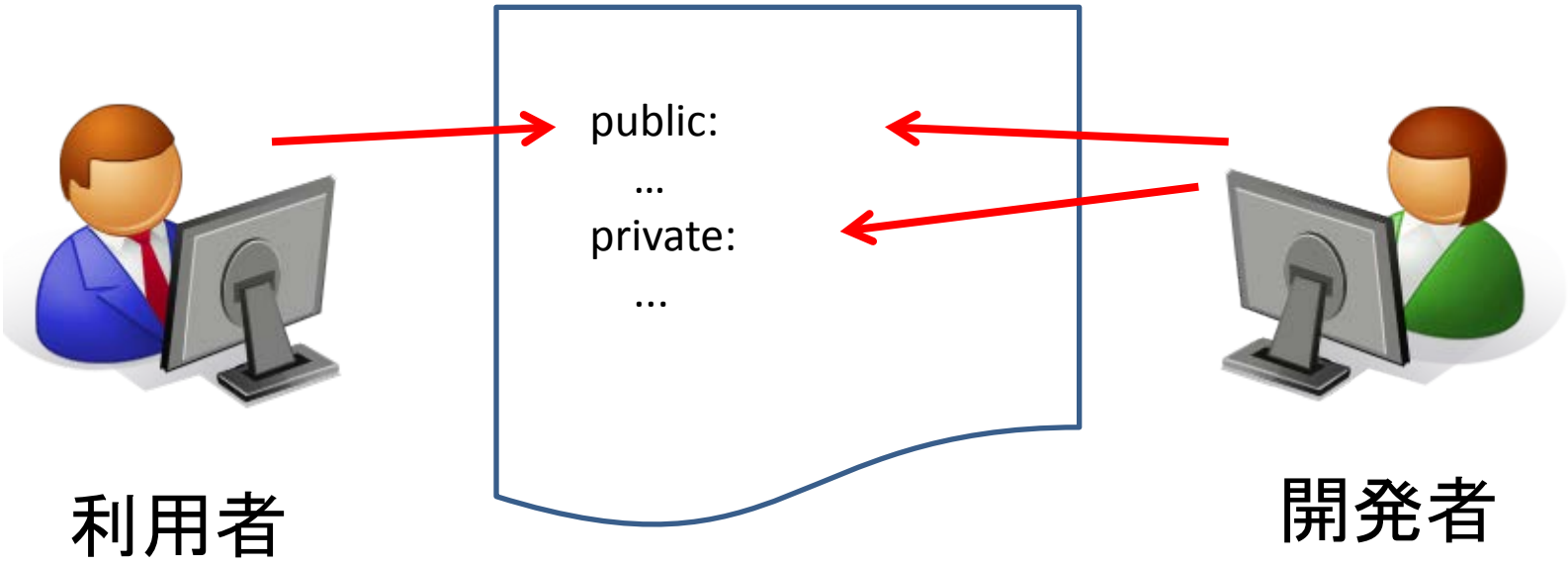
Class_intro.cppを書く
g++ -o class_intro class_intro.cpp
./class_intro

```
Vector::Vector( double px, double py, double
pz )
   : x(px), y(py), z(pz) {
}

void Vector::set( double px, double py, double
pz ) {
   x = px;
   y = py;
   z = pz;
}

int main( ) {

Vector v(1.0,2.0,3.0);
Vector * vp;
Vector & vr = v;

   std::cout << "X = " << v.getX( ) << " Y = " <<
v.getY( )
      << " Z = " << v.getZ( ) << std::endl;

   vp = & v;

}
```

メンバーの可視性

- public
- private
- protected

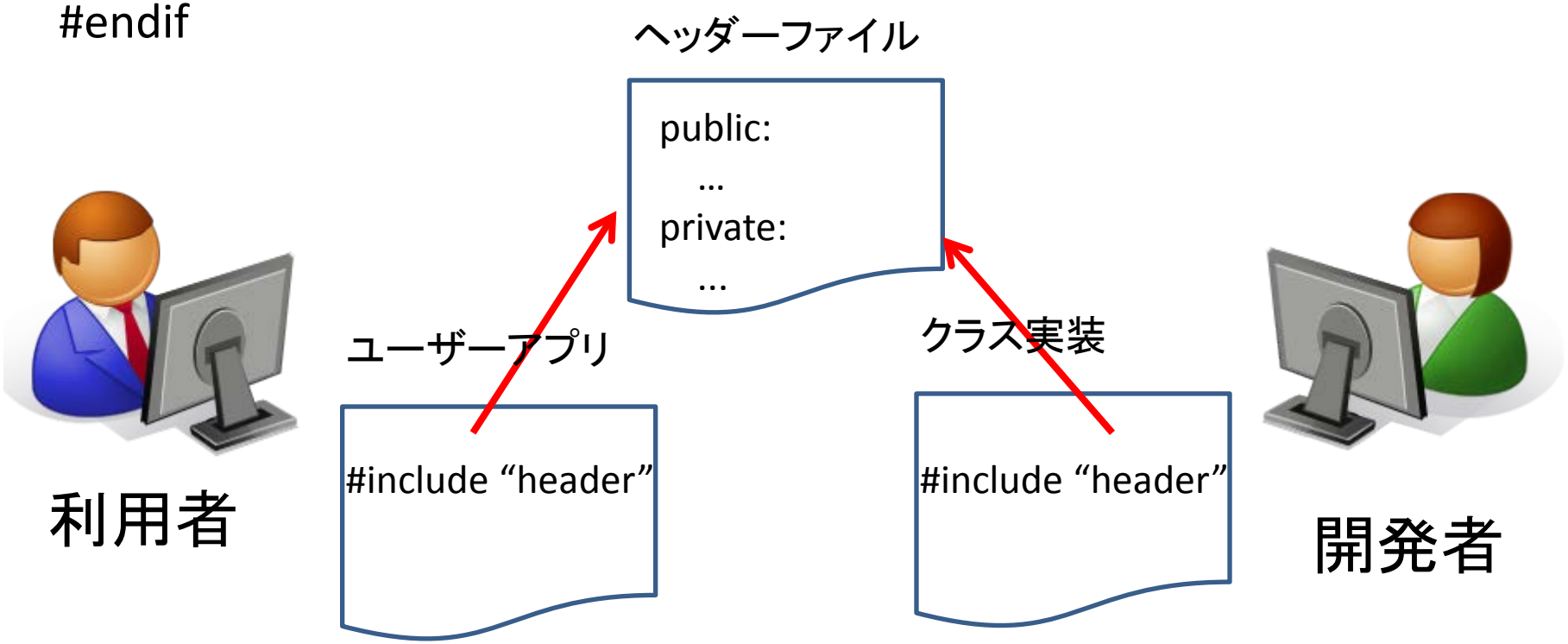
- インターフェース
  - 利用者は公開メンバーのみ参照。
  - 開発者は公開メンバーで規定された仕様に基づいて実装
  - 実装そのものは利用者からは見えない。



クラスヘッダーファイル

```
#ifndef __VECTOR_H
#define __VECTOR_H
class Vector {
…
};
#endif
```

クラスヘッダーファイルは開発者と利用者の両方が使う。



作業14

- 操作は次ページ

```
/*

      Vector_main.cpp

*/
#include "Vector.h"
#include <iostream>

int main( ) {

   double x = 1.0;
   double y = 2.0;
   double z = 3.0;
   Vector v( x, y, z );
   v.print( );
   std::cout << " size = " << v.size( ) << std::endl;
   v.normalize( );
   v.print( );
   std::cout << " size = " << v.size( ) << std::endl;

}
```

```
/*

      Vector.h

*/
#ifndef __VECTOR_H
#define __VECTOR_H

class Vector {

public:
   Vector( );
   Vector( double px, double py, double pz );
   void set( double px, double py, double pz );
   double getX( ) { return x; }
   double getY( ) { return y; }
   double getZ( ) { return z; }
   double size( );
   bool normalize( );
   void print( );
private:
   double x;
   double y;
   double z;

};

#endif
```

