C++プログラミングI

- 第13回：ポインタの基本
- 担当：二瓶芙巳雄

本日の授業：ポインタ

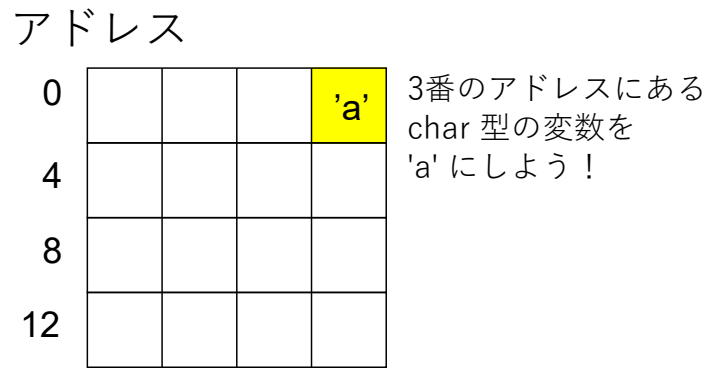
- 多くの人がつまずく内容
  - ※C++の後続の言語では，ポインタを徹底的に排除したものも
- ポインタは，実体を**指し示すだけ**のもの
- なぜポインタを使うのか？
  - スコープを抜けても生存できる変数を動的に生成できるようになる
  - ヒープ領域にアクセスできるようになる
- この授業では，ポインタの**基本**を説明
  - スコープとかヒープ領域とかそういう話はしない
  - 今後の授業でも頑張ってください

```cpp
int main() {
  int num {42};
  int* ptr {&num};

  std::cout << "num: " << num << "\n";
  std::cout << "ptr: " << ptr << "\n";
  std::cout << "&num: " << &num << "\n";
  std::cout << "*ptr: " << *ptr << "\n\n";

  *ptr = 100;
  std::cout << "num: " << num << "\n";
  std::cout << "ptr: " << ptr << "\n";
  std::cout << "&num: " << &num << "\n";
  std::cout << "*ptr: " << *ptr << "\n";
}
```

```
% ./a.out
num: 42
ptr: 0x7fff54ab6c5c
&num: 0x7fff54ab6c5c
*ptr: 42

num: 100
ptr: 0x7fff54ab6c5c
&num: 0x7fff54ab6c5c
*ptr: 100
```

