

---

---

# Geant4ユーザのためのC++入門

## 演習パッケージ: P03\_C++Tutorial

---

---

Geant4講習会資料

# 本資料に関する注意

---

- 本資料の知的所有権は、高エネルギー加速器研究機構およびGeant4 collaborationが有します
- 以下のすべての条件を満たす場合に限り無料で利用することを許諾します
  - 学校、大学、公的研究機関等における教育および非軍事目的の研究開発のための利用であること
    - ・ Geant4の開発者はいかなる軍事関連目的へのGeant4の利用を拒否します
  - このページを含むすべてのページをオリジナルのまま利用すること
    - ・ 一部を抜き出して配布したり利用してはいけません
  - 誤字や間違いと疑われる点があれば報告する義務を負うこと
- 商業的な目的での利用、出版、電子ファイルの公開は許可なく行えません
- 本資料の最新版は以下からダウンロード可能です
  - <http://geant4.kek.jp/lecture/>
- 本資料に関する問い合わせ先は以下です
  - Email: [lecture-feedback@geant4.kek.jp](mailto:lecture-feedback@geant4.kek.jp)

# はじめに

## ■ このHands-Onの目的

- Geant4を使用するにあたり、知っておいてほしいC++言語のオブジェクト指向要素の最低限の知識を習得する

## ■ オブジェクト指向とは

- プログラムをオブジェクト単位にモジュール化し、作成する技法
  - ・ Geant4の様な大規模なプログラムを作成するのに不可欠
- オブジェクト指向言語の基本要素はオブジェクトとクラス
  - ・ hands-onでこの二つを理解しよう

## ■ Geant4がC++言語を使用している理由

- 確立されたオブジェクト指向言語
- プログラムの処理・実行が高速

## ■ Hands-onの進め方

- Hands-on用コードのzipファイルを解凍して以下のdirectoryに移行する  

Geant4Tutorial2020 ➡ P03 C++Tutorial
- このdirectoryにある演習ファイルをスライドに従って順次実行していく
- 演習ファイルのC++コード編集はVS Codeを使用する(受講者の演習環境に設定済み)
- Hands-onの演習の回答はこのdirectory内の'Z\_AnswersDir'にある

[注] 言葉は真似をして使って習得するのが最短 — 文法は意識せず、使って慣れるのがHands-onの方針

# オブジェクトとは何か？

- オブジェクトとは手続き型言語(例: C言語)の「変数」を概念拡張したもの

[注] C++はCからの派生言語

- 変数 = (指定された**タイプ/型**が決める内容の情報を持つ)
- オブジェクト = (指定された**タイプ/型**で決める内容の情報を持つ) + (情報内容进行操作する関数を持つ)

- C++ではC言語の「変数」を含め、**情報を保持するものは全てオブジェクトと呼ぶ**

- C++が標準として提供する**タイプ/型**のいくつかの例を以下に示す:

[built-inタイプ 例]

タイプ名	情報内容(データ)	操作関数例
int	一つの整数	+, -, *, /, ...
float	一つの小数(単精度)	同上
double	一つの小数(倍精度)	同上
char	一つの文字	同上

[標準ライブラリ提供タイプ 例]

タイプ名	情報内容(データ)	操作関数例
vector	オブジェクト(任意の数)	size, at, ...
complex	一つの複素数	real, imag, +, -, *, / ...
array	オブジェクト(任意の数)	at, size, swap, ...
string	文字列	find, substring, ...

- オブジェクトの作り方 — 手続き型言語の「変数」と同じで、**タイプ/型**を指定して作る

```
int i = 5;
int j { 5 };
vector<double> v{ 3.4, 5.6 };
complex<float> c(1.0f, 2.0f);
```

C言語以来の伝統的な変数の作り方 — C++でもそのまま使える  
C++11以降の**標準的なオブジェクトの作り方** — 上記と同じ意味  
vectorオブジェクト生成: <>の中に保存するオブジェクトタイプを書く  
complexオブジェクト生成: <>の中にreal/imaginaryのタイプを書く

[注] {}の代わりに()を使うこともOK: c++11より前は()しか無かった  
{}(/)を使うことでオブジェクト生成のconstructor関数が呼ばれる

- オブジェクトの使い方

```
int k = i + j;
double f = v.at(1);
int l = v.size();
float r = c.real();
```

整数オブジェクトの加算  
vectorオブジェクト'v'の2番目の要素値をとる (f=5.6)  
vectorオブジェクト'v'のサイズ情報をとる (l=2)  
complexオブジェクトのreal値をとる

