2024 機械情報夏学期
ソフトウェア第二

担当：岡田 慧 (k-okada-soft2@jsk.t.u-tokyo.ac.jp)

6. オブジェクト指向プログラミング

1　オブジェクト指向復習

今日の講義ではオブジェクト指向について勉強する．オブジェクト指向は大規模なソフトウェア開発に必要不可欠な概念である．たとえばロボット等の代表される複雑なシステムのソフトウェアでは，数多くのソースと複数の人が開発に関わるが，

1. 他の開発者が書いた関数を使っても安全であること，
2. コードが他の開発者や自分にとってわかりやすいこと，
3. 他の開発者や自分が書いたコードを再利用できるようにすること，

が重要になる．オブジェクト指向はこのようなことを実現するための考え方であり，この考えに基づいた言語をオブジェクト指向言語と呼ぶ．近年利用されている言語は全てオブジェクト指向言語であるといっても過言ではない．

1.1　プログラミング言語の歴史

1.1.1　低級言語

まずは簡単にプログラミング言語の歴史を振り返ってみよう．

1950 年代前半，UNIVAC や IBM701 等の初期のコンピュータでは CPU が直接理解し実行するプログラミング言語である機械語 (またはマシン語と呼ぶ）でプログラミングがなされていた．機械語の実態は 2 進数や 16 進数であらわされており，コンピュータがそのまま実行できる形式である．たとえばウインドウズでは.exe という拡張子がついているファイルは全て機械語である．したがって CPU のアーキテクチャに互換性がなければある CPU 用に作られた機械語プログラムを他の CPU 上で実行することはできない．

機械語の命令を人間が理解しやすい記号に置き換えたものをアセンブリ言語と呼ぶ．アセンブリ言語を機械語に変換することをアセンブルと呼び，それを行うプログラムのことをアセンブラと言う．また，機械語からアセンブリ言語を生成することを逆アセンブルと呼ぶ．アセンブラの言語は基本的に機械語と１対１で対応する．そのためコンパイラでは実現できない最適化や，コンパイラが対応していない CPU の命令などを直接呼び出したい場合，あるいは CPU の動作を制御する必要がある場合は現在でも頻繁に利用されている．

アセンブリ言語の文法は CPU のアーキテクチャに依存するため，他の CPU に移植することは難しい．したがって機械語は低級言語と呼ばれる．これに対して，CPU のアーキテクチャに依存しない言語を高級言語と呼ぶ．

1.1.2 高級言語

機械語，アセンブリ言語以外のものを高級言語と呼ぶ．プログラムは一度アセンブリ言語に変換され，さらに機械語に変換されて計算機で実行される．

初期の高級言語には FORTLAN, LISP, COBOL がある．

FORTRAN は 1954 年に IBM のジョン・バッカスにより考案された高級プログラミング言語．科学技術計算に向いた逐次型の手続き型言語．FORTRAN, FORTRAN66, FORTRAN77, Fortran90, Fortran95, Fortran2003, Fortran2008 となり現在も使われている．Fortran90 で構造化記述の導入，再帰呼び出し，文脈自由文法の導入，Fortran2003 ではオブジェクト指向の導入がなされるなど進化を続けている．

Lisp は 1958 年に MIT のジョン・マッカーシーによって考案され，人工知能のコミュニティで流行した．当初はいくつかの Lisp 方言が存在したが，1990 年代に Common Lisp として標準仕様が固まった．Lisp でオブジェクト指向を実現したものは CLOS(Common Lisp Object System) と呼ばれ，Common Lisp の一部として仕様が決まっている．

1950 年代に開発された COBOL は米国政府の事務処理用言語として開発された言語であり，自然言語（英語）に近い記述が可能なような文法を持っている．たとえば足し算 A = A + B は ADD B TO A と表記できる．

1950 年代後半にはヨーロッパで ALGOL が提案された．BNF 記法[1]で記述された文法や構造化プログラミング[2]，ブロック構造化[3]など現在にも通じる概念が導入されたが，コンパイラの実装の困難さや標準化作業の遅れからほとんど普及しなかった．

C 言語は 1972 年に AT&T ベル研究所のデニス・リッチーが，UNIX の移植性を高めるために開発した言語である．アセンブラとの親和性が高いためハードウェアに密着したコーディングが容易であり OS 等の低レベルの記述に広く使われている．そのため，低級言語により近い高級言語であるといわれている．C という名前は，その前に AT&T 研究所でケン・トンプソンが開発した B 言語を改良したものであり，B の次は C ということで名前がつけられた．

`http://www.levenez.com/lang/` にプログラミング言語の進化系統樹があるので見てみよう．

1.2 オブジェクト指向言語

計算機の性能向上により大規模なソフトウェアがかかれるようになるにつれて，開発コストが上昇し 1960-70 年代にはこれ以上システムが大規模になるとソフトウェアの開発が不可能になるとの予測からソフトウェア危機[4]という言葉が誕生した．

---

[1] Backus-Naur Form : 自由文脈文法を定義するために使うメタ言語．ジョン・バッカスとピーター・ナウアが考案．その後正規表現を導入した拡張 BNF が提案され XML の構文定義など広く使われている．C 言語の定義は `https://t.ly/FOI9` で見ることが出来る．

[2] 構造化プログラミングはプログラムの手続き（処理）をいくつかの単位にわけ，サブルーチン（関数）の集合により全体の振る舞いを記述する方法である．サブルーチンは逐次処理を行う順次，一定の条件が満たされている間処理を繰り返す反復，条件判定に元づき処理を選択する分岐を基本的な論理構造とし，1967 年にエドガー・ダイクストラにより提唱された．

[3] begin/end あるいは {} 等を用いた入れ子によるブロックの構造化

[4] 「(ソフトウェア危機の主たる原因は）マシンがますます強力になってきたことだ！ はっきり言ってしまえば，マシンさえなければプログラミングには何の問題もない．貧弱なコンピュータが数台あるだけだったなら，プログラミングは穏

このような背景に基づいてソフトウェアの再利用や部品化が意識され，ソフトウェア工学などの分野が生まれてきた．構造化プログラミングではサブルーチンにより再利用性を高め，部品化を促進する仕組みが提唱された (Pascal 1971)．これはプログラムを構成するデータ構造とアルゴリズムに関して，アルゴリズムに関する構造化がなされたことになる．一方，データの構造化としてデータの定義とそれを処理する手続きをまとめるモジュールという考え方 (Modula-2 1979) が導入され，さらに異なるデータに関して複数の実体（インスタンス）を生成，管理する機構としてクラスや継承，オブジェクトという概念が生じてきた (Simula 1962).