メンバー関数の定義

```
#include “Vector.h”
#include <math.h>
Vector::Vector( ) : x(0.0), y(0.0), z(0.0) { }
Vector::Vector( double sx, double sy, double sz ) :
   x(sx), y(sy), z(sz) { }
Vector::Vector( double sx, double sy, double sz ) {
   x = sx; y = sy; z = sz; }
double Vector::size( ) { return sqrt( x*x+ y*y + z*z ); }
void Vector::normalize( ) { double s = size( );
   x /= s; y /= s; z /= s; }
```

constの使い方

- constな関数。メンバーを変更しない。
  - int mfunc( ) const;
- constな値を返す
  - const char * cfunc( );
- constな引数を取る
  - int afunc(const char * carg);

作業14(続き)

Vector.h、Vector.cpp、Vector_main.cppを書く
g++ -c Vector.cpp
g++ -o Vecror_main Vector_main.cpp Vector.o
./Vector_main

```
/*

      Vector.cpp

*/
#include "Vector.h"
#include <iostream>
#include <math.h>

Vector::Vector( ) {
}

Vector::Vector( double px, double py, double pz )
   : x(px), y(py), z(pz) {
}

void Vector::set( double px, double py, double pz ) {
   x = px;
   y = py;
   z = pz;
}
```

```
double Vector::size( ) {
   return sqrt( x * x + y * y + z * z );
}

bool Vector::normalize( ) {
   double s = size( );
   if( s == 0.0 ) return false;
   x /= s;
   y /= s;
   z /= s;
   return true;
}

void Vector::print( ) {
   std::cout << "( " << x << ", " << y << ", " << z << " )";
}
```

特殊なメンバー関数

- コンストラクタ(構築子)
  - クラス名と同じ名前のメソッド
  - クラス生成時に呼ばれる
  - 自動的に生成される(デフォルトコンストラクタ)
  - 引数なしコンストラクタ
  - コピーコンストラクタ
- デストラクタ(破棄子)
  - ~(否定演算)にクラス名をつけた名前
  - クラス破棄時に呼ばれる

作業15

constructor.cppを書く
g++ -o constructor constructor.cpp
./constructor

```
/*

      constructor.cpp

*/

#include <iostream>


class Test {

public:
   Test( );
   Test( double p );
   Test( Test & t );

private:
   double value;
};
```

```
Test::Test( ) : value( 0.0 ) {
   std::cout << "Constructor without
arguments" << std::endl;
}

Test::Test( double p ) : value( p ) {
   std::cout << "Constructor with an argument"
<< std::endl;
}

Test::Test( Test & t ) : value( t.value ) {
   std::cout << "Copy constructor" << std::endl;
}

int main( ) {
   Test t1;
   Test t2( 1.0 );
   Test t3 = t2;

}
```

クラスオブジェクト

- クラスとオブジェクト: 型と変数の関係
- オブジェクトの定義
- オブジェクトの利用
- オブジェクトへの参照
- オブジェクトへのポインター
- オブジェクトのライフタイム

オブジェクトの定義

```
#include “Vector.h”
Vector p;
Vector q(1.0,2.0,-3.0);
    q.normalize();
    double sp = p.size();
```

オブジェクトの一生

- 関数内で定義されたオブジェクト
  - 関数内の定義位置で生成され関数から制御が抜けるときに消滅する。
- メンバーデータとして定義されたオブジェクト
  - クラスオブジェクトが定義されたときに生成され、そのオブジェクトが破棄されたときに消滅する。
- staticもしくは大域的に定義されたオブジェクト
  - プロセス生成時に生成されプロセス終了時に消滅する。
- newで生成されたオブジェクト
  - プロセス終了時もしくはdeleteにより消滅する。

newとdelete

- オブジェクトの生成と消滅を任意にプログラムできる。
- ポインターを定義し、newでオブジェクトの実体を生成する。
  - MyClass * pobject = new MyClass( );
- deleteでオブジェクトを破棄する。
  - delete pobject;
- 実体が破棄される前にポインターが破棄されるとアクセスの手がかりがなくなる。リーク。

作業16

new_delete.cppを書く
g++ -o new_delete new_delete.cpp
./new_delete

```
/*

   new_delete.cpp

*/
#include <iostream>

class Test {

public:
   Test( );
   ~Test( );

};

Test::Test( ) {
   std::cout << "Constructor" << std::endl;
}

Test::~Test( ) {
   std::cout << "Destructor" << std::endl;
}
```

```
int main( ) {
   Test * p;

   p = new Test;
   delete p;

   Test * pp;
   pp = new Test[10];
   delete [] pp;
}
```

staticメンバー

- メンバーデータをstaticに宣言できる。

```
MyClass {
public:
    static int status;
};
```

- staticなメンバーデータは別途定義が必要
  - (オブジェクトと無関係に存在できる)

```
#include “MyClass.h”
int MyClass::status = 123;
```

- メンバー関数をstaticに宣言できる
  - オブジェクトが生成されていなくても呼べる。
  - Staticなメンバーデータのみ扱える。

シングルトン

初期化子がprivateなのでオブジェクトの生成ができない。
できるのはこのクラスのメソッドの内部からのみ。

- 一つしか作りたくないクラス

```
class MySingleton {
    public:
        static MySingleton * get( );
    private:
        MySingleton( );
        static MySingleton * singleton;
};
```

作業17

Singleton.cppを書く
g++ -o singleton singleton.cpp
./singleton

```
/*

      singleton.cpp

*/
#include <iostream>

class Singleton {

public:
   static Singleton * get( );

private:
   Singleton( );
   static Singleton * thesingleton;

};
```

```
Singleton * Singleton::thesingleton;

Singleton::Singleton( ) {
   std::cout << "In the constructor" << std::endl;
}

Singleton * Singleton::get( ) {

   if( thesingleton == 0 ) thesingleton = new
Singleton( );
   return thesingleton;

}

int main( ) {

//    Singleton s;
   Singleton * sp;
   sp = Singleton::get( );

}
```

演算子オーバーロード

- 演算子の定義
  - クラスはユーザ定義型
  - 演算子もメンバー関数として定義できる。
- 左辺項と右辺項
  - 左辺項:このクラスの型。メンバー関数として定義
  - 右辺項:左辺の型が異なる。大域関数として定義
- friend宣言
  - 大域関数にprivateメンバーへのアクセスを許す

演算子の定義

```
class Vector {
public:
    …
    Vector & operator = ( Vector & p );
}

Vector & Vector::operator = ( Vector & p ) {
    x = p.x; y = p.y; z = p.z; return * this; }

…
p=q;
```

右辺項と左辺項

```
class Vector {
Vector operator * ( double d );
friend Vector operator * ( double d, const Vector & p );
};

Vector Vector::operator * ( double d ) {
    return Vector( x*d,y*d,z*d ); }
Vector operator * ( double d,const Vector & p ) {
    return Vector( d*p.x, d*p.y, d*p.z ); }
```

作業18

Vector.hを修正する。左の箱
Vector.cppに追加。下の箱
Vector_main.cppを修正する。次頁
g++ -c Vector.cpp
g++ -o Vector_main Vector_main.cpp
Vector.o (g++からここまで一行)
./Vector_main

```
/*

      Vector.h

*/
#ifndef __VECTOR_H
#define __VECTOR_H

#include <iostream>

class Vector {

public:
   Vector( );
   Vector( double px, double py, double pz );
   void set( double px, double py, double pz );
   double getX( ) { return x; }
   double getY( ) { return y; }
   double getZ( ) { return z; }
   double size( );
   bool normalize( );
   void print( );
   Vector & operator = ( Vector & v );
   friend std::ostream & operator << ( std::ostream & os, Vector & v );

private:
   double x;
   double y;
   double z;

};

#endif
```

```
Vector & Vector::operator = ( Vector & v ) {
   x = v.x;
   y = v.y;
   z = v.z;
   return * this;
}

std::ostream & operator << (std::ostream & os, Vector & v ) {
   os << "( " << v.x << ", " << v.y << ", " << v.z << " )";
   return os;
}
```