# ポインター・タイプ

- C++が標準として提供するタイプ (型)
  - このタイプのオブジェクトはポインタと呼ばれ、オブジェクトのメモリ・アドレス情報を持つ

## ポインタの作成とアクセス

[ Hands-on ] C03\_PointerReference.ccファイルをeditorで開いて実習

```
int i = 123; ← intオブジェクトiを作る
int* ip = &i; ← iへのポインタip (int*タイプ)を'&オペラ'を使い作成 [注1]
int j = *ip; ← ipがポイントしているintオブジェクト(i)をjにコピー [注2]
```

[注1] '&オペラ'はオブジェクトのメモリアドレスを取得する; 通常、使われることは少ない

[注2] '\*オペラ'はポインタからそれが指すオブジェクトを取得する; 'dereferencing'オペラとよばれ、前の行のポインタタイプの宣言(int\*)の'\*'とは全く別のものであることに注意

```
vector<double> v = vector<double>{ 1.2, 2.3, 3.4 }; ← vectorオブジェクトvを作る
vector<double>* vp = &v; ← vへのポインタvp(vector<double>*タイプ)を作成
vector<double> ww = *vp; ← vpがポイントしているvectorオブジェクトをコピー
int vp_Size = vp->size(); ← vpがポイントしているvectorオブジェクトが持つ関数を使う [注3]
double vp_At = vp->at(1);
```

[注3] ポインタ経由でオブジェクトの関数を使う場合は'->'を使う: 直接オブジェクトから関数を使う場合は'.'を使う (演習参照)

- ポインタがなぜ必要なのか – なぜオブジェクトを直接使わないのか?
  - 大情報を持つオブジェクトは、'new'オペラを使い大容量メモリ領域(heap)にしか作れない
    - このオブジェクトへのアクセスはポインタ経由でしかできない
  - 関数の中で作ったオブジェクトを関数外に渡したい場合はポインタを使うのが便利

# レファレンス・タイプ

## ■ C++が標準として提供するタイプ (型)

- このタイプのオブジェクトはレファレンスと呼ばれ、オブジェクトの別名を作りたいときに使用

## ■ レファレンスの作成とアクセス

```
int i = 123; ← intオブジェクトiを作る
int& ir = i; ← iのレファレンス/参照 ir(int&タイプ)を作成
               — iに別名irを付けるという言い方もする
```

[注] レファレンスタイプの宣言(int&)の'&'と'&オペレータ'は全く別のものであることに注意

## ■ レファレンスがなぜ必要なのか – なぜオブジェクトを直接使わないのか？

- 関数に対するオブジェクトを受け渡しはreferenceを使うと高速である
  - オブジェクトを直接渡すと関数はそのコピーを作って処理するので時間がかかる (演習参照)

[注] 上のコード例のようにreferenceでオブジェクトの別名を作り、それを使用することは稀である

# クラスとは何か？

- 前演習で作ったオブジェクトの**タイプ**(型)はC++が標準的に用意しているもの
  - ユーザが**独自のタイプ**を作りたいとき、'class'を使う
    - ユーザが'class'を使って作ったタイプは**ユーザ定義タイプ**とよばれる
  - 本演習では'class'を使い以下の**新しいタイプ**を作り、そのオブジェクトを使ってみる
    - **タイプ名**: Rectangular (長方形)
    - **タイプが保持する情報と操作関数**
      - 情報: 長方形の2辺の長さ (sizeX, sizeY) : このデータはユーザから**アクセス不可**
      - 関数: オブジェクト生成関数(**constructor**)  
面積値(**area**)、形の名前(**shape**)
- これらの関数はユーザから**アクセス可能**
- C04\_WhatIsClass.ccをeditorで開き、以下を参照して**[演習1]**のコード完成させる

<pre>class Rectangular {     public:         Rectangular( double x, double y ) {             sizeX = x; sizeY = y; }         string shape() { return "Rectangular"; }         double area() { return sizeX*sizeY; }     private:         double sizeX, sizeY; };</pre>	<p>← classを使いRectangularタイプを宣言定義する</p> <p>← public下のデータ、関数はユーザが<b>アクセス可能</b></p> <p>← constructor関数の定義: <b>返り値の型の定義はない</b></p> <p>← constructor関数からの入力値をsizeX/Yに設定する</p> <p>← shape関数の定義</p> <p>← area関数の定義</p> <p>← private下のデータ、関数はユーザが<b>アクセス不可</b></p> <p>← sizeX/Yは非公開データ、ユーザが<b>アクセス不可</b></p>
--	--