このような概念を整理して実装された言語が Smalltalk(1972), C++(1979) であり，その後開発された Objective-C(1983), Python(1990), Ruby(1993), Perl(1994), Java(1995), C#(2000), Go(2009), Rust(2010), 等の言語は全てオブジェクト指向言語である．

1.3 オブジェクト指向言語の特徴

オブジェクト指向では，アルゴリズムとデータ構造がひとつになったオブジェクトと呼ばれる存在がメッセージを受け取り，その内部のデータを書き換え，他のオブジェクトにメッセージを送ることで目的とする機能を実現している．

この機能を実現するため，一般的なオブジェクト指向言語では以下のような概念が実現されている．

- **カプセル化**
  関連する関数や変数を一つのオブジェクトにまとめること．外部からオブジェクト内の不必要な関数や変数に直接に直接アクセスできない用に隠蔽することで安全性を高めることができ，オブジェクトの実装の詳細を外部からは隠すことができるため，実装を後からでも容易に変更することができる．

- **継承**
  既存のクラスに基づいて新しいクラスを作成する機能のこと．新しいクラスは基本クラスの全ての関数と変数を利用することができ，さらに新しい関数や変数を追加することができる．

- **多態性（ポリモーフィズム）**
  同じ関数であっても，引数になるオブジェクトの型に応じて適切な関数が選択され処理されること．これにより，オブジェクトの型に関係なく同じ名前の項目を使用できるようになる．

2 オブジェクト指向言語によるデータの抽象化

2.1 Python オブジェクト指向によるデータの抽象化

前章（データの抽象化）で示したタプルによる有理数の作成プログラムは，Python のオブジェクト指向を用いると以下のように記述することが出来る．

やかな問題になる．しかし現在の我々は強大なコンピュータを所有しているため，プログラミングも同様に強大な問題となっているのだ」Edsger Dijkstra: The Humble Programmer

```
class Rational:
   def __init__(self, n, d):
      # g = gcd(n, d)  # gcd を定義し，この行を追加すると簡約できる．
      self.numer = n
      self.denom = d

   def add_rat(self, y):
      return Rational(self.numer*y.denom + y.numer*self.denom, self.denom*y.denom)

   def sub_rat(self, y):
      return Rational(self.numer*y.denom - y.numer*self.denom, self.denom*y.denom)

   def mul_rat(self, y):
      return Rational(self.numer*y.numer, self.denom*y.denom)

   def div_rat(self, y):
      return Rational(self.numer*y.denom, self.denom*y.numer)

   def equal_rat(self, y):
      return self.numer*y.denom == y.numer * self.denom

   def print_rat(self):
      print (str(self.numer) + "/" + str(self.denom))
```

これを例えばrat.pyというファイルにセーブし，Pythonインタプリタ上からexec(open("rat.py").read())として読み込み[5]，以下のように実行することが出来る．

```
>> one_half = Rational(1, 2)
>> one_half.print_rat()
1/2
>>> one_third = Rational(1, 3)
>>> one_half.add_rat(one_third).print_rat()
5/6
>>> one_half.mul_rat(one_third).print_rat()
1/6
>>> one_third.add_rat(one_third).print_rat()
6/9
```

2.1.1 演算子オーバーロード

オーバーライド／オーバーロードはオブジェクト指向の多態性を実現する機能の一つである．オーバーライドとは，親クラスで定義されたメンバ関数を，子クラスで再定義することであり，オーバーロードは同一の関数名でも引数の型，数，並び順が異なる関数（メンバ関数を含む）を複数定義出来る機能である．

演算子オーバーロードは，演算子に対して引数に応じて異なる計算を可能にするものである．これにより，有理数をオブジェクトとして扱い，オブジェクト同士の四則演算が可能になる．

[5]Python2 の場合は，execfile('rat.py') とする

```
class Rational:
   def __init__(self, n, d):
      # g = gcd(n, d)  # gcd を定義し，この行を追加すると簡約できる.
      self.numer = n
      self.denom = d

   def __add__(self, y):
      return Rational(self.numer*y.denom + y.numer*self.denom, self.denom*y.denom)

   def __sub__(self, y):
      return Rational(self.numer*y.denom - y.numer*self.denom, self.denom*y.denom)

   def __mul__(self, y):
      return Rational(self.numer*y.numer, self.denom*y.denom)

   # Python3 用のコード
   def __truediv__(self, y):
      # Python3 では，2/3 は 0 ではなく 0.66 を返します.
      # https://docs.python.org/ja/3/library/operator.html
      # 0 を返す 2//3 のオペレータは __floordiv__ です
      return Rational(self.numer*y.denom, self.denom*y.numer)

   # Python2 用のコード
   def __div__(self, y):
      # __future__.division が有効でなければ， a / b は a // b と同じ結果を返します.
      # これは "古典的な (classic)" 除算とも呼ばれます.
      # https://docs.python.org/ja/2.7/library/operator.html
      return Rational(self.numer*y.denom, self.denom*y.numer)

   def __equal__(self, y):
      return self.numer*y.denom == y.numer * self.denom

   def __repr__(self):
      return (str(self.numer) + "/" + str(self.denom))
```

```
>>> exec(open('rat2.py').read())
>>> one_half = Rational(1, 2)
>>> one_half
1/2
>>> one_third = Rational(1, 3)
>>> one_half + one_third
5/6
>>> one_half * one_third
1/6
>>> one_third + one_third
6/9
```

2.2 C++ オブジェクト指向によるデータの抽象化

有理数の作成プログラムは，C++言語オブジェクト指向を用いて以下のように記述することが出来る.

g++ -o rat rat.cpp としてコンパイルし，./rat として実行する.

```
#include <iostream>

class Rational {
public:
  Rational(int n, int d)
  {
    this->numer = n;
    this->denom = d;
  }

  Rational add_rat(Rational y)
  {
    return Rational(this->numer*y.denom + y.numer*this->denom, this->denom*y.denom);
  }

  Rational sub_rat(Rational y)
  {
    return Rational(this->numer*y.denom - y.numer*this->denom, this->denom*y.denom);
  }

  Rational mul_rat(Rational y)
  {
    return Rational(this->numer*y.numer, this->denom*y.denom);
  }

  Rational div_rat(Rational y)
  {
    return Rational(this->numer*y.denom, this->denom*y.numer);
  }

  bool equal_rat(Rational y)
  {
    return this->numer*y.denom == y.numer * this->denom;
  }

  void print_rat()
  {
    std::cout << this->numer << "/" << this->denom << std::endl;
  }
private:
  int numer;
  int denom;
};

int  main()
{
  Rational one_half = Rational(1, 2);
  one_half.print_rat();
  // std::cerr << one_half.numer << "/" << one_half.denom << std::endl;
  Rational one_third = Rational(1, 3);
  one_half.add_rat(one_third).print_rat();
  one_half.mul_rat(one_third).print_rat();
  one_third.add_rat(one_third).print_rat();
}
```

C++のコンパイルは `c++` コマンドを使ってもよい．`which c++`とすると，どの実行ファイルが実

行されているかわかる．調べていくと，c++もg++も同じバイナリとなることがわかる．また，gccでもgcc -o rat rat.cpp -lstdc++とC++のライブラリを指定するとコンパイルできる．さらに最近はclang / clang++と呼ばれるコンパイラも存在する．これはLLVM(Low Level Virtual Machine)と呼ばれる仮想マシン用の中間コード（ビットコード）を経由し，機械語を生成する．この時言語やCPUに独立な最適化が行われる．clang++ -S -emit-llvm foo.cppとするとintermediate representation (IR) ファイルを出力でき， clang -c -emit-llvm foo.cpp -o - | hexdump -Cとするとビットコードが出力できる．

2.2.1 アクセス制限

C++言語では，Pythonよりも強力なアクセス制限の機構を有している，

まず，public:キーワード以下で定義された関数，変数は制約がない．すなわちPythonでのメンバ関数・変数と同じ扱いになる．しかしprivate:キーワード以下で定義されたメンバ関数，変数をクラスの外部からのアクセス禁止を宣言している．この例ではnumer，denom変数のクラス外からのアクセスが禁止されている．

これ以外に，子クラスからは利用することが出来るようになるprotected:キーワード，privateメンバ変数を関数を，外部のクラス・関数からも利用することが出来るようになるfriend:キーワードを指定することが出来る．