作業18続き

- 同様に様々な演算子を定義しよう。

```
/*
   Possible operators.

   Vector & operator += ( Vector & v );
   Vector & operator -= ( Vector & v );
   Vector & operator *= ( double a );
   Vector & operator /= ( double a );
   Vector operator + ( Vector & v );
   Vector operator - ( Vector & v );
   Vector operator * ( double a );
   friend Vector operator * ( double a, Vector &
v );
   Vector operator / ( double a );
   double operator & ( Vector & v ); // inner
product
   Vector operator * ( Vector & v ); // outer
product
*/
```

```
/*

      Vector_main.cpp

*/
#include "Vector.h"
#include <iostream>

int main( ) {

   double x = 1.0;
   double y = 2.0;
   double z = 3.0;
   Vector v( x, y, z );
   std::cout << v << " size = " << v.size( ) << std::endl;
   v.normalize( );
   std::cout << v << " size = " << v.size( ) << std::endl;

   Vector w;
   w = v;
   std::cout << w << std::endl;

}
```

作業19

- 演習の準備を始める
- 次のクラスを作る。
- Vector
  - 前回の設計にメンバーを追加する。
- Rectangle
  - 4角形。当面センサーをこれで実装する。
- Ray
  - 光線。
- Source
  - 光源。四方八方に等方にRayを放つ
- これらを使って光源から光線を放ち、それがセンターに入る個数を数える。

```
/*

      Vector.h

*/
#ifndef __VECTOR_H
#define __VECTOR_H

#include <iostream>

class Vector {
public:
   Vector( );
   Vector( double px, double py, double pz );
   void set( double px, double py, double pz );
   double getX( ) { return x; }
   double getY( ) { return y; }
   double getZ( ) { return z; }
   double size( );
   bool normalize( );
   void print( );
   Vector & operator = ( Vector & v );
   friend std::ostream & operator << ( std::ostream & os, Vector & v );
   Vector operator - ( Vector & v );
private:
   double x;
   double y;
   double z;
};

#endif
```

```
/*

      Vector.cpp

*/
#include "Vector.h"
#include <iostream>
#include <math.h>

Vector::Vector( ) {
}

Vector::Vector( double px, double py, double pz )
   : x(px), y(py), z(pz) {
}

void Vector::set( double px, double py, double pz ) {
   x = px;
   y = py;
   z = pz;
}

double Vector::size( ) {
   return sqrt( x * x + y * y + z * z );
}

bool Vector::normalize( ) {
   double s = size( );
   if( s == 0.0 ) return false;
   x /= s;
   y /= s;
   z /= s;
   return true;
}

void Vector::print( ) {
   std::cout << "( " << x << ", " << y << ", " << z << " )";
}

Vector & Vector::operator = ( Vector & v ) {
   x = v.x;
   y = v.y;
   z = v.z;
   return * this;
}

std::ostream & operator << (std::ostream & os, Vector & v ) {
   os << "( " << v.x << ", " << v.y << ", " << v.z << " )";
   return os;
}

Vector Vector::operator - ( Vector & v ) {
   return Vector( x - v.x, y - v.y, z - v.z );
}
```

```
/*

      Rectangle.h

*/
#ifndef __RECTANGLE_H
#define __RECTANGLE_H

#include "Vector.h"

class Ray;

class Rectangle {

public:
   Rectangle( Vector & pposition, Vector & pwidth, Vector & pheight );
   bool isCrossing( Ray * pray );

private:
   Vector position;
   Vector width;
   Vector height;

};


#endif
```

```
/*

      Rectangle.cpp

*/

#include "Rectangle.h"
#include "Ray.h"

Rectangle::Rectangle( Vector & pposition, Vector & pwidth, Vector & pheight ) :
   position( pposition ), width( pwidth ), height( pheight ) {

}

bool Rectangle::isCrossing( Ray * pray ) {

   Vector v = position - pray -> getPosition( );
   Vector r = pray -> getDirection( );
   std::cout << "v: " << v << " r: " << r << std::endl;
   double rx = r.getX( );
   double ry = r.getY( );
   double rz = r.getZ( );
   double vx = v.getX( );
   double vy = v.getY( );
   double vz = v.getZ( );
   double sx = width.getX( );
   double sy = width.getY( );
   double sz = width.getZ( );
   double tx = height.getX( );
   double ty = height.getY( );
   double tz = height.getZ( );
   double k;
   double l;
   double m;

   double a1 = rx * ty - ry * tx;
   double a2 = rx * tz - rz * tx;
   double a3 = ry * tz - rz * ty;
   double b1 = sx * ty - sy * tx;
   double b2 = sx * tz - sz * tx;
   double b3 = sy * tz - sz * ty;
   double c1 = vx * ty - vy * tx;
   double c2 = vx * tz - vz * tx;
   double c3 = vy * tz - vz * ty;

   std::cout << "a1:" << a1 << " a2:" << a2
      << " b1:" << b1 << " b2:" << b2
      << " c1:" << c1 << " c2:" << c2 << std::endl;
   double d1 = a1 * b2 - a2 * b1;
   double d3 = a1 * b3 - a3 * b1;
   std::cout << "d1 = " << d1 << " d3 = " << d3 << std::endl;

   if( d1 != 0.0 ) {
     k = ( c1 * b2 - c2 * b1 ) / d1;
     l = ( c1 * a2 - c2 * a1 ) / d1;
     if( tx != 0.0 )
       m = ( k * rx - l * sx - vx ) / tx;
     else if( ty != 0.0 )
       m = ( k * ry - l * sy - vy ) / ty;
     else if( tz != 0.0 )
       m = ( k * rz - l * sz - vz ) / tz;
     else
       return false;
   }
   else if( d3 != 0.0 ) {
     k = ( c1 * b3 - c3 * b1 ) / d3;
     l = ( c1 * a3 - c3 * a1 ) / d3;
     if( tx != 0.0 )
       m = ( k * rx - l * sx - vx ) / tx;
     else if( ty != 0.0 )
       m = ( k * ry - l * sy - vy ) / ty;
     else if( tz != 0.0 )
       m = ( k * rz - l * sz - vz ) / tz;
     else
       return false;
   }

   if( l >= 0.0 && l <= 1.0 &&
      m >= 0.0 && m <= 1.0 )
      return true;
   return false;
}
```

```
/*

      Ray.h

*/
#ifndef __RAY_H
#define __RAY_H

#include "Vector.h"

class Ray {

public:
   Ray( Vector & pposition, Vector & pdirection );
   Vector & getPosition( ) { return position; }
   Vector & getDirection( ) { return direction; }
private:
   Vector position;
   Vector direction;

};

#endif
```

```
/*

      Ray.cpp

*/
#include "Ray.h"

Ray::Ray( Vector & pposition, Vector & pdirection ) :
   position( pposition ), direction( pdirection ) {

}
```