- 続いて **[演習2]**のコード完成させ、プログラムを実行してみる



# クラス継承とは何か？

- クラスで定義されているmembers(dataとfunctions)を再利用しながら、**新しいクラスを作ること**
  - もとのクラスを**base class**、新しく作るクラスを**derived class**とよぶ
- 本演習ではbase classから二つのderived classesを作り、それらのオブジェクトを生成してみる
  - base class: **Shape\_2P** (2つのパラメーターで表現できる図形のクラス)
  - derived class: **Rectangular**, **Ellipse** (2つのderived classes)
- **C05\_WhatIsInheritance.cc**をeditorで開き、以下を参照して**[演習1、2、3、4]**のコード完成させ

```
class Shape_2P {  
public:  
    string shape(){ return name; }  
    double area(){ return 0.; }  
protected:  
    double sizeX, sizeY;  
    string name = "Shape_2P";  
    const double pi = 3.14159265;  
};
```

← もとのclass (**base class**)を定義する

[注] constructorを定義しないと、C++が自動的にdefaultを作成する

← **derived class**はこれらのmembersを全て共有する

- **public**と宣言されたmembersは**derived class**でも**public**
- **protected**と宣言されたmembersは**derived class**だけが**アクセス可能**

```
class Rectangular : public Shape_2P {  
public:  
    Rectangular ( double x, double y ) {  
        sizeX = x; sizeY = y; name = "Rectangular"; }  
    double area(){ return sizeX * sizeY; }  
};
```

← **derived class - Rectangular**を定義

[注] 'public'で継承することでbase classのprotected membersにアクセス可能となる

← **Rectangular**のconstructorを定義

← **area**関数定義を再定義

```
class Ellipse : public Shape_2P {  
public:  
    Ellipse( double x, double y ) {  
        sizeX = x; sizeY = y; name = "Ellipse"; }  
    double area(){ return sizeX * sizeY * pi/4.0; }  
};
```

← **derived class - Ellipse**を定義

← **Ellipse**のconstructorを定義

← **area**関数定義を再定義

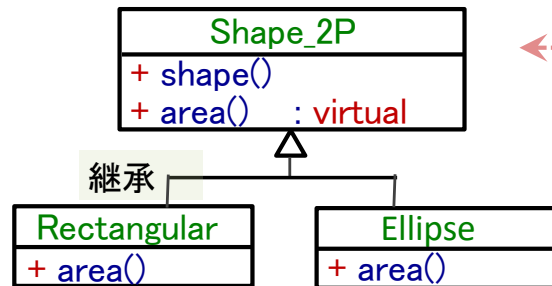


# 多態性(*Polymorphism*)および仮想関数(*Virtual Function*)とは？

## ■ C++のクラス継承での多態性という言葉の意味：

- base classオブジェクトのポインタはderived classオブジェクトをポイントできる
- base classの関数に**virtual**を指定すると、その関数の振る舞いはbase classオブジェクトがポイントしているオブジェクトにより変化する (*dynamic binding*) ← 多態性とよぶ

## ■ 前演習のクラス継承を使って多態性を実装をすると以下のようになる：



```
class Shape_2P {
public:
    string shape(){ return name; }
    virtual double area(){ return 0.; }
protected:
    double sizeX, sizeY;
    string name = "Shape_2P";
    const double pi = 3.14159265;
};
```

```
Shape_2P aShape{};
Rectangular aRect{ 2.0, 3.0 };
Ellipse aElli{ 2.0, 3.0 };
```

上の3つのclassのオブジェクトを作成

```
Shape_2P* pShape;
```

base class のオブジェクトへのポインタを作成

```
pShape = &aShape; pShape->area();
pShape = &aRect;  pShape->area();
pShape = &aElli;  pShape->area();
```

Shape\_2Pのarea()を実行  
Rectangularのarea()を実行  
Ellipseのarea()を実行

[注]  
virtualの指定がないと全て  
Shape\_2Pのarea()が実行される  
(*static binding*)

## ■ 純粋仮想関数(*pure virtual function*)

- 純粋仮想関数とは以下のように定義をされた関数；

```
virtual double area() = 0;
```

area()は未定義で、ユーザの実装が不可欠であることを宣言

- classが純粋仮想関数を持つと、その関数が実装されない限り、そのclassのオブジェクトは作成不可  
← Geant4を実行するためにユーザが必ず作成しなければならないclassは純粋仮想関数を持っている

演習 終了