C++ではコンパイル時にこれを検出する．例えば，main文でコメントアウトされている

std::cerr << one_half.numer << "/" << one_half.denom << std::endl; をコメントインしてコンパイルすると，以下のようにエラーとなる．

```
$ g++ -o rat rat.cpp
rat.cpp: In function ‘ int main() ’:
rat.cpp:49:25: error: ‘ int Rational::numer ’ is private within this context
   std::cerr << one_half.numer << "/" << one_half.denom << std::endl;
                         ^~~~~
rat.cpp:41:7: note: declared private here
   int numer;
       ^~~~~
（省略）
```

これで，プライベートなメンバとして定義された変数は，そのクラス内だけで利用されている事が保障される．従って，例えばnumer，denomという変数名をn，dと変更した場合においても，その影響は定義されているクラス内に留まり，このクラスを用いて構築されているソフトウェアへの影響が無いことが保障されている．言い換えれば，変数がクラス内に隠蔽され，安全性が高まっていると言える．

2.2.2 演算子オーバーロード

C++言語における演算子オーバーロードは以下のように書くことが出来る．ここでは，出力演算子オーバーロード (operator<<) をfriend:キーワードを用いてプロトタイプ宣言している．これにより，出力演算子オーバーロードの定義内で，Rationalクラス内のメンバ関数を特別にアクセスできるよう指示している．

```
#include <iostream>

class Rational {
public:
  Rational(int n, int d)
  {
    this->numer = n;
    this->denom = d;
  }
  Rational operator+(const Rational &y)
  {
    return Rational(this->numer*y.denom + y.numer*this->denom, this->denom*y.denom);
  }

  Rational operator-(const Rational &y)
  {
    return Rational(this->numer*y.denom - y.numer*this->denom, this->denom*y.denom);
  }

  Rational operator*(const Rational &y)
  {
    return Rational(this->numer*y.numer, this->denom*y.denom);
  }

  Rational operator/(const Rational &y)
  {
    return Rational(this->numer*y.denom, this->denom*y.numer);
  }

  bool operator=(const Rational &y)
  {
    return this->numer*y.denom == y.numer * this->denom;
  }
  friend std::ostream& operator<<(std::ostream& os, const Rational& y);

private:
  int numer;
  int denom;
};

std::ostream &operator<<(std::ostream &os, const Rational &y)
{
  os << y.numer << "/" << y.denom;
  return os;
}

int  main()
{
  Rational one_half = Rational(1, 2);
  std::cerr << one_half << std::endl;
  Rational one_third = Rational(1, 3);
  std::cerr << one_half + one_third << std::endl;
  std::cerr << one_half * one_third << std::endl;
  std::cerr << one_third + one_third << std::endl;
}
```

2.2.3 テンプレート関数

線型結合 $ax + by$ のプログラムはオーバロード機能を用いて以下のように書ける．

```
int linear_combination (int a, int b, int x, int y) {
  return (a*x) + (b*y);
}

Rational linear_combination (Rational a, Rational b, Rational x, Rational y) {
  return (a*x) + (b*y);
}
```

ここでは，linear_combination という 1 つの関数名ながら，int と Rational という異なる引数の関数を定義することが出来ている．また，プログラム実行時には与えられた引数に応じて適した関数が呼ばれている[6]．

```
int main() {
  std::cerr << linear_combination(1, 2, 3, 4) << std::endl;
  std::cerr << linear_combination(one_half, one_half, one_third, one_third) << std::endl;
}
```

一方で，linear_combination の 2 つの関数自体は同じ記述になっている．ここでは任意の型に適用可能な関数やクラスを定義するテンプレート関数を用いることが出来る．

このような様々な型のデータに適用できる操作を定義する関数を汎用関数と呼ぶ．template 直後の型は template キーワードの直後の<と>の間に書かれる．汎用関数は処理するデータの型を仮引数として受け取る．

template キーワードの構文は

```
template <class t>  戻り値の型  関数名 (T 引数, T 引数, ....) {
}
```

である．テンプレート関数は実際に使われたときにはじめて具体的な型名が入って解釈され，異なる型に対してその型名が入ったコードの実体が生成されている．型名が入ったコードは，正当なコードでなければいけない．

linear_combination をテンプレート関数で定義すると以下のようになる．ここでは任意の型 T の変数 a，b，x，y を引数とし，線形結合を返す関数を定義している．この例ではオペレータ+，* が定義されていない型に対しては使用できない．

```
template <class T> T linear_combination (T a, T b, T x, T y) {
  return (a*x) + (b*y);
}
```

[6] nm -A をつかってオブジェクト内で定義されているシンボル（名前）を見ることが出来る．これで，オーバーロードの仕組みを垣間見ることが出来る

3 C++言語の拡張

C++言語は 1998 年に標準化されているが，C++の標準化委員会の委員を中心に先駆的な開発者のコミュニティによって次世代の C++の標準化に向けたアイデアのテストと基準となる実装を Boost C++ライブラリ[7]として提供している．

一方，C++自体は 2003 年に C++03, 2011 年に C++11, 2014 年に C++14, 2017 年に C++17, 2020 年に C++20 と呼ばれる標準仕様が決定されている．現在は C++23 の仕様策定が進んでいる[8]．

GCC コンパイラはこれらの新しい仕様をサポートしている[9]．

C++11 の機能は GCC 4.8.1 以降からデフォルトで有効になっている．それ以前では，`-std=c++11` で利用できる．

C++14 の機能は GCC 6.1 以降からデフォルトで有効になっている．それ以前では，`-std=c++14` で利用できる．

C++17 の機能は GCC 5 以降から`-std=c++17` オプションで利用出来るようになっているが，GCC 10 までは試験的機能であり注意が必要である．また，GCC 11 以降でデフォルトで有効になっている．

C++20 の機能は GCC 8 以降から`-std=c++2a` オプションで試験的に利用出来るようになっている．GCC 10 以降では`-std=c++20` としてオプションしていする．

Ubuntu 20.04 にインストールされている GCC のバージョンは `gcc --version` として知ることができ，現在は 9.4.0 である．[10]

3.1 型推論 (auto)

C++11 から変数宣言時に具体的な型名の代わりに `auto` キーワードを指定することで，変数の型を推論出来るようになった．また，C++14 からは，ユーザ定義関数の戻り値や引数の型を指定出来るようになっている,[11]．

例えば高階関数を用いた `sum` の実装は以下のように書くことができる実際に，全ての int を auto に書き換えるとプログラムは動くだろうか？確認してみよう．

[7] `https://ja.wikipedia.org/wiki/Boost_C%2B%2B%E3%83%A9%E3%82%A4%E3%83%96%E3%83%A9%E3%83%AA`
[8] `https://ja.wikipedia.org/wiki/C%2B%2B`
[9] `https://gcc.gnu.org/projects/cxx-status.html`
[10] Ubuntu 18.04 にインストールされている GCC のバージョンは 7.5.0 である．
[11] `https://cpprefjp.github.io/lang/cpp11/auto.html`

```
#include <iostream>

auto sum(auto term, auto a, auto next, auto b) {
  if ( a > b ) {
    return 0;
  } else {
    return term(a) + sum(term, next(a), next, b);
  }
}

int inc(int n) { return n + 1; }
int identity(int x) { return x; }
int sum_integers(int a, int b) {
  return sum(identity, a, inc, b);
}

int cube(int x) { return x * x * x; }
int sum_cubes(int a, int b) {
  return sum(cube, a, inc, b);
}

int main() {
  std::cout << "sum_integers(1, 10): " << sum_integers(1, 10) << std::endl;
  std::cout << "sum_cubes(1, 10): " << sum_cubes(1, 10) << std::endl;
}
```

3.2 無名関数 (generic lambdas)