本当はこれでは一様に振ったことにならないのだが。

```
/*

      Source.h

*/
#ifndef __SOURCE_H
#define __SOURCE_H

#include "Vector.h"

class Ray; // forward declaration

class Source {

public:
   Source( Vector & pposition );
   Ray * emit( );
   // this will 'new' an instance of Ray
private:
   Vector position;
   long int seed;
};


#endif
```

```
/*

      Source.cpp

*/
#include "Source.h"
#include "Ray.h"
#include <cstdlib>

Source::Source( Vector & pposition ) :
   position( pposition ) {

   seed = 1234567;
   srand48( seed );
}

Ray * Source::emit( ) {
   double x = 2.0 * drand48( ) - 1.0;
   double y = 2.0 * drand48( ) - 1.0;
   double z = 2.0 * drand48( ) - 1.0;
   Vector v( x, y, z );
   v.normalize( );
   return new Ray( position, v );

}
```

```
g++ -c Vector.cpp
g++ -c Rectangle.cpp
g++ -c Ray.cpp
g++ -c Source.cpp
g++ -o main main.cpp Vector.o Rectangle.o Ray.o Source.o
./main
```

```
/*

    main.cpp

*/

#include <iostream>
#include "Source.h"
#include "Ray.h"
#include "Rectangle.h"

int main( ) {

  Ray * pray;
  Vector source_position( 0.0, 0.0, 0.0 );
  Source source( source_position );

  Vector sensor_position( 0.0, 0.0, 0.5 );
  Vector sensor_width( 1.0, 0.0, 0.0 );
  Vector sensor_height( 0.0, 1.0, 0.0 );
  Rectangle sensor( sensor_position,
    sensor_width, sensor_height );

  int nhit = 0;
  int ntotal = 1000;

  for( int i = 0; i < ntotal; i ++ ) {
    pray = source.emit( );
    if( sensor.isCrossing( pray ) )
      nhit ++;
    delete pray;
  }

  std::cout << "Hits " << nhit << " among "
    << ntotal << " trys" << std::endl;

}
```

クラスの継承

- 基底クラス
- 継承
- メンバーオーバーライド
- メンバーのスコープ

- 共通のメンバーを持つクラスのグループ化
  - メンバーデータが共通
  - メンバー関数が共通
    - 実装まで同じもの
    - 実装が異なるもの
- まずすべてが共通のものを取り出す。
  - 基底クラス

継承の例

```
class Box {
public:
    Box( double w, double h,
    double d );
    void move( const Vector & p );
    void draw( );
private:
    double width;
    double height;
    double depth;
    Vector position;
};
```

```
class Ball {
public:
    Ball( double r );
    void move( const Vector & p );
    void draw( );
private:
    double radius;
    Vector position;
};
```

継承の例（続き）

```
class Shape {
public:
    Shape( );
    void move( const Vect &
    p );
protected:
    Vect position;
};
```

```
class Ball : public Shape {
public:
    Ball( double r );
    void draw( );
private:
    double radius;
};
```

継承時のコンストラクタとデストラクタ

- コンストラクタは基底クラスのコンストラクタが呼ばれてから継承クラスのコンストラクタが呼ばれる。
  - Ball ball(10.0);　　//Shape::Shape()に続いてBall::Ball(double)が呼ばれる。
- デストラクタは逆に、継承クラスのデストラクタが呼ばれてから基底クラスのデストラクタが呼ばれる。

作業20

- Rectangleをコピーしてを作る。
- Ellipse.h
- Ellipse.cpp

isCrossingのメソッドの最後

```
    if( l * l + m * m < 1.0 )
        return true;
    return false;
}
```

```
/*

      Ellipse.h

*/
#ifndef __ELLIPSE_H
#define __ELLIPSE_H

#include "Vector.h"

class Ray;

class Ellipse {

public:
    Ellipse( Vector & pposition, Vector & pwidth, Vector &
pheight );
    bool isCrossing( Ray * pray );

private:
    Vector position;
    Vector width;
    Vector height;

};


#endif
```

作業20の続き

- Makefileを書く (頭の空白はtab文字)
- gmake

```
CC = g++
AR = ar
OBJECTFILES = Vector.o Ray.o Rectangle.o Ellipse.o Source.o

.cpp.o:
	${CC} -c $<

main: main.o libapp.a
	${CC} -o main main.o -L. -lapp

libapp.a: ${OBJECTFILES}
	${AR} r libapp.a ${OBJECTFILES}
```

作業21

- EllipseとRectangleの基底クラスSurfaceを作る。

```
/*

      Surface.h

*/
#ifndef __SURFACE_H
#define __SURFACE_H

#include "Vector.h"

class Ray;

class Surface {

public:
   Surface( Vector & pposition, Vector & pwidth, Vector &
pheight );

protected:
   bool testCrossing( double & k, double & l, double & m, Ray
* pray );
   Vector position;
   Vector width;
   Vector height;

};


#endif
```

```
/*

      Surface.cpp

*/

#include "Surface.h"
#include "Ray.h"

Surface::Surface( Vector & pposition, Vector & pwidth, Vector &
pheight ) :
   position( pposition ), width( pwidth ), height( pheight ) {

}

bool Surface::testCrossing( double & k, double & l, double & m, Ray *
pray ) {

   Vector v = position - pray -> getPosition( );
   Vector r = pray -> getDirection( );
   double rx = r.getX( );
   double ry = r.getY( );
   double rz = r.getZ( );
   double vx = v.getX( );
   double vy = v.getY( );
   double vz = v.getZ( );
   double sx = width.getX( );
   double sy = width.getY( );
   double sz = width.getZ( );
   double tx = height.getX( );
   double ty = height.getY( );
   double tz = height.getZ( );

   double a1 = rx * ty - ry * tx;
   double a2 = rx * tz - rz * tx;
   double a3 = ry * tz - rz * ty;
   double b1 = sx * ty - sy * tx;
   double b2 = sx * tz - sz * tx;
   double b3 = sy * tz - sz * ty;
   double c1 = vx * ty - vy * tx;
   double c2 = vx * tz - vz * tx;
   double c3 = vy * tz - vz * ty;

   double d1 = a1 * b2 - a2 * b1;
   double d3 = a1 * b3 - a3 * b1;

   if( d1 != 0.0 ) {
     k = ( c1 * b2 - c2 * b1 ) / d1;
     l = ( c1 * a2 - c2 * a1 ) / d1;
     if( tx != 0.0 )
       m = ( k * rx - l * sx - vx ) / tx;
     else if( ty != 0.0 )
       m = ( k * ry - l * sy - vy ) / ty;
     else if( tz != 0.0 )
       m = ( k * rz - l * sz - vz ) / tz;
     else
       return false;
   }
   else if( d3 != 0.0 ) {
     k = ( c1 * b3 - c3 * b1 ) / d3;
     l = ( c1 * a3 - c3 * a1 ) / d3;
     if( tx != 0.0 )
       m = ( k * rx - l * sx - vx ) / tx;
     else if( ty != 0.0 )
       m = ( k * ry - l * sy - vy ) / ty;
     else if( tz != 0.0 )
       m = ( k * rz - l * sz - vz ) / tz;
     else
       return false;
   }

   return true;
}
```

```
/*

      Rectangle.h

*/
#ifndef __RECTANGLE_H
#define __RECTANGLE_H

#include "Vector.h"
#include "Surface.h"

class Ray;

class Rectangle : public Surface {

public:
   Rectangle( Vector & pposition, Vector & pwidth, Vector &
pheight );
   bool isCrossing( Ray * pray );

private:

};


#endif
```

```
/*

      Rectangle.cpp

*/

#include "Rectangle.h"
#include "Ray.h"

Rectangle::Rectangle( Vector & pposition, Vector &
pwidth, Vector & pheight ) :
   Surface( pposition, pwidth, pheight ) {

}

bool Rectangle::isCrossing( Ray * pray ) {

   double k;
   double l;
   double m;

   if( ! testCrossing( k, l, m, pray ) )
      return false;

   if( l >= 0.0 && l <= 1.0 &&
      m >= 0.0 && m <= 1.0 )
      return true;
   return false;
}
```

```
/*

      Ellipse.h

*/
#ifndef __ELLIPSE_H
#define __ELLIPSE_H

#include "Vector.h"
#include "Surface.h"

class Ray;

class Ellipse : public Surface {

public:
   Ellipse( Vector & pposition, Vector & pwidth, Vector &
pheight );
   bool isCrossing( Ray * pray );

private:

};


#endif
```

```
/*

      Ellipse.cpp

*/

#include "Ellipse.h"
#include "Ray.h"

Ellipse::Ellipse( Vector & pposition, Vector & pwidth, Vector
& pheight ) :
   Surface( pposition, pwidth, pheight ) {

}

bool Ellipse::isCrossing( Ray * pray ) {

   double k;
   double l;
   double m;

   if( ! testCrossing( k, l, m, pray ) )
      return false;

   if( l * l + m * m < 1.0 )
      return true;
   return false;
}
```