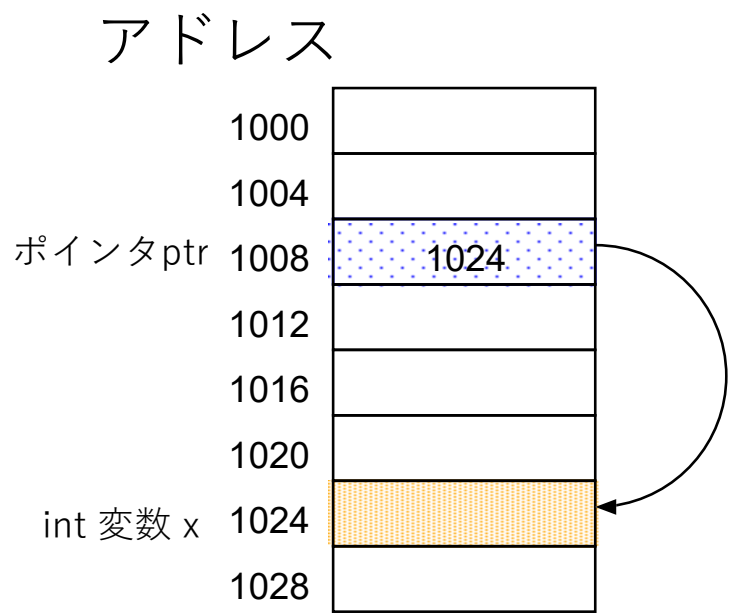
変数とメモリのアドレス

- 変数はコンピュータのメモリ上に作られる
- メモリ中の場所はアドレスで指定する
- アドレスにより変数を間接的に操作できる



ポインタとは

- ポインタはアドレスを覚えておくための変数
  - ポインタは指し示すものの意味
- `ptr` という変数は、1024 という数値を保存しており、変数 `x` が置かれた場所。
- 変数 `x` がメモリ上に作られていて、そのアドレスは1024 番地
- ※変数の宣言とは…
  - メモリの確保：intの場合は4バイトを確保
    - 4バイト＝32ビット
  - 型の解釈：0と1の並びの解釈
  - 変数名とメモリアドレスの関連付け
  - …などなど



ポインタの宣言

- ポインタには型がある
  - 指し示した先にある変数の種類が分かる

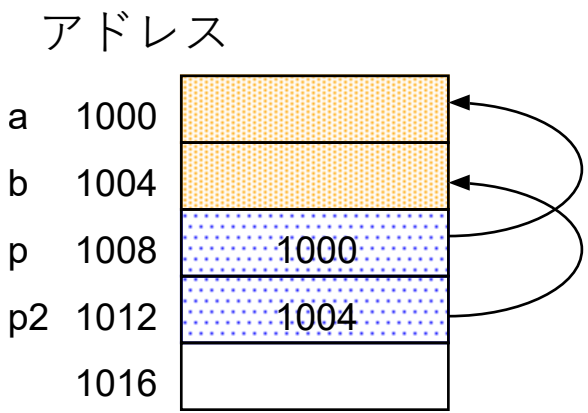
```
int *p1;    // int型変数を指す
char *p2;   // char型変数を指す
double *p3; // double型変数を指す
string *p4; // string型変数を指す
```

- 注意）`*`の両側のスペース文字はあってもなくても良い

```
int *ptr; // スペースが*の前
int* ptr; // スペースが*の後
int*ptr;  // スペースがない
```

アドレス演算子 &

- `&`（アンパサンド）記号の演算子
- 指定した変数のメモリアドレスを得る
- ポインタ変数の初期化, 代入, 引数などに使用する




```cpp
void func( int* a, int* b ) {
    std::cout << "addr a: " << a << "\n";
    std::cout << "addr b: " << b << "\n";
}

int main() {
  int a {10}, b {20};

  int *p { &a }; //ポインタpの初期化

  int *p2;
  p2 = &b;  // 代入

  func( &a, &b ); // 引数
}
```

```
% ./a.out
addr a: 0x7ffdda988b70
addr b: 0x7ffdda988b74
```

&記号の3種類の意味

- **リファレンス指定**：変数宣言で変数名の前に置く場合

```
int& x { a };
void func( int& n ) {...}
```

- **アドレス取得**：式において変数の前に置く場合

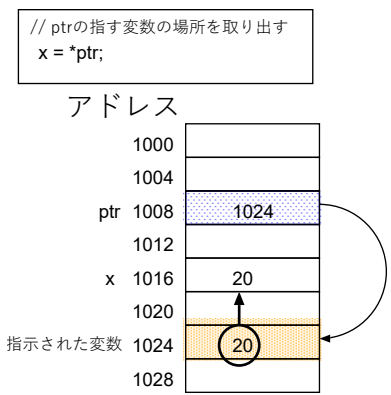
```
ptr = &n;
func( &x );
func( &a[0] );
```

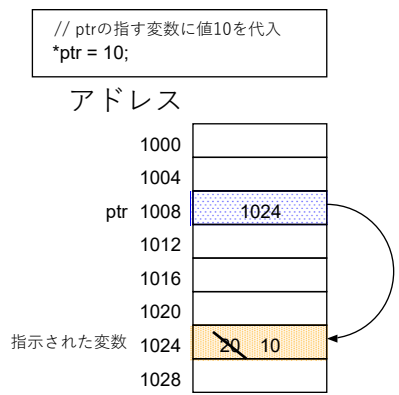
- **ビットごとの論理積**：二つのオペランド（二項演算子）

```
x = y & z;
```