また，C++11 から無名関数が利用できるようになっている．C++14 では無名関数の引数に auto キーワードを用いる事が出来る[12]．例えば

```
auto plus = [](auto a, auto b) { return a + b; };
```

という記述はコンパイル時に，以下のようなテンプレート関数よる定義に変換され，コンパイルされる．

```
template <class T1, class T2>
auto operator()(T1 a, T2 b) const
{
  return a + b;
}
```

無名関数を用いた例を以下に示そう．

[12] https://cpprefjp.github.io/lang/cpp14/generic_lambdas.html

```
#include <iostream>

auto sum(auto term, auto a, auto next, auto b) {
  if ( a > b ) {
    return 0.0; // THIS IS VERY IMPORTANT
  } else {
    return term(a) + sum(term, next(a), next, b);
  }
}

//double pi_term(double x) {return 1.0 / (x * (x + 2.0));}
//double pi_next(double x) {return x + 4;}
double pi_sum(int a, int b) {
  return sum([](auto x){return 1.0 / (x * (x + 2.0));},
             (double)a,
             [](auto x){return x + 4.0;},
             (double)b);
}

int main() {
  std::cout << "8 * pi_sum(1, 10000): " << 8 * pi_sum(1, 100000) << std::endl;
}
```

3.3 クロージャ

C++では，無名関数（ラムダ式）の中では，外部の環境の変数の束縛を明示的に指定する必要がある．C++ではキャプチャと呼ばれている．キャプチャには，参照キャプチャ，コピーキャプチャがある．例えば以下の例では，`[x]()` とすることによって，変数 x をラムダ式の中でも参照出来るようになる，さらに，`[&y]()` とすることで，変数 y をラムダ式の中で変更が可能になる．

一方で，コピーキャプチャでも `mutable` キーワードを付けることで，変数の変更が可能になる．ただし，値はコピーされているので，元の環境の変数は変更されることはない．

```
#include <iostream>

int main() {
  int x = 1;
  int y = 10;
  int z = 100;

  [x](){ std::cout << "x : " << x << std::endl; }();
  std::cout << "x = " << x << ", y = " << y << ", z = " << z << std::endl;

  [&y](){ y += 1; std::cout << "y : " << y << std::endl; }();
  std::cout << "x = " << x << ", y = " << y << ", z = " << z << std::endl;

  [z]() mutable{ z += 1; std::cout << "z : " << z << std::endl; }();
  std::cout << "x = " << x << ", y = " << y << ", z = " << z << std::endl;
}
```

例えば Python 言語で紹介したクロージャのプログラムは以下のように書ける．

```
#include <iostream>

auto f(auto a)
{
  auto b = 100;
  return [a, b](auto c) {return a + b + c; };
}

int main() {
  int a = -10;
  int b = -100;
  int c = -1000;

  auto h = f(10);
  std::cerr << h(1000) << std::endl;
}
```

なお，このコードはコンパイルすると以下のようなワーニングが表示される．

```
warning: use of ‘ auto ’ in parameter declaration only available with ‘ -fconcepts ’
```

この場合は，表示に従い `g++ -fconcepts -o lambda ./lambda.cpp` としてオプションを追加してコンパイルするか，以下のように無名関数を利用するとよい．

```
auto f = [](auto a)
{
  auto b = 100;
  return [a, b](auto c) {return a + b + c; };
};
```

また，同じコードを `clang++ -o lambda lambda.cpp` として clang でコンパイルするとエラーとなる．clang では `clang++ -std=c++20 -o lambda lambda.cpp` として，C++20 を有効にするとエラーを解消できる．

GCC では，GCC 9 では未サポートだが GCC 10 ではサポートされている．`g++-9 -std=c++2a ...`, `g++-10 -std=c++20 ...` として確認できる．

宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．

画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．

ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：オーバライド，オーバーロード，アクセス制限

1. make_withdraw 関数と同様の挙動を示すプログラムを，オブジェクト指向のクラスを作成し実装せよ．プログラム言語は問わない．また，余裕のあるものは複数の言語で実装してみよう．

2. make_withdraw 関数を C++言語でクロージャを用いて記述せよ．クロージャの中で変数の値を更新するため，クロージャーをコピーキャプチャとして mutable にするとよい．

   また，Python のコードでは条件に応じて数値や文字列を返しているが C++ではそれはできないため，return 'Insufficient funds' の部分は，例えばメッセージを表示した上で 0 を返す等の処置が必要である．

   以下ががヒントになる．

```
#include <iostream>

auto make_withdraw(auto init_balance)
{
  auto balance = init_balance;
  return [balance](auto amount) mutable
  {
    ....
  };
}

int main() {
  auto kei = make_withdraw(100);
  auto bill = make_withdraw(1000);

  std::cout << "kei  :" << kei(25) << std::endl;
  std::cout << "bill :" << bill(25) << std::endl;

  std::cout << "kei  :" << kei(25) << std::endl;
  std::cout << "bill :" << bill(25) << std::endl;

  std::cout << "kei  :" << kei(60) << std::endl;
  std::cout << "bill :" << bill(60) << std::endl;
}
```

3. 以下の平方を管理するプログラムをテンプレートを用い Square<int> a(10) や Square<double> b(10.1) として int 型でも double 型でも Square クラスが利用出来るようにせよ．

```
$ cat test.cpp
#include <iostream>

class Square {
public:
  int data;
  Square (int _data) { data = _data * _data; }
  ~Square () {};
};

int main () {
  Square a(10);

  std::cerr << a.data << std::endl;
}
$ g++ -o test test.cpp
$ ./test
100
```

以下のプログラムをC++で書いてください。

```
def make_withdraw():
  balance = 100
  def withdraw(amount):
    nonlocal balance
    if balance >= amount:
      balance = balance - amount
      return balance
    else:
      return 'Insufficient funds'
  return withdraw

withdraw = make_withdraw()

print (withdraw(25))
print (withdraw(25))
print (withdraw(60))
print (withdraw(15))
```

あなたが示したPythonプログラムは閉包(closure)を使っています。C++では、ラムダ関数とstd::functionを使って類似の動作を模倣することができます。以下にC++のコードを示します。

cpp Copy code

```cpp
#include <iostream>
#include <functional>

std::function<int(int)> make_withdraw() {
    int balance = 100;
    return [balance](int amount) mutable -> int {
        if (balance >= amount) {
            balance -= amount;
            return balance;
        } else {
            std::cout << "Insufficient funds" << std:
            return balance;
        }
    };
}

int main() {
    auto withdraw = make_withdraw();

    std::cout << withdraw(25) << std::endl;
    std::cout << withdraw(25) << std::endl;
    std::cout << withdraw(60) << std::endl;
    std::cout << withdraw(15) << std::endl;

    return 0;
}
```

このC++コードでは `make_withdraw` 関数は、バランスを管理するラムダ関数を生成して返します。ラムダ関数は `balance` をキャプチャしており、呼び出されるたびにそのバランスを減らします。そのため、Pythonの閉包のように振る舞います。ただし、`Insufficient funds`のメッセージが表示されるとき、C++のバージョンではその後もバランスが表示されます。これはC++がPythonのように異なる型を返すことができないためです。

以下のプログラムをtemplate で書き直してください。

```
#include <iostream>

class Square {
public:
 int data;
 Square (int _data) { data = _data * _data; }
 ~Square () {};
};

int main () {
 Square a(10);

 std::cerr << a.data << std::endl;
}
```