- Makefileを更新

```
#
#       Makefile
#
CC = g++
AR = ar
OBJECTFILES = Vector.o Ray.o Surface.o Rectangle.o Ellipse.o Source.o

.cpp.o:
	${CC} -c $<

main: main.o libapp.a
	${CC} -o main main.o -L. -lapp

libapp.a: ${OBJECTFILES}
	${AR} r libapp.a ${OBJECTFILES}

Vector.o : Vector.h Vector.cpp

Ray.o : Vector.h Ray.h Ray.cpp

Surface.o : Vector.h Ray.h Surface.h Surface.cpp

Rectangle.o : Vector.h Ray.h Surface.h Rectangle.h Rectangle.cpp

Ellipse.o : Vector.h Ray.h Surface.h Ellipse.h Ellipse.cpp

Source.o : Vector.h Ray.h Source.h Source.cpp
```

仮想継承

- 仮想継承クラスの定義
  - 仮想メンバー関数を持つ
  - 名前、引数、戻り値が同じだが実装が異なる
  - 基底クラスで仮想メンバー関数を定義しておく
    - 継承クラスで定義されていなければそれが使われる
    - 継承クラスで定義があればそれを使う。
- 仮想継承クラスオブジェクト

早い結合と遅い結合

仮想でない継承

- ポインター型が親クラス型だと親クラスのオブジェクトとして使われる。
- ポインター型が子クラス型だと子クラスのオブジェクトとして使われる。
- コンパイル時に確定している。(型が決まっている)
- Early binding(早い結合)

仮想継承

- ポインター型が親クラス型であってもオブジェクトが子クラスのものである場合、子クラスのオブジェクトとして取り扱われる。
- ポインターに何が代入されるかは実行時までわからない。実行時に初めて型が決まる。
- Late binding(遅い結合)

仮想継承の例

```
class Shape {
public:
    Shape( );
    void move( const Vect &
    p );
    virtual void draw( );
protected:
    Vect position;
};
```

```
class Ball: public Shape {
public:
    Ball( double r );
    void draw( );
private:
    double radius;
};
```

Shapeのオブジェクトも作れる。役には立たないので作ってほしくない。

仮想継承の利用

- 仮想クラスオブジェクトを基底クラスポインターで扱う

```
Shape * shapelist[10];
    shapelist[0] = new Ball( 1.0 );
    shapelist[1] = new Box( 1.0, 2.0, 3.0 );
    …
    for( int i = 0; i < n; i ++ ) {
        shapelist[i]->draw( );
    }
```

```
/*

      Surface.h

*/
#ifndef __SURFACE_H
#define __SURFACE_H

#include "Vector.h"

class Ray;

class Surface {

public:
   Surface( Vector & pposition, Vector & pwidth, Vector &
pheight );
   virtual bool isCrossing( Ray * pray );

protected:
   bool testCrossing( double & k, double & l, double & m, Ray
* pray );
   Vector position;
   Vector width;
   Vector height;

};


#endif
```

作業22

- Surfaceに仮想メンバーisCrossingを追加する。

```
bool Surface::isCrossing( Ray * pray ) {

   double k;
   double l;
   double m;

   return testCrossing( k, l, m, pray );

}
```

Main.cppを修正する。
gmake
./main

```
/*

      main.cpp

*/

#include <iostream>
#include "Source.h"
#include "Ray.h"
#include "Rectangle.h"
#include "Ellipse.h"
```

```
int main( ) {

   Ray * pray;
   Vector source_position( 0.0, 0.0, 0.0 );
   Source source( source_position );

   Vector sensor_position( 0.0, 0.0, 0.5 );
   Vector sensor_width( 1.0, 0.0, 0.0 );
   Vector sensor_height( 0.0, 1.0, 0.0 );
   Surface * sensor = new Ellipse( sensor_position,
      sensor_width, sensor_height );

   int nhit = 0;
   int ntotal = 1000;

   for( int i = 0; i < ntotal; i ++ ) {
      pray = source.emit( );
      if( sensor -> isCrossing( pray ) )
         nhit ++;
      delete pray;
   }

   std::cout << "Hits " << nhit << " among "
      << ntotal << " trys" << std::endl;

}
```

仮想継承のデストラクタ

- 例の続き

```
for( int i = 0; i < n; i++ ) {
        delete shapelist[i];
}
```

- デストラクタが仮想的でないとShapeのデストラクタが呼ばれる。
- こういう場合デストラクタはvirtual宣言されるべきである。

継承クラスのコンストラクタ

- 継承クラスのオブジェクトは明示的にクラス名が与えられる。デストラクタのようにvirtualである必要はない。
- コンストラクタの呼び出し順
  - 基底クラスのコンストラクタが呼び出されてから継承クラスのコンストラクタが呼び出される。
  - 基底クラスの引数付きコンストラクタを呼ぶときは継承クラスコンストラクタの定義時の初期化リストに基底クラスコンストラクタを加える。
  - 継承クラスのコンストラクタ内部から基底クラスのコンストラクタを呼び出すと、2度呼び出すことになる。

キャスト

- 基底クラスのポインターがいずれかの継承クラスのオブジェクトを示していることがわかっているとする。
- ある継承クラスのポインターにその値を代入したい。
- 強引に代入すると何が起こるかわからない。
- 継承クラスポインター=dynamic_cast<基底クラス>(基底クラスオブジェクトへのポインター);
- 失敗すると0を返す。

純粋仮想関数

```
class Shape {
public:
    Shape( );
    void move( const Vector
    & p );
    virtual void draw( )=0;
protected:
    Vector position;
};
```

```
class Ball: public Shape {
public:
    Ball( double r );
    void draw( );
private:
    double radius;
};
```

Shapeのオブジェクトは作れない。Shapeの子クラスはdrawを必ず実装しなければならない。

抽象クラス

- 純粋仮想関数を一つでもメンバーに持てば抽象クラスとなる。それ自身のオブジェクトを作れない。
  - クラスの使用者(継承して使う)にメソッドの実装を強制できる。
- 任意の子クラスをアクセスするための方法を規定することになる。インターフェースと呼ばれることもある。
  - プログラムの枠組みはインターフェースを用いてかける。
  - 実装の局所化：プログラム開発の分業を可能に

作業23

- Surfaceを抽象クラスにする。
- Surface.hを修正
- Surface.cppからisCrossingの実装を消去。
- gmake
- ./main

```
/*

      Surface.h

*/
#ifndef __SURFACE_H
#define __SURFACE_H

#include "Vector.h"

class Ray;

class Surface {

public:
   Surface( Vector & pposition, Vector & pwidth, Vector &
pheight );
   virtual bool isCrossing( Ray * pray ) = 0;

protected:
   bool testCrossing( double & k, double & l, double & m, Ray
* pray );
   Vector position;
   Vector width;
   Vector height;

};


#endif
```

多重継承

- 複数の基底クラスを持つクラス
- 親のメンバーを全部受け継ぐ。
- 先祖のクラスが同一の場合、先祖のメンバーをそれぞれの親ごとに持つ。
  - メンバー名は同じだがスコープ演算子で指定が可能。
- 仮想継承では先祖のクラスが同一の場合先祖のメンバーを重複しては持たない。

多重継承の例

- 異なる種類の属性グループ
- Shapeという形状属性
- Mateiralという光学属性
- class Ball : public Shape {…};
- class Glass : public Material {…};
- class GlassBall : public Ball, public Glass {…};

仮想多重継承

- ShapeもMaterialもStreamable

```
class Streamable {
public:
    virtual int read( )=0;
    virtual int write( )=0;
protected:
   int classIdentifier;
};
```