間接演算子 *

- `*`（アスタリスク）記号の演算子
- ポインタ変数が持つアドレスにある変数を操作
- 指し示した変数への値取得と代入






```
1  int main() {
2    int num {20};
3    int* ptr {&num};
4    cout << "num: " << num << "\n";
5    cout << "*ptr: " << *ptr << "\n";
6
7    *ptr = 10;
8    cout << "num: " << num << "\n";
9
10   int x;
11   x = *ptr;
12   cout << "x: " << x << "\n";
13 }
```

```
% ./a.out
num: 20
*ptr: 20
num: 10
x: 10
```

`*`記号の3種類の意味

- **ポインタ変数の宣言**：変数宣言で変数名の前に置く場合

```
int *x;
void func(int* n) { ... }
```

- **間接参照**：式において変数の前に置く場合

```
n = *ptr;
*ptr = 10;
```

- **乗算**：2個の変数の間に置く場合

```
a = y * z;
```

- 注意）`*x = y**p;` は累乗ではない

ヌルポインタとは

- ヌルポインタ：ポインタがどこも指していないことを表す
- ポインタ変数には`nullptr`を初期化/代入しても良い
- ポインタ変数は`nullptr`と比較しても良い
- `nullptr`はアドレス値ゼロを表す
  - `C++98`ではリテラル0を使用していた

```
1  int main() {
2    int *p;  // 宣言しただけでは値は不定
3    // *p = 10; // 未初期化のポインタが指す先へのアクセスは，プログラム暴走の可能性あり！
4
5    int *ptr {nullptr}; // 初期化
6    ptr = nullptr;      // 代入
7
8    if (ptr != nullptr) *ptr = 10;
9  }
```

推奨されるポインタの使い方

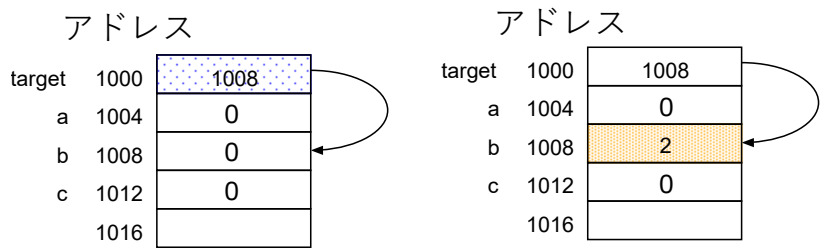
- ポインタ宣言は初期化とともに。
  - 指すべき変数が決まってから宣言する
  - 指すべき相手がないならば`nullptr`で初期化
- ヌルポインタでないことを確認してから使う
  - `nullptr`で初期化したポインタ変数の場合
  - 仮引数がポインタ変数の場合

ポインタ変数は他の変数を指し示すためにある。 無効なアドレスを保持したポインタ変数に、間接演算子を適用すると、プログラムが暴走したり異常終了する可能性がある

```
1  int main() {
2    int *p1; // 非推奨な例
3
4    int *p2 {nullptr}; // 推奨な例①
5
6    int x {120};
7    int *p3 { &x }; // 推奨な例②
8
9
10   std::cout << *p2 << "\n"; // 非推奨な例
11
12   if( p2 != nullptr ) {
13     std::cout << *p2 << "\n"; // 推奨な例
14   }
15 }
```

ポインタの例（局所変数）

- `target` が間接的に変数を更新



※ `a = 2;` とは記述していない．しかしポインタを使用すると，間接的に変数 `a` の値を変更することができる．

```cpp
int main() {
  int a{}, b{}, c{};

  int *target{ &a }; // 初期値はa のアドレス
  ...

  target = &b;  // 左図：指し示す変数の変更
  *target = 2;  // 右図：b を間接的に更新
  ...

  std::cout << "a = " << a << ", "
            << "b = " << b << ", "
            << "c = " << c << "\n";
}
```

```
% ./a.out
a = 0, b = 2, c = 0
```