以下に、C++テンプレートを使用して書き直したコードを示します。このコードでは、Square クラスが任意の型の値を受け取り、その値の二乗を計算するようになっています。

cpp Copy code

```cpp
#include <iostream>

template <typename T>
class Square {
public:
  T data;
  Square (T _data) { data = _data * _data; }
  ~Square () {};
};

int main () {
  Square<int> a(10);

  std::cerr << a.data << std::endl;
}
```

このプログラムでは、Squareクラスがテンプレートクラスとなり、コンストラクタの引数とデータメンバーが任意の型になります。テンプレートパラメータ`T`は型パラメータで、コンストラクタの引数とデータメンバーの型を表します。

`main`関数の中で、`Square<int>`のようにクラス名の後に`<型名>`を記述することで、その型のインスタンスを生成できます。ここでは`int`型のインスタンスを生成していますが、他の型のインスタンスを生成することも可能です。

A オブジェクト指向言語

A.1 Python 言語

代表的なオブジェクト指向言語である Python を用いて，オブジェクト指向の概念を学ぼう

A.1.1 クラスの定義

変数や関数をカプセルかするための仕組みがクラスである．python では以下のように定義する．
クラスでは変数だけでなく，関数もクラスに含めることができる．クラスに含まれる変数，関数をそれぞれメンバ変数，メンバ関数という．

```
1: class Student:
2:     name = None
3:     grade = 0
4:
5:     def __init__(self, name):
6:         self.name = name
7:
8:     def set_grade(self,grade):
9:         self.grade = grade
10:
11:    def get_grade(self):
12:       return self.grade
13:
14:    def get_name(self):
15:        return self.name
```

1 行目は class キーワードを用いてクラスの定義を宣言している．2,3 行目のでメンバ変数を定義している．5 行目の__init__はコンストラクタで，第一引数の self はインスタンスを指し，第二引数からユーザ定義の引数パラメータになっている．ここでは，コンストラクタで name という引数をとり，これをメンバ変数である self.name に代入している．
8 行目，11 行目，14 行目ではそれぞれメンバ関数を定義しておりそれぞれ引数を用いてメンバ変数の値を更新していたり，あるいはメンバ変数を返している．
このプログラムをこのクラスを使ったサンプルプログラムは以下のようなものになる．

```
s1 = Student("yamada")
s2 = Student("suzuki")

s1.set_grade(1)
s2.set_grade(2)

print("student name = %s (%d)"%(s1.get_name(), s1.get_grade()))
print("student name = %s (%d)"%(s2.get_name(), s2.get_grade()))
```

このプロゴラムコードをクラス定義と同じファイルに記述し

```
$ python # または ipython
>>> exec(open('ファイル名').read())
```

として実行できる．例えば

```
>>> exec(open('main.py').read())
student name = yamada (1)
student name = suzuki (2)
```

となる．クラス定義とサンプルプログラムを別のファイルにする場合は，サンプルプログラムの先頭に `from student import Student` を追加するとよい．この場合クラス定義は同じディレクトリのの `student.py` というファイル名のファイルで記述されていることを想定している．

ただしこの場合 `student.py` に書かれている内容を変更した際には

```
import student
reload(student)
```

などとする必要が有る．

A.1.2 継承

Student クラスと同様に Professor クラスを定義してみると次のようになる．

```
class Professor:
    name = None
    room = 0

    def __init__(self, name):
        self.name = name

    def set_room(self,room):
        self.room = room

    def get_room(self):
        return self.room

    def get_name(self):
        return self.name
```

Studnet クラスと比較すると同じメンバ変数 (name)，メンバ関数 (`get_name()`) を持っていることに気がつく．このように異なるクラスに渡って共通するメンバ変数，メンバ関数を持つ場合，共通する部分を取り出した LabMember というクラスを作り Student, Professor というクラスはこの LabMember というクラスのメンバ変数，メンバ関数を引き継ぐように出来るとよい．これを出来るようにするのが継承である．LabMember クラスを Student, Professor クラスの親クラス（スーパークラス）と呼び，逆に Student, Professor クラスは LabMember クラスの子クラス（サブクラス）となり，LabMember クラスを継承しているという．LabMember を継承した Student クラスの実装は次のようになる．

```
class LabMember:
    name = None

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name
```

LabMember クラス

```
class Student(LabMember):
    grade = 0

    def __init__(self, name):
        LabMember.__init__(self, name)

    def set_grade(self,grade):
        self.grade = grade

    def get_grade(self):
        return self.grade
```

LabMember を継承した Student クラスの実装

```
class Professor(LabMember):
    room = 0
    def __init__(self, *args, **kwargs):
        LabMember.__init__(self, *args, **kwargs)
    def set_room(self,room):
        self.room = room
    def get_room(self):
        return self.room
```

LabMember を継承した Professor クラスの実装

これらのクラスを利用したプログラムとして，以下の student2.py を作ってみる.（上の LabMember, Student, Professor クラスの定義を同じファイルに記述するか別ファイルにした場合はモジュールとしてロードすること)

```
p1 = Professor("yamada")
s1 = Student("suzuki")

p1.set_room(123)
s1.set_grade(4)

print("professor name = %s (%d)"%(p1.get_name(), p1.get_room()))
print("student name = %s (%d)"%(s1.get_name(), s1.get_grade()))
```

これを実行すると，

```
>>> execfile('student2.py')
professor name = yamada (123)
student name = suzuki (4)
```

となるはずである.

Student クラス，Professor クラスの定義時に，

```
class Professor(LabMember)
class Student(LabMember)
```

と書くことで，これらのクラスが LabMember クラスを継承する子クラスであることを示している．これにより Student クラス，Professor クラスでは LabMember クラスのメンバ変数である name やメンバ関数である get_name を定義する必要がない．また，子クラスのコンストラクタで親クラスのメンバ変数を初期化する必要があるが，これは，

```
def __init__(self, name):
    LabMember.__init__(self, name)
```

の部分に当たる．Student クラスのコンストラクタが呼ばれたとき，その引数_name を使って親クラスである LabMember のコンストラクタを呼び出している．

A.1.3 オーバーライド

オブジェクト指向の多態性を実現する機能の一つにオーバーライドがある．

オーバーライドとは，子クラスにおいて親クラスの関数を定義しなおすことである．たとえば LabMember クラスに set_name() というメンバ関数を追加し，その実装が

```
class LabMember:

  ...

   def set_name(self, name):
       self.name = "Mr. " + name
```

となっているとする．

あるインスタンスに対して

```
s1.set_name("Tanaka");
```

というメンバ関数を呼び出すと，そのインスタンスのメンバ変数 name は” Mr. Tanaka” という文字列になる．これに対し Professor クラスに対しても同じ set_name() というメンバ関数を定義するのがオーバーライドである．

すなわち，以下のような定義を Professor クラスに加え，Professor クラスのインスタンスに対して set_name() メンバ関数を呼び出すと”Prof. Tanaka” という文字列が name メンバ変数に代入される．

```
class Professor(LabMember):

   ...

   def set_name(self, name):
       self.name = "Prof. " + name
```

このように同じ名前・引数のメンバ関数を異なるクラスで定義することで，クラスに応じて適切な処理を行うことである．

オーバーライドを使ったプログラムをまとめると以下のようになる．

```
class LabMember:
    name = None

    def __init__(self, name):
        self.name = name

    def set_name(self, name):
        self.name = "Mr. " + name

    def get_name(self):
        return self.name

class Student(LabMember):
    grade = 0

    def __init__(self, name):
        LabMember.__init__(self, name)

    def set_grade(self,grade):
        self.grade = grade

    def get_grade(self):
        return self.grade
```

```
class Professor(LabMember):
    room = 0

    def __init__(self, *args, **kwargs):
        LabMember.__init__(self, *args, **kwargs)

    def set_name(self, name):
        self.name = "Prof. " + name

    def set_room(self,room):
        self.room = room

    def get_room(self):
        return self.room

p1 = Professor("")
s1 = Student("")
p1.set_name("yamada")
s1.set_name("suzuki")
p1.set_room(123)
s1.set_grade(4)

print("professor name = %s (%d)"
          %(p1.get_name(), p1.get_room()))
print("student name = %s (%d)"
          %(s1.get_name(), s1.get_grade()))
```

これまでのプログラムと同様に実行し以下のような結果を得られるはずである.

```
$ python student3.py
professor name = Prof. yamada (123)
student name = Mr. suzuki (4)
```

最後に，Student と Professor を構成員とする Lab クラスを考えてみる.