仮想多重継承(続き)

- class Shape : virutal public Streamable {…};
- class Material : virtual public Streamable {…};
- class Ball : virtual public Shape {…};
- class Glass : virtual public Material {…};
- class GlassBall : virtual public Ball, virtual public Glass {…};
- Ball::classIdentifierとGlass::classIdentiferは同じものを指す。

スコープ(シンボルの可視範囲)

- 複文：{}の内側
- 関数：関数定義の{}の内側
- クラス：クラスメンバー
- ファイル：ファイル内にないものはextern宣言
- 名前空間：namespaceで定義
- 大域変数や大域関数

近いもの(内側から探して早く見つかったもの)を使う。

スコープの例

```
int func( int iarg ) {
    int iarg;     //これはだめ。
    int i=10;
    if( iarg > 0 ) {
        int i = 20;
        for( int i=0; i < iarg; i ++ ) {
            int      i = 30;
            {
                int      i=40;
```

- 無論、マナーとしてこのような使い方をしてはいけない。

ネームスペース

- ネームスペースの宣言

```
namespace MySpace {
   class MyClass {…}
}
```

- ネームスペースの外からの利用
  - スコープ演算子を使う
    - MySpace::MyClass myobject;
  - using namespace宣言をする
    - using namespace MySpace;
- 大域変数・大域関数は無名名前空間にある
  - ::errnoなどと、名前なしでスコープ演算子::を使う

テンプレート

- 型をパラメータとする機能
- テンプレート関数
- テンプレートクラス
- テンプレートの利用：インスタンシエーション

パラメータ型

- template<class T>
- 以後Tをクラス名と同様に使える。
  - 必ずしもclassではなくても良い。intやdoubleなども。

```
template<class T> class Queue {
    T * get( int index );
    int  set( int index, T * object );
    …
    T    store[SIZE];
};
```

関数テンプレート

```
template<class T> T tmax(T t1, T t2) {
    if( t1 > t2 ) return t1;
    else return t2;
}
```

- これを使う。

```
double x = 10.0, y=20.0, z;
    z = tmax<double>(x,y);
int ix = 10, iy=20, iz;
    iz = tmax<int>(ix,iy);
```

クラステンプレート

```
template<class T> class Tlist {
public:
    Tlist( int size );
    ~Tlist( );
    int put( int index, T * value );
    T * get( int index );
protected:
    T ** list;
    int listsize;
};
```

メンバー関数の定義

```
template<class T> Tlist<T>::Tlist( int size ) {
    list = new T*[size];
    listsize = size;
}
template<class T> Tlist<T>::~Tlist( ) {
    delete [] list;
}
template<class T> int Tlist<T>::put( int index, T* value ) {
    list[index]=T;
    return index;
}
template<class T> T * Tlist<T>::get( int index ) {
    return list[index];
}
```

インスタンシエーション

- テンプレートを使ってクラスを作る。

```
#include “Tlist.h”

    Tlist<int> ilist;
    Tlist<double> dlist;
    Tlist<MyClass> mlist;
    mlist.put(0, new MyClass());
```

標準テンプレートライブラリ

- std名前空間に定義されている。
  - #include <ライブラリ名>
  - 従来のCのライブラリは無名名前空間にあり、.hファイルで定義されている。
- 文字列
- 入出力
- 数学関数等
- コンテナ

STL: 文字列

- C言語ではchar型配列として文字列を扱った。
  - 終端文字'¥0'までを文字列ととらえる。
  - 文字列のコピーなどはバグの元凶で危険。
  - C++では安全な文字列を別に定義している。
- #include <string>
  - std名前空間内に定義されている。
- std::string

STL文字列の例

```
std::string s(“Hello world”);  //変数の定義C型文字列で初期化
std::string t;                 //空で定義
t = s + “ of c++”;             //結合と代入
std::string u = “12345”;
int i = atoi(u.c_str());       //C型の文字列への変換
char c = u[3];                 //文字の取り出し
t.append(u);                   //文字列の結合の別の形
t.insert(10,”---”);            //文字列の挿入
int ipos = t.find(“++”);       //文字列の検索
t.substr(ipos);                //部分文字列の取り出し
int length = t.size( );        //文字列の長さ
```

STL:入出力

- #include <iostream>
  - std名前空間に定義されている。
- 標準入出力
  - std::cin (stdinに対応)
  - std::cout (stdoutに対応)
  - std::cerr (stderrに対応)
- 使い方

```
std::cout << “Hello world!” << std::endl;
std::string command;
std::cout << “Command: “;
std::cin >> command;
std::cout << “Command is : “ << command << std::endl;
```

整数の出力書式

- 出力幅

```
int       i;
std::cin >> i;
std::cout.width(10);
std::cout << i << std::endl;
```

- マニピュレータを使う

```
#include <iomanip>
std::cout<<std::setw(10)<<i<<std::endl;
```

- 進法

```
std::cout<<std::hex<<i<<“ “<<std::oct<<i<<“ “<<std::dec<
    <i<<std::endl;
```

浮動小数点数の出力書式

- 出力形式

```
double d;
std::cout << std::fixed << d << “ “ << std::scientific <<
    d << std::endl;
```

- 出力精度

```
std::cout.precision(8);            //小数点以下8桁
std::cout << std::fixed << d << “ “ << std::scientific <<
    d << std::endl;
std::cout << std::setprecision(8) << d;       //これも可
```

ファイルからの入力

```
#include <fstream>
std::ifstream f(“inputfile.dat”);
if(!f) { std::cout<<“Could not open.” << std::endl;
return 1; }
int i;
f >> i;
std::string s;
f >> s;
```

ファイルからの一行入力

```
#include <fstream>
std::ifstream f(“inputfile.dat”);
char buf[256];
while( true ) {
    f.getline( buf, 256 );
    if( f.eof( ) ) break;
    std::cout << buf << std::endl;
}
```

文字列からの入力

```
#include    <sstream>
std::istringstream s(buf);     //bufは前のページ
int i;
s>>i;
std::string ss;
s>>ss;
```

数学関数関係

- 数値表現の最大最小

```
#include <limits>
numeric_limits<double>::min()
numeric_limits<double>::max()
```

数学関数関係2

#include <cmath>

double d, e;

sqrt(d), pow(d,e)

sin(d), cos(d), tan(d)

asin(d), acos(d), atan(d), atan2(d,e)

sinh(d), cosh(d), tanh(d)

exp(d), log(d), log10(d)

ceil(d), floor(d), abs(d), fabs(d)

数学関数(3)

- 複素数

```
#include <complex>
std::complex<double> d;
```

四則演算、数学関数も定義されている。

- 乱数

```
#include <cstdlib>
long   long_seed = 123456;
srand48(long_seed);
double d = drand48();
```

STL:コンテナクラス

- コンテナ
  - <vector>、<list>、<queue>、<map>など
- 反復子
  - vector<T>::iterator

std::vector

```
#include <vector>
using namespace std;
vector<int> iv;
for(int i =0; i<10; i++ ) iv.push_back( i*2 );
cout << iv.size( ) << endl;
for( int i = 0; i<iv.size( ); i++ ) {
   cout << iv[i] << ‘ ‘;    }
cout << endl;
```

std::vector iterator

続き

```
iv.pop_back();    //
vector<int>::const_iterator it=iv.begin( );
while( it != iv.end( ) ) {
   cout << * it ++ << ‘ ‘;        }
cout << endl;
vector<int>::iterator jt=iv.begin( );
//今度は修正可能
```

