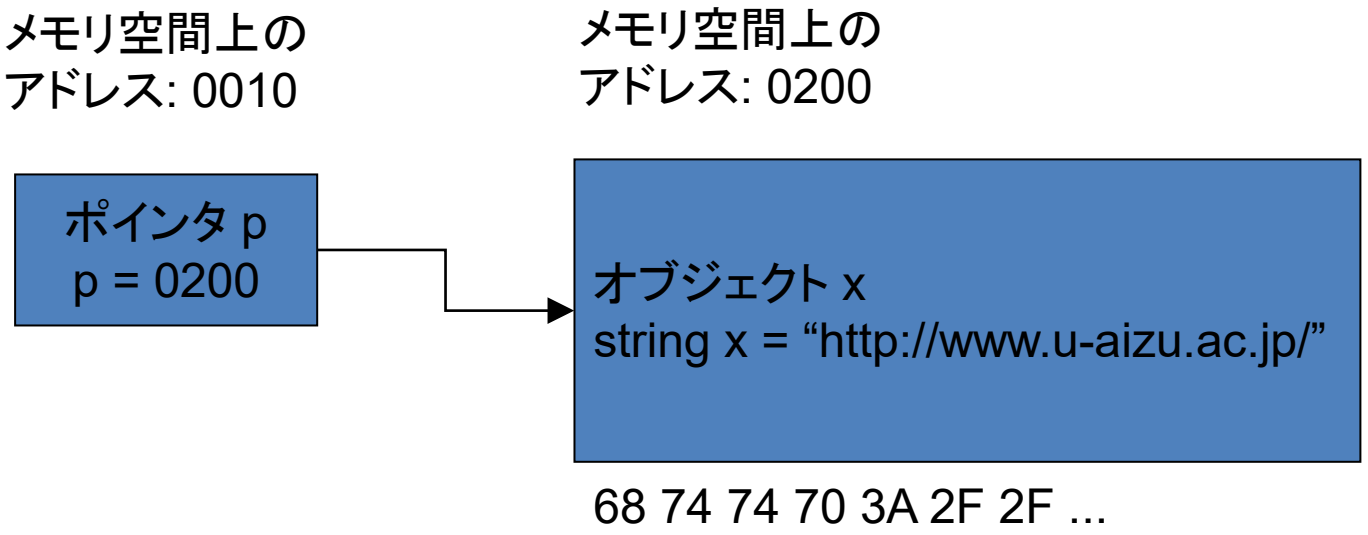
第10章「メモリ管理と低レベルのデータ構造」

主なトピック

- ポインタ
- ファイルの入出力
- メモリ管理

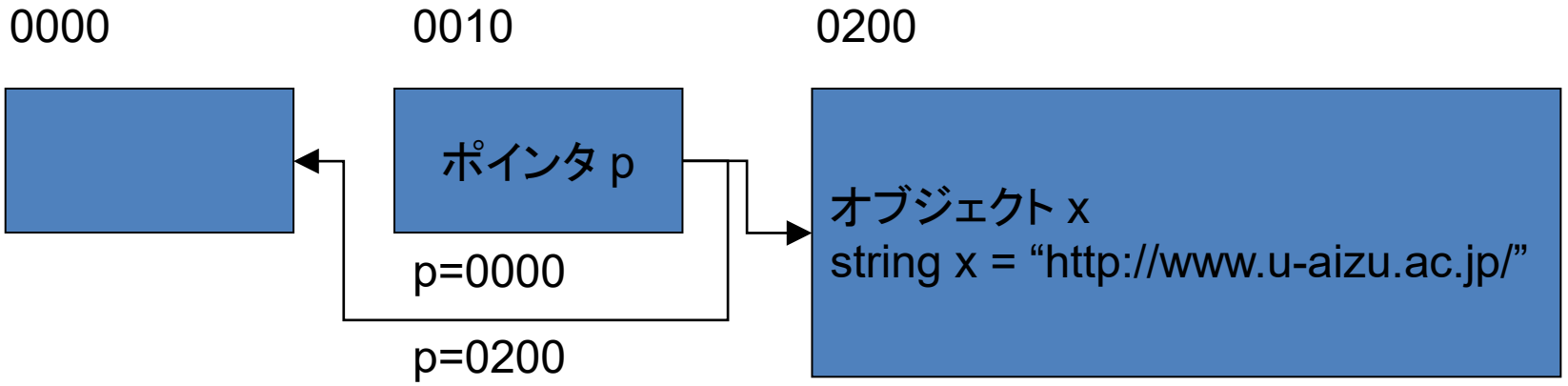
ポインタとは

- ポインタはオブジェクト（変数）のメモリ空間上のアドレスを表す
  - ポインタ p がオブジェクト x を指す
  - p は x を指すポインタ