Lab の構成員は Professor クラスと Student クラスの 2 つのインスタンスの可能性があるが，どちらもまとめてリストを用いて以下のように管理する,

```
class Lab:
    members = None

    def __init__(self):
        self.members = []

    def add_member(self, new_member):
        self.members.append(new_member)

    def print_member(self):
        for member in self.members:
            member.print_info()
```

Lab クラスの定義

これを使ったプログラムの例を以下に示す.

```
class LabMember:
    name = None

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

class Student(LabMember):
    grade = 0

    def __init__(self, name):
        LabMember.__init__(self, name)

    def set_grade(self,grade):
        self.grade = grade

    def get_grade(self):
        return self.grade

    def print_info(self):
        print("student name = %s (%d)"%(self.name, self.grade))

class Professor(LabMember):
    room = 0

    def __init__(self, *args, **kwargs):
        LabMember.__init__(self, *args, **kwargs)

    def set_room(self,room):
        self.room = room

    def get_room(self):
        return self.room

    def print_info(self):
        print("professor name = %s (%d)"%(self.name, self.room))
```

```
jsk = Lab()
p1 = Professor("yamada")
s1 = Student("suzuki")
p1.set_room(123)
s1.set_grade(4)
jsk.add_member(p1)
jsk.add_member(s1)
jsk.print_member()
```

A.1.4 演算子オーバーロード

Python では演算子のオーバーロード[13]が可能である.
そのために「特殊メソッド (special method name)」名が定義されている. 詳しくは

[13]オーバライドとはスーパークラスで定義されたメソッドをサブクラスで再定義することであり, オーバロード (多重定義) とは同一のメソッド名でも引数の型, 数, 並び順が異なるメソッドを複数定義すること

http://docs.python.jp/2/reference/datamodel.html を参照すること．
例えば，

```
class Professor(LabMember):

   ...

    def __str__(self):
        return "professor name = %s (%d)"%(self.name, self.room)
```

と定義すると，print p1 とすると以下の表な表示を得ることができる．

```
>>> print p1
professor name = yamada (123)
```

一方で，演算子オーバーロドしていない Student クラスのインスタンスは以下のようなままである．

```
>>> print s1
<__main__.Student instance at 0x7f430a4053b0>
```

A.2 C++言語

代表的なオブジェクト指向言語である C++を用いて，オブジェクト指向の概念を学ぼう

A.2.1 クラスの定義

変数や関数をカプセル化するための仕組みがクラスである．例えば C 言語では構造体を使ってカプセル化を実現することができる．C++言語ではクラスを用いてカプセル化を実現する．

```
struct Student {
  char name[32];
  int grade;
};
```

C 言語における構造体の定義

```
class Student {
  char name[32];
  int grade;
};
```

C++言語におけるクラスの定義

C++言語におけるクラスでは変数だけでなく，関数もクラスに含めることができる．C++言語ではクラスに含まれる変数，関数をそれぞれメンバ変数，メンバ関数という．

```
 1: class Student {
 2:   private:
 3:    char name[32];
 4:    int grade;
 5:
 6:   public:
 7:    Student(const char* _name);
 8:    ~Student();
 9:
10:    void SetGrade(int _grade);
11:    int GetGrade();
12:    char* GetName();
13: };
```

C++言語におけるクラスの定義．メンバ変数，メンバ関数の定義が含まれている (student.h)．

2 行目の private:により，この行以降のメンバ関数，メンバ変数の外部からのアクセスを禁止を宣言しており，6 行目の public:により，この行以降のメンバ関数，メンバ変数の外部からのアクセスを許可している．

7,8 行目の Student(),〜Student() はそれぞれコンストラクタ，デストラクタと呼び，インスタンスの生成，消去を行っている．コンストラクタ，デストラクタの名前はクラス名と同じで，デストラクタは名前の前に〜をつけたものとなる．それぞれ値を返すことはできず，デストラクタは引数をとることができない．

10 行目の SetGrade 関数は外部からアクセスできない grade 変数の値を設定するときに使われる．メンバ変数を隠蔽し，メンバ関数によるアクセスのみを許すことによって，プログラムミスの原因を特定しやすくする，変数の読み取りだけを許す，ということが可能になる．また，ある変数の変化に伴い別の変数を書き換え整合性を取る必要がある場合に，書き換え漏れを防ぐことができるようになる．

A.2.2 メンバ関数の定義

メンバ関数の定義には，クラス定義の中に埋め込む方法と，クラス定義の外に記述する方法の 2 つがある．

前者はメンバ関数の実体をクラス定義の中にまとめて記述する方法で，これら全体を一つのヘッダファイルに入れることが多い．

一方，後者はクラス定義の外で Student::を追加することで Student クラスのメンバ関数であることを明示する方法である．この場合は，クラス定義をヘッダファイルに，メンバ関数の定義を対応する名前のソースファイルに記述することが多い．C++言語のソースファイルの拡張子は.cpp が一般的であるが，.C であったり．cxx が使われることもある．

```
class Student {
  private:
   char name[32];
   int grade;

  public:
    Student(const char* _name) {
      strcpy(name, _name);
      grade = 0;
    }
    ~Student() {
    }
    void SetGrade(int _grade) {
      grade = _grade;
    }
    int GetGrade() {
      return grade;
    }
    char* GetName() {
      return name;
    }
};
```

クラス定義の中にメンバ関数を定義する方法 (student_h.h)

```
#include "student.h"
#include <string.h>

Student::Student(const char* _name) {
    strcpy(name, _name);
    grade = 0;
}
Student::~Student() {
}
void Student::SetGrade(int _grade) {
    grade = _grade;
}
int Student::GetGrade() {
    return grade;
}
char* Student::GetName() {
    return name;
}
```

クラス定義の外にメンバ関数を定義する方法 (student.cpp)

このクラスを使ったサンプルプログラムは以下のようなものになる.

```
#include <stdio.h>
#include "student.h"

int main (int argc, char **argv) {
  Student s1("yamada");
  Student s2("suzuki");

  s1.SetGrade(1);
  s2.SetGrade(2);

  printf("student name = %s (%d)\n", s1.GetName(), s1.GetGrade());
  printf("student name = %s (%d)\n", s2.GetName(), s2.GetGrade());

  return 0;
}
```

C++言語によるサンプルプログラム (test-student.cpp)

このプログラムのコンパイルは C++言語用のコンパイラ g++を用いて

```
$ g++ -o test-student test-student.cpp student.cpp
```

のようにしてコンパイルすることができる. このようにして出来たプログラム (機械語のファイル)test-student を実行すると,

```
$ ./test-student
student name = yamada (1)
```

```
student name = suzuki (2)
```