例外処理

- throwとcatch：例外の受け渡し
- 例外の定義
- tryとcatch
- throw宣言

throwとcatch

- メンバー関数などで例外を発見する。
  - 0で割り算をする。配列の引数が範囲外である。コマンドの文法違反。などなど。
  - それに対する対処はその関数自身は知らない。呼び出した側に任せるしかない。
- 例外の受け渡しの仕組み
  - メンバー関数側は「例外」をthrowする。
  - 利用者側はtry句内でメンバー関数を呼び出し、catch句で受ける。
  - 受けるcatch句が無いときはstd::terminate()が呼ばれる。

throwとcatchの例

- 例外を定義する。

```
class RangeError { };
```

- 例外を投げる

```
Shape * ShapeList::get( int index ) {
    if( index < 0 || index >= size ) throw RangeError( );
```

- 例外を受け取る

```
ShapeList sl;
try {
    Shape * p;
    p = sl.get( i );
    //ここは正常処理
}
catch( RangeError ) {
    //エラーがあればここに来る。
}
```

例外の定義

- 例外は型で区別される。ユーザ型を使う。

```
class Error {    };
class ZeroDivide : public Error {   };
class RangeError : public Error {
public:        int index;  };
class SyntaxError : public Error {
public:        const char * statement;   };
```

例外の受け取り

```
try {
    …
}
catch( ZeroDivide ) {
    std::cerr << “Divide by zero” << std::endl;
}
catch ( RangeError e ) {
    std::cerr << “Range error : “ << e.index << std::endl;
}
catch ( SyntaxError e ) {
    std::cerr << “Syntax error at “ << e.statement << std::endl;
}
catch ( Error e ) {
    std::cerr << “Undefined error” << std::endl;
}
```

throw宣言

```
class Thrower {
public:
    void set( int index, double * values ) throw
    ( RangeError, ZeroDivide );
};
```

CLHEP

- 高エネルギー物理用クラスライブラリ
- http://proj-clhep.web.cern.ch/proj-clhep/
- Hep3Vector
  - #include “CLHEP/Vector/ThreeVector.h”
- HepLorentzVector
  - #include “CLHEP/Vector/LorentzVector.h”

ROOT

- C++文法に基づいたデータ解析プラットフォーム
- http://root.cern.ch/drupal/
- ヒストグラムやnTupleなどデータ解析に必要な機能を満載。

コーディングルール

- クラス名とnamespace
  - 大文字ではじめる。単語の頭を大文字で他は小文字で書く。
- メンバー名
  - 小文字ではじめる。二つ目以後の単語の頭は大文字で書く。
- ファイルの構成
  - 一つのクラスは1つのファイルで定義する。
  - ファイル名はクラス名と同じにする。
  - ヘッダーファイルは.h、ソースファイルは.cpp/.cxx/.C

Makefile

- プログラムの作り方を書く。プロジェクトの一部。

```
CC=g++
demo: demo.o
    $(CC) –o demo demo.o
```

- demoという実行可能形式を作る方法が書かれている。段下げは空白ではなくてタブ文字が使われる。
- demo.oの作り方は書かれていないが、.Cから.oを作る方法をmakeが知っているので書かなくてもできる。依存するファイルが他にあるときは

```
demo.o: demo.C MyClass.h
```

Makefile続き

- たくさんのオブジェクトファイルを使うときはライブラリを作る

```
CC = g++
AR = ar
demo: demo.o libDemo.a
    $(CC) –o demo demo.o –lDemo

libDemo.a:  Shape.o Box.o Ball.o Cylinder.o
    $(AR) –r libDemo.a Shape.o Box.o Ball.o
    Cylinder.o
```

クラス設計

- オブジェクト指向解析・設計(OOA/OOD)
  - 問題領域(Problem Domain)
- クラスの発見
  - 登場人物(オブジェクト)
  - 登場人物間のやりとり(インタラクション)
  - 登場人物の共通性(クラス)
  - 登場人物の振るまい(メソッド)
  - 登場人物の属性(アトリビュート)
  - 登場人物の利用者(アクター)

課題の検討

- 問題領域
  - 光源、光センサー、鏡、その他の光学系
  - 三次元シミュレーション
- 登場人物と振る舞い
  - 幾何学形状(光源、センサー、鏡...)
    - 光の放出、反射、屈折、吸収
  - 光(ビーム)

オブジェクトの関係

- 所有(属性)
  - メンバーデータとして持つ関係
- 継承
  - 親クラスとして持つ関係
- 関連
  - メソッドの引数を通して持つ関係

クラスの発明

- オブジェクトの発見
  - プログラム上の存在(相互作用)
  - オブジェクトの共通性
- クラスの発明
  - 他との関連
  - 振る舞い
  - 属性

ソフトウエア開発

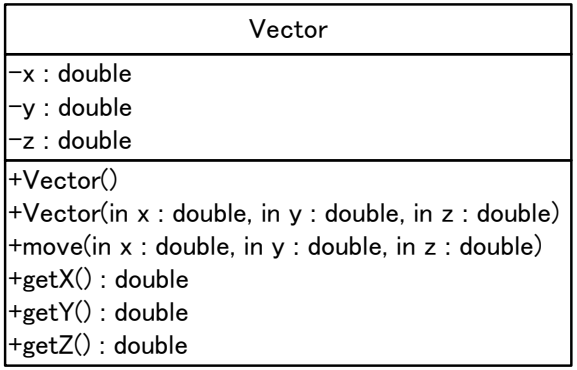
- ソフトウエアの仕様と実装
  - 利用者は仕様を知っていれば使える。
  - 開発者は仕様を守って実装する。

UML (Unified Modeling Language)

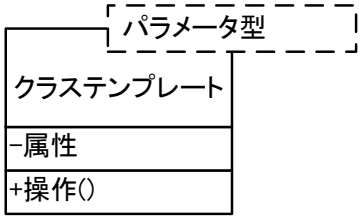
- モデリング
  - プログラムへの要請、プログラムの振る舞い
  - プログラムの構成、制御の流れ
  - クラスとその間の関係
- ダイアグラム(図)で表現
  - クラス図、パッケージ図、コンポジット図、コンポーネント図、配置図
  - ユースケース図、状態マシン図、アクティビティ図
  - 相互作用図

クラス図

- クラスの記述


Vector
−x : double
−y : double
−z : double
+Vector()
+Vector(in x : double, in y : double, in z : double)
+move(in x : double, in y : double, in z : double)
+getX() : double
+getY() : double
+getZ() : double

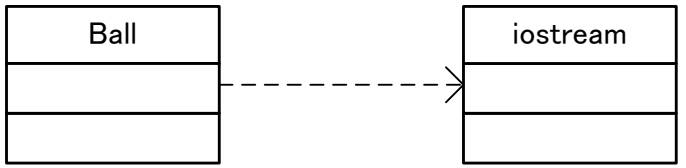

クラス名
−private属性
+public属性
#protected属性
+public操作()
#protected操作()
−private操作()


パラメータ型
クラステンプレート
−属性
+操作()