大域変数の例

- `countup()` は `target` が指している変数を変更
- `Ctrl+D` で終了
- 大域変数は各所からひとつの変数を操作できて便利だが…
  - 大きなプログラムでは、不具合が生じたときに、把握しづらい
- `if (target != nullptr) { ... }`
  - いわゆるヌルチェック

```
1  int *target { nullptr };  // 初めは何も対象としない
2  void countup() {
3    if (target != nullptr) ++(*target);
4  }
5  int main() {
6    int a{}, b{}, c{};
7    for (char ch; std::cin >> ch;) {
8      switch (ch) {
9        case 'a': target = &a; break;
10       case 'b': target = &b; break;
11       case 'c': target = &c; break;
12       default:
13         std::cout << "input error: " << ch << "\n";
14         target = nullptr;
15         break;
16     }
17     countup();
18   }
19   std::cout << a << ", " << b << ", " << c;
20 }
```

```
% ./a.out
a c b a c c
(Ctrl+D)
2, 1, 3
```

ポインタの引数

- 実引数で更新対象を指定する，大域変数を追い出す
- 左：1p前のコード，右：書き換え後のコード

```cpp
int *target { nullptr };  // 初めは何も対象としない
void countup() {
  if (target != nullptr) ++(*target);
}
int main() {
  int a{}, b{}, c{};
  for (char ch; std::cin >> ch;) {
    switch (ch) {
      case 'a': target = &a; break;
      case 'b': target = &b; break;
      case 'c': target = &c; break;
      default:
        std::cout << "input error: " << ch << "\n";
        target = nullptr;
        break;
    }
    countup();
  }
  std::cout << a << ", " << b << ", " << c;
}
```

```cpp
// 大域変数をなくし，等価なコードを実装
void countup(int *target) {
  if (target != nullptr) ++(*target);
}
int main() {
  int a{}, b{}, c{};
  for (char ch; std::cin >> ch;) {
    switch (ch) {
      case 'a': countup(&a); break;
      case 'b': countup(&b); break;
      case 'c': countup(&c); break;
      default:
        std::cout << "input error: " << ch << "\n";

        break;
    }

  }
  std::cout << a << ", " << b << ", " << c;
}
```

vector要素のアドレス

- vectorの各要素もアドレスが与えられている
- `&v[0]` は `&(v[0])` と同じ意味
- 配列要素を指定するかぎ括弧の演算子(`[]`) は、`&` 演算子よりも優先される

```
% ./a.out
a b c x a a b y z z
(Ctrl+D)
3, 2, 1, 4
```

```
1  void countup(int *target) {
2    if (target != nullptr) ++(*target);
3  }
4
5  int main() {
6    std::vector<int> v{0, 0, 0};
7    int x{};
8    for (char ch; std::cin >> ch;) {
9      switch (ch) {
10       case 'a': countup(&v[0]); break;
11       case 'b': countup(&v[1]); break;
12       case 'c': countup(&v[2]); break;
13       default: countup(&x); break;
14     }
15   }
16
17   std::cout << v[0] << ", " << v[1] << ", "
18             << v[2] << ", " << x;
19 }
```