という結果が表示されるはずである．また，test-student.cpp のなかで student.h の代わりに，クラス定義の中にメンバ関数を定義した student_h.h をインクルードし，

```
$ g++ -o test-stuent test-student.cpp
```

のようにコンパイルしてもよい．

A.2.3 継承

Student クラスと同様に Professor クラスを定義してみると次のようになる．

```
class Professor {
  private:
   char name[32];
   int room;

  public:
   Professor(const char* _name);
   ~Professor();
   void SetRoom(int _room);
   int GetRoom();
   char* GetName();
}
```

Studnet クラスと比較すると同じメンバ変数 (name)，メンバ関数 (GetName()) を持っていることに気がつく．このように異なるクラスに渡って共通するメンバ変数，メンバ関数を持つ場合，共通する部分を取り出した LabMember というクラスを作り Student, Professor というクラスはこの LabMember というクラスのメンバ変数，メンバ関数を引き継ぐように出来るとよい．これを出来るようにするのが継承である．LabMember クラスを Student, Professor クラスの親クラス（スーパークラス）と呼び，逆に Student, Professor クラスは LabMember クラスの子クラス（サブクラス）となり，LabMember クラスを継承しているという．LabMember を継承した Student クラスの実装は次のようになる．

```
class LabMember {
  private:
   char name[32];

  public:
    LabMember(const char* _name) {
      strcpy(name, _name);
    }
    ~LabMember() {
    }
    char* GetName() {
      return name;
    }
};
```

LabMember クラス (labmember.h)

```
class Student : public LabMember {
  private:
   int grade;

  public:
    Student(const char* _name)
      : LabMember(_name) {
      grade = 0;
    }
    ~Student() {
    }

    void SetGrade(int _grade) {
      grade = _grade;
    }

    int GetGrade() {
      return grade;
    }
};
```

LabMember を継承した Student クラスの実装 (student2.h)

```
class Professor : public LabMember {
  private:
   int room;

  public:
    Professor(const char* _name)
      : LabMember(_name) {
      room = 0;
    }
    ~Professor() {
    }

    void SetRoom(int _room) {
      room = _room;
    }

    int GetRoom() {
      return room;
    }
};
```

LabMember を継承した Professor クラスの実装 (professor2.h)

これらのクラスを利用したプログラムとして，以下の test-student2.cpp を作ってみる．

```
#include <stdio.h>
#include <string.h>
#include "labmember.h"
#include "student2.h"
#include "professor2.h"

int main (int argc, char **argv) {
  Professor p1("yamada");
  Student s1("suzuki");

  p1.SetRoom(123);
  s1.SetGrade(4);

  printf("professor name = %s (%d)\n", p1.GetName(), p1.GetRoom());
  printf("student name   = %s (%d)\n", s1.GetName(), s1.GetGrade());

  return 0;
}
```

このプログラムのコンパイルは

```
$ g++ -o test-student2 test-student2.cpp
```

のようにして行い，その結果は

```
professor name = yamada (123)
student name   = suzuki (4)
```

となるはずである．

student2.h, professor2.h において，Student クラス，Professor クラスの定義時に，

```
class Professor : public LabMember
class Student : public LabMember
```

と書くことで，これらのクラスが LabMember クラスを継承する子クラスであることを示している．これにより Student クラス，Professor クラスでは LabMember クラスのメンバ変数である name やメンバ関数である GetName を定義する必要がない．また，子クラスのコンストラクタで親クラスのメンバ変数を初期化する必要があるが，これは，

```
Student (const char* _name) : LabMember(_name) {
  room = 0;
}
```

の部分に当たる．Student クラスのコンストラクタが呼ばれたとき，その引数_name を使って親クラスである LabMember のコンストラクタを呼び出している．

また，LabMember クラスの private メンバである name 変数には，どの子クラスからもアクセスすることは出来ない．すなわち，student2.h において

```
Student(char* _name) : LabMember(_name) {
  grade = 0;
  strcpy(name, "who");
}
```

のように記述を変更しコンパイルしようとすると．

```
$ g++ -o test-student2 test-student2.cpp
In file included from test-student2.cpp:4:
student2.h: In constructor `Student::Student(char*)':
labmember.h:3: error: `char LabMember::name[32]' is private
student2.h:8: error: within this context
```

のようにエラーとなるはずである．

外部からはアクセスできないが子クラスからはアクセスできるメンバ変数を宣言するためには，private:の代わりに protected:を利用する．従って

```
class LabMember {
  protected:
   char name[32];

  public:
    LabMember(char* _name) {
      strcpy(name, _name);
    }
    ~LabMember() {
    }

    char* GetName() {
      return name;
    }
};
```

のように定義すればコンパイルのエラーを回避できる．

A.2.4 オーバーライドとオーバーロード

C++においてオブジェクト指向の多態性を実現しているのがオーバーライドとオーバーロードの機能である．

オーバーライドとは，子クラスにおいて親クラスの関数を定義しなおすことである．たとえばLabMember クラスに SetName() というメンバ関数を追加し，その実装が

```
class LabMember {
    ...
    char* SetName(char* _name) {
      sprintf(name, "Mr. %s", _name);
    }
    ...
};
```

となっているとする．

あるインスタンスに対して

```
  s1.SetName("Tanaka");
```

というメンバ関数を呼び出すと，そのインスタンスのメンバ変数 name は” Mr. Tanaka” という文字列になる．これに対し Professor クラスに対しても同じ SetName() というメンバ関数を定義するのがオーバーライドである．

すなわち，以下のような定義を Professor クラスに加え，Professor クラスのインスタンスに対して SetName() メンバ関数を呼び出すと”Prof. Tanaka” という文字列が name メンバ変数に代入される．

```
class Professor : public LabMember{
    ...
    char* SetName(char* _name) {
      sprintf(name, "Prof. %s", _name);
    }
    ...
};
```

このように同じ名前・引数のメンバ関数を異なるクラスで定義することで，クラスに応じて適切な処理を行うことである．

一方，オーバーロードとは関数（メンバ関数である必要はない）に対して同じ関数名で，引数のみ異なる関数を定義することを言う．つまり，

```
void print_labmember(Student s) {
    printf("student name = %s, grade = %d\n", s.GetName(), s.GetGrade());
}

void print_labmember(Professor p) {
    printf("professor name = %s, room = %d\n", p.GetName(), p.GetRoom());
}
```

のように，同じ名前で異なる引数を持つ関数を定義することが可能であり，同じ名前の関数を呼び出していても，引数の型に応じて適切な関数が選ばれ，プログラムの可読性や保守の容易さの向上につながる．

オーバーライド，オーバーロードを使ったプログラムをまとめると以下のようになる．

```
/* labmember3.h */

class LabMember {
  protected:
   char name[32];

  public:
    LabMember(char* _name) {
      strcpy(name, _name);
    }
    ~LabMember() {
    }

    char* GetName() {
      return name;
    }
};
```

```
/* student3.h */

class Student : public LabMember {
 private:
  int grade;

 public:
   Student(char* _name)
  : LabMember(_name) {
     grade = 0;
   }
   ~Student() {
   }

   char* SetName(char* _name) {
     sprintf(name, "Mr. %s", _name);
   }
   void SetGrade(int _grade) {
     grade = _grade;
   }
   int GetGrade() {
     return grade;
   }
};
```

```
/* test-student3.cpp */
#include <stdio.h>
#include <string.h>
#include "labmember3.h"
#include "student3.h"
#include "professor3.h"

void print_labmember(Student s) {
  printf("student name = %s,
  grade = %d\n", s.GetName(),
  s.GetGrade());
}

void print_labmember(Professor p) {
  printf("professor name = %s,
  room = %d\n", p.GetName(),
  p.GetRoom());
}

int main (int argc, char **argv) {
  Professor p1("");
  Student s1("");

  p1.SetName("yamada");
  s1.SetName("suzuki");
  p1.SetRoom(123);
  s1.SetGrade(4);

  print_labmember(s1);
  print_labmember(p1);

  return 0;
}
```

```
/* professor3.h */

class Professor : public LabMember {
 private:
  int room;

 public:
   Professor(char* _name)
  : LabMember(_name) {
     room = 0;
   }
   ~Professor() {
   }

   char* SetName(char* _name) {
     sprintf(name, "Prof. %s", _name);
   }
   void SetRoom(int _room) {
     room = _room;
   }
   int GetRoom() {
     return room;
   }
};
```