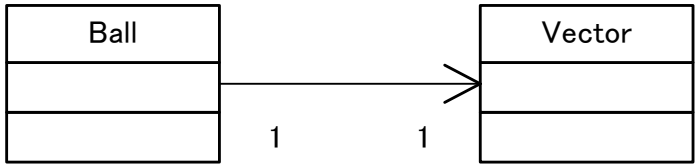
クラス図

- クラスの関係

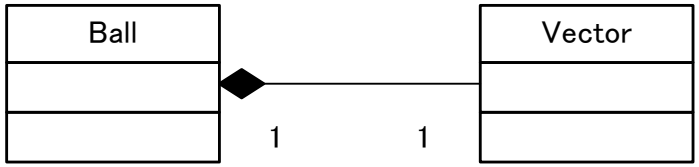
依存



関連



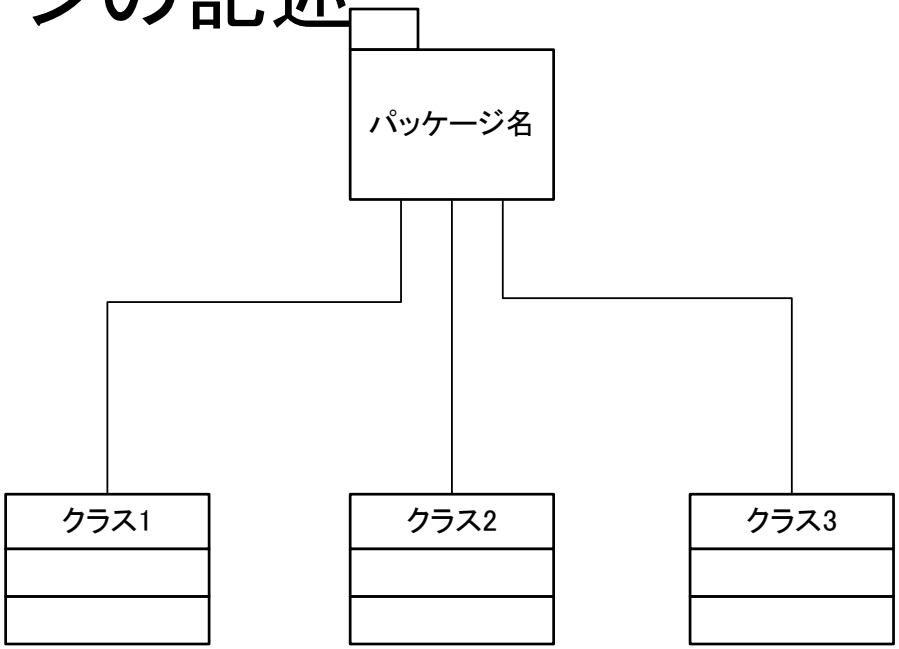
集約



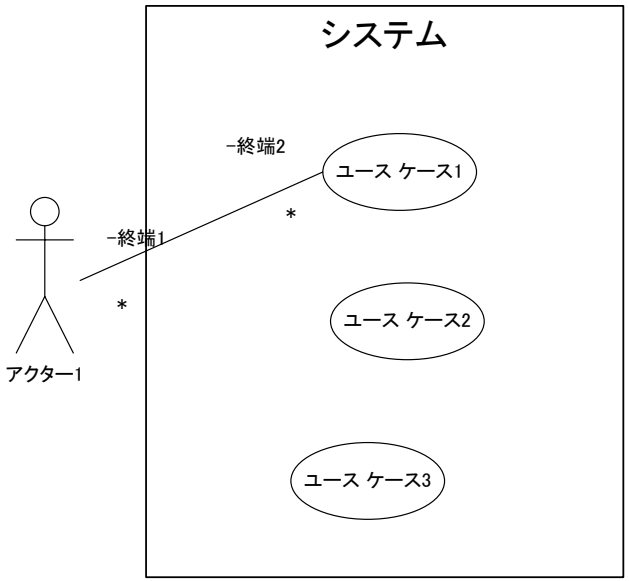
汎化

パッケージ図

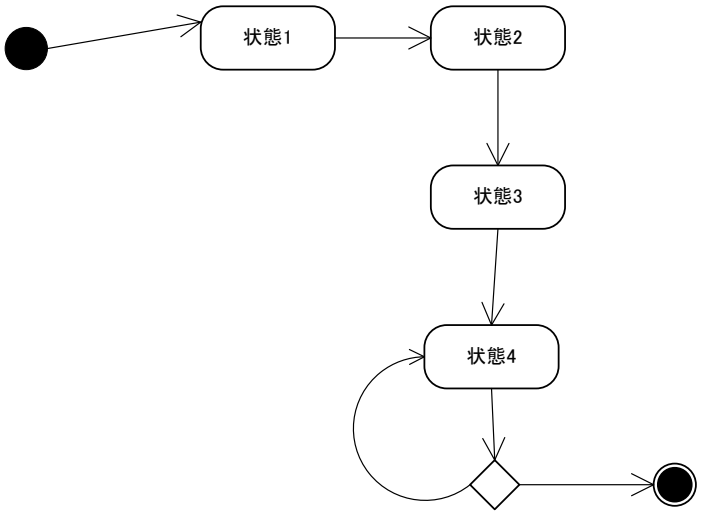
- パッケージの記述



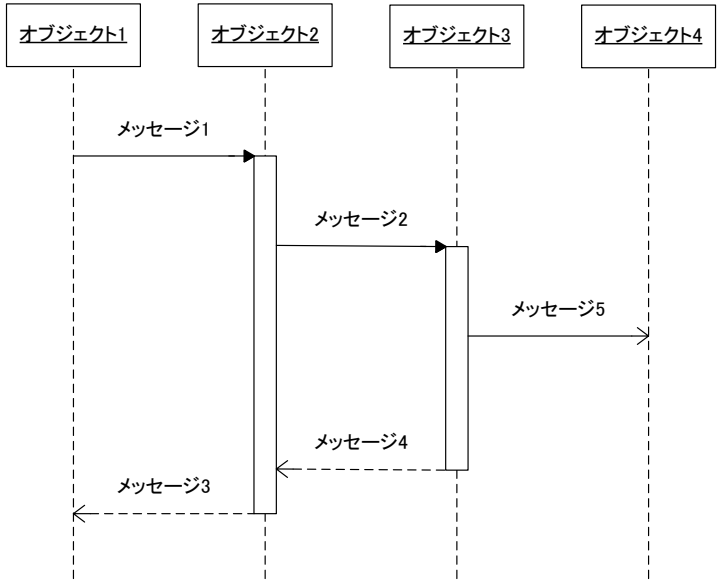
ユースケース図



状態マシン図



相互作用図


オブジェクト1
オブジェクト2
オブジェクト3
オブジェクト4
メッセージ1
メッセージ2
メッセージ5
メッセージ4
メッセージ3

オブジェクト指向技術の限界

- クラスは使われることを前提として設計されている。
  - メソッドやインターフェースの提供
  - 使う側を規定する構文を持たない
- クラスはプログラムプロセスとしてのみ存在する。
  - プロセス終了と同時に消滅する。
  - オブジェクトの保存と読み出し、オブジェクトのプロセス間共有、リモートオブジェクト間通信

ソフトウエア技術の進化

- たくさんのデータ
  - 構造化データへの進化(データのグループ化)
- たくさんのコード
  - 構造化プログラミングへの進化(コードのグループ化)
- たくさんの関数
  - クラスの導入(関数のグループ化)
- たくさんのクラス
  - クラス継承の導入(クラスの(一部の)グループ化)
- たくさんのメソッド
  - ？(メソッドのグループ化？)
- たくさんの関連
  - ？(関連のモジュール化、アスペクト指向)
- たくさんの継承
  - ？？

グループ化

- 関係のあるものをひとくくりにする。
- 関係のないものを切り離す。

クラス設計の例

- 図形を扱う
  - 形状(球、円筒、直方体等)
  - 形による違いと共通点
    - 共通点=親クラスの属性
    - 違い=子クラスの属性
  - 描画
  - 位置関係
    - 交わるかどうか
    - 手前にあるかどうか

クラス設計の演習

- 課題
  - 数値シミュレーション
  - 光源と光センサー
  - 光源の光のうちどれだけがセンサーに検出されるか。
  1. 光源とセンサーの位置を変えてみる。
  2. 鏡をおいてみる。どのように置けばどうなる。
  3. レンズをおいてみる。どのように置けば…
  4. 光源は直方体もしくは円筒の内部一様に分布している。その光源を効率よく集める光学系を設計する。

課題の進め方

- 最終目標まで含めてユースケースを考える。
- 結果を求める手順を考える。状態マシン図を
- 登場人物(オブジェクト)を決める。
- オブジェクトの相互作用を考える。
- クラスを定義する。
- 実装する。

考え得る登場人物

- 光ビーム
  - ある点からある点へ向かう
- 光源
  - 光ビームを発射する～どこからどちら向きに？
- 光センサー
  - 有感領域：幾何学形状
- 鏡
  - 形状：平面鏡・球面鏡など