構造体とポインタ①

- 構造体の変数にもアドレスが割り当てられている
- 基本データ型と同じようにポインタを利用可能
- ※ `&num` で得られる値と `&point` で得られる値は見た目が似ているが，本質的に異なる値

```
% ./a.out
num: 3.14
dptr: 0x7ffdb10bad50
&num: 0x7ffdb10bad50
*dptr: 3.14

point: (X, 1, 2)
pointPtr: 0x7ffdb10bad70
&point: 0x7ffdb10bad70
*point: (X, 1, 2)
```

```cpp
struct Point {
  std::string name;
  int x, y;
};
std::ostream& operator<<( std::ostream& out, const Point& p ) {
  return out << "(" << p.name << ", " << p.x << ", " << p.y << ")";
}

int main() {
  double num { 3.14 };
  double* dptr { &num };

  std::cout << "num: " << num << "\n";
  std::cout << "dptr: " << dptr << "\n";
  std::cout << "&num: " << &num << "\n";
  std::cout << "*dptr: " << *dptr << "\n\n";


  Point point {"X", 1, 2};
  Point* pointPtr { &point };

  std::cout << "point: " << point << "\n";
  std::cout << "pointPtr: " << pointPtr << "\n";
  std::cout << "&point: " << &point << "\n";
  std::cout << "*point: " << *pointPtr << "\n\n";
}
```

構造体とポインタ②

- 構造体はデータをまとめるもの
  - データメンバの値に興味
- 構造体のポインタでデータメンバにアクセスするには？
- **アロー演算子** `->`
  - `(*p).x` は `p->x` と省略できる

```
% ./a.out
point.name: X
(*pointPtr).name: X
pointPtr->name: X

point.x: 1
(*pointPtr).x: 1
pointPtr->x: 1
```

```cpp
struct Point {
  std::string name;
  int x, y;
};

int main() {
  Point point {"X", 1, 2};
  Point* pointPtr { &point };

  std::cout << "point.name: " << point.name << "\n";
  std::cout << "(*pointPtr).name: " << (*pointPtr).name << "\n";
  std::cout << "pointPtr->name: " << pointPtr->name << "\n\n";

  std::cout << "point.x: " << point.x << "\n";
  std::cout << "(*pointPtr).x: " << (*pointPtr).x << "\n";
  std::cout << "pointPtr->x: " << pointPtr->x << "\n";
}
```

構造体のポインタ引数とアロー演算子

- 構造体の変数にもアドレスが割り当てられている
- 構造体へのポインタのメンバへのアクセス方法？
  - 構造体ポインタ用の演算子がある
- `(*p).x` は `p->x` と省略できる
  - `*p.x` は `*(p.x)` という意味になってしまう

```
% ./a.out
X,3,5
A,5,7
A,7,10
B,7,9
```

```
1  struct Point { std::string name; int x, y; };
2  void update(Point *p) {
3    if (p != nullptr) {
4      p->x += 2;  // (*p).x += 2; と同じ
5      p->y += 3;  // (*p).y += 3; と同じ
6    }
7  }
8  void print(const Point *p) {
9    if (p != nullptr)
10     std::cout << p->name << ","
11               << p->x << "," << p->y << "\n";
12 }
13
14 int main() {
15   Point a{"X", 1, 2};
16   update(&a);
17   print(&a);
18
19   std::vector<Point> b{{"A", 3, 4}, {"B", 5, 6}};
20   update(&b[0]);
21   print(&b[0]);
22
23   for (auto &x : b) {  // &はリファレンス指定
24     update(&x);        // x は個々の要素
25     print(&x);
26   }
27 }
```

ポインタ引数と リファレンス引数

- どちらも同じことができる
- リファレンス引数の方が記述が簡潔
- 関数の呼び出しだけ見ると，リファレンス引数は値渡しと参照渡しの区別がつかない
- 引数としてアドレスを渡しているので、呼び出した先の関数でアドレスに関連する変数が変更される可能性を認識できる

```
% ./a.out
1
2
```

```cpp
// リファレンスを使った仮引数
void countup1(int& x) {
  ++x;
}

// ポインタを使った仮引数
void countup2(int* y) {
  if (y != nullptr) ++(*y);
}

int main() {
  int z{0};
  countup1(z); // 値渡し？参照渡し？
  std::cout << z << "\n";

  countup2(&z); // アドレスを渡した！
  std::cout << z << "\n";
}
```