これまでのプログラムと同様にコンパイル，実行し以下のような結果を得られるはずである．

```
$ g++ -o test-student3 test-student3.cpp
$ ./test-student3
student name = Mr. suzuki, grade = 4
professor name = Prof. yamada, room = 123
```

B 仮想関数

継承を利用するもう一つの利点として，異なるオブジェクト統一的に扱うことが出来る点である．Student と Professor を構成員とする Lab クラスを考えてみる．

Lab の構成員は Professor クラスと Student クラスの 2 つのインスタンスの可能性があるので，通常は以下のように 2 種類の変数 (professors, students) を用いて構成員を管理することを考える．

```
class Lab {
  private:
   Professor* professors[4];
   int n_professors;
   Student* students[32];
   int n_students;

  public:
   Lab();
   ~Lab();

   void AddProfessor(Professor* new_professor);
   void AddStudent(Student* new_student);
};
```

Lab クラスの定義

しかし，Professor クラスと Student クラスは LabMember を親クラスとして持つので，以下のように LabMember の配列として構成員を管理し，LabMember を引数をしてもつメンバ関数を定義することが出来る．

```
class Lab {
  private:
   LabMember* members[36];
   int n_members;

  public:
   Lab();
   ~Lab();

   void AddMember(LabMember* new_member);
};
```

仮想関数を使った Lab クラスの定義

これで，メンバ変数，メンバ関数の数が大幅に減らすことが出来ている．また，AddMember() メンバ関数の引数はインスタンスへのポインタとなっていることに注意すること．インスタンスへの

ポインタを用いてメンバ変数，メンバ関数にアクセスする場合は，これまでの”.” ではなく，“− >” を用いてアクセスすることが出来る．

仮想関数を使ったプログラムの例を以下に示す．

```
/* lab4.h */

class Lab {
  private:
   LabMember* members[36];
   int n_members;

  public:
   Lab() {
     n_members = 0;
   }
   ~Lab() {}

   void AddMember(LabMember* new_member) {
     members[n_members] = new_member;
     n_members++;
   }
   void PrintMember() {
     int i;
     for(i=0;i<n_members;i++){
        members[i]->PrintInfo();
     }
   }
};
```

```
/* labmember4.h */

class LabMember {
  protected:
   char name[32];

  public:
    LabMember() {} ;
    LabMember(char* _name) {
      strcpy(name, _name);
    }
    ~LabMember() {
    }

    char* SetName(char* _name) {
      sprintf(name, "Mr. %s", _name);
    }
    char* GetName() {
      return name;
    }
    virtual void PrintInfo() {};
};
```

```
/* student4.h */

class Student : public LabMember {
  private:
   int grade;

  public:
    Student() : LabMember() {}
    Student(char* _name)
    : LabMember(_name) {
      grade = 0;
    }
    ~Student() {
    }

    void SetGrade(int _grade) {
      grade = _grade;
    }
    int GetGrade() {
      return grade;
    }
    void PrintInfo () {
      printf("student name = %s, grade =
      %d\n", GetName(), GetGrade());
    }
};
```

```
/* professor4.h */

class Professor : public LabMember {
  private:
   int room;

  public:
    Professor() : LabMember() {}
    Professor(char* _name)
    : LabMember(_name) {
      room = 0;
    }
    ~Professor() {
    }

    char* SetName(char* _name) {
      sprintf(name, "Prof. %s", _name);
    }
    void SetRoom(int _room) {
      room = _room;
    }
    int GetRoom() {
      return room;
    }
    void PrintInfo () {
      printf("professort name = %s, room =
      %d\n", GetName(), GetRoom());
    }
};
```

```
/* test-student4.cpp */
#include <stdio.h>
#include <string.h>
#include "labmember4.h"
#include "student4.h"
#include "professor4.h"
#include "lab4.h"

int main (int argc, char **argv) {
  Lab jsk;
  Professor *p1 = new Professor();
  Student *s1 = new Student();

  p1->SetName("yamada");
  s1->SetName("suzuki");
  p1->SetRoom(123);
  s1->SetGrade(4);
  jsk.AddMember(p1);
  jsk.AddMember(s1);

  jsk.PrintMember();

  delete p1;
  delete s1;
  return 0;
}
```

test-student4.cpp の

```
Professor *p1 = new Professor();
Student *s1 = new Student();
```

の部分は new を利用してメモリを確保するオペレータであり，

```
delete p1;
delete s1;
```

として確保したメモリを開放している．C 言語では malloc/free に対応するが，malloc のようにメモリ量の指定の必要がなく，クラスや型の名前を利用することが出来る．

次はオーバーライドの例を見てみよう．

```
p1->SetName("yamada");
s1->SetName("suzuki");
```

の部分では，1 行目の p1 は Professor のインスタンスへのポインタから呼ばれているので，Professor クラスのメンバ関数が呼ばれている．一方，2 行目では Student のインスタンスへのポインタから呼ばれているが，Student クラスには独自に定義した SetName() メンバ関数は存在しないため，親クラスの LabMember クラスのメンバ関数が呼ばれていることになる．

一方，lab4.h の中の

```
 void PrintMember() {
   int i;
   for(i=0;i<n_members;i++){
```

```
      members[i]->PrintInfo();
    }
  }
```

の部分を見てみよう．members は LabMember* members[36]; として定義されているため，このままでは，LabMember の PrintInfo() というメンバ関数が呼ばれることになる．しかしそれぞれの members は Professor か Student クラスのインスタンスであるため，子クラスで定義された PrintInfo() メンバ関数を呼び出してほしい．そこで，labmember4.h では，

```
virtual void PrintInfo() {};
```

という風にメンバ関数を定義する際に virtual という宣言がなされている．このようなメンバ関数を仮想関数と呼ぶ．仮想関数として宣言されたメンバ関数は，インスタンスの親クラスのポインタを介して呼び出された場合でも，そのインスタンスが生成された型のメンバ関数の定義が呼び出される．また，一度仮想関数として宣言しておくと，それを継承するサブクラスでも自動的に仮想関数となる．

C　フレンドクラス

一般に private, protected なメンバ変数，メンバ関数は他のクラスからアクセスできない．しかしながら Lab と LabMember など関連の強いクラス同士は互いにすべての変数．関数にアクセスできるようになっているほうが便利である．そのようなクラスを friend クラスと呼ぶ．たとえば LabMember を以下のように定義すると，

```
class LabMember {
  protected:
   char name[32];
   friend class Lab;
   ...
```

Lab のメンバ関数の中から LabMember の private, protected なメンバ変数，メンバ関数へのアクセスが可能になり，

```
  void PrintMember() {
    int i;
    for(i=0;i<n_members;i++){
       printf("%s", members[i]->name);
    }
  }
```

のようなコードを書くことが出来る．