演算子の優先順位（基本）

- `&`と`*`演算子の3つの使い方から考える

```
1   // 基本
2   int  a { 2 };
3   int& b { a };  // リファレンス
4   int *c { &a }; // *はポインタ，&はアドレス取得
5   int *x = &a;   // C++98形式
6   a = 2* b;      // 乗算
7   a = 2 * *c;    // 左側が乗算，右側は間接演算子
8   b = b & 1;     // ビットごとの論理積
9   *c = 2;        // 間接演算子
10  a *= 3;        // a = a * 3;
11  b &= 3;        // b = b & 3;
```

演算子の優先順位（複雑な例）

```
1   // やや複雑
2   int x{}, z{};
3   vector ai {0,2,4,8,16};
4   int *ip { &ai[0] }; // int *ip { &(ai[0]) };
5   *ip = *ip + 10;     // (*ip) = (*ip) + 10;
6   *ip += 10;          // (*ip) += 10;
7   *ip = x  + *ip;     // (*ip) = x + (*ip);
8   *ip = x * *ip;      // (*ip) = x * (*ip);
9   x = ++*ip;          // x = ++(*ip);
10  z = *ip++;          // z = *(ip++);
```

※面倒なら括弧で制御する

- 演算子を分類して考える
  - 二項演算: `a + b` のように項が2 個
  - 前置き単項演算： `-5` のように演算子が前に置かれた項
  - 後置き単項演算：`x++` のような数式にないプログラム特有のもの。配列の `a[1]` や関数呼び出しの `func()` も、後置きの単項演算子と考えてよい。
- 後のものほど強い結び付き → 優先度の高い演算子。
  - `*ip++` という式は、 `ip` と `++` が先に結び付いて `*(ip++)` と解釈

アドレス値の出力

- `cout` でアドレスの値を出力できる
- 大域変数, 局所変数, vector 要素は異なる領域を使う
- 変数の大きさに応じて値が変化する
- vector要素は連続したアドレス

```
% ./a.out
&ga = 0x4061e0
&gb = 0x4061e4
&gd = 0x4061f0
&gv[0]= 0x827eb0
&gv[1]= 0x827eb4
&a = 0x7ffeac18991c
&b = 0x7ffeac189918
&d = 0x7ffeac189908
&e = 0x7ffeac189900
ptr = 0x7ffeac18991c
&iv[0]= 0x828ee0
&iv[1]= 0x828ee4
&iv[2]= 0x828ee8
&dv[0]= 0x828f00
&dv[1]= 0x828f08
&dv[2]= 0x828f10
```

```cpp
int ga, gb; double gc, gd; std::vector<int> gv{0, 1, 2, 3};
int main() {
  cout << "&ga = " << &ga << "\n&gb = " << &gb << "\n";
  cout << "&gc = " << &gc << "\n&gd = " << &gd << "\n";

  for (size_t i = 0; i < gv.size(); i++)
    cout << "&gv[" << i << "]= " << &gv[i] << '\n';

  int a, b, c; double d, e, f;
  cout << "&a = " << &a << "\n&b = " << &b << "\n&c = " << &c << "\n"
       << "&d = " << &d << "\n&e = " << &e << "\n&f = " << &f << "\n";

  int *ptr{&a}; cout << "ptr = " << ptr << "\n";
  ptr = &b; cout << "ptr = " << ptr << "\n";
  double *ptr2{&d}; cout << "ptr2 = " << ptr2 << "\n";
  ptr2 = &e; cout << "ptr2 = " << ptr2 << "\n";

  std::vector<int> iv{1, 2, 3, 4, 5};
  for (size_t i = 0; i < iv.size(); i++)
    cout << "&iv[" << i << "]= " << &iv[i] << "\n";

  std::vector<double> dv{1.0, 2.0, 3.0, 4.0, 5.0};
  for (size_t i = 0; i < dv.size(); i++)
    cout << "&dv[" << i << "]= " << &dv[i] << "\n";
  char ch{'a'}; cout << "&ch = "
    << static_cast<const void *>(&ch) << "\n";  // 変換すれば出力可能
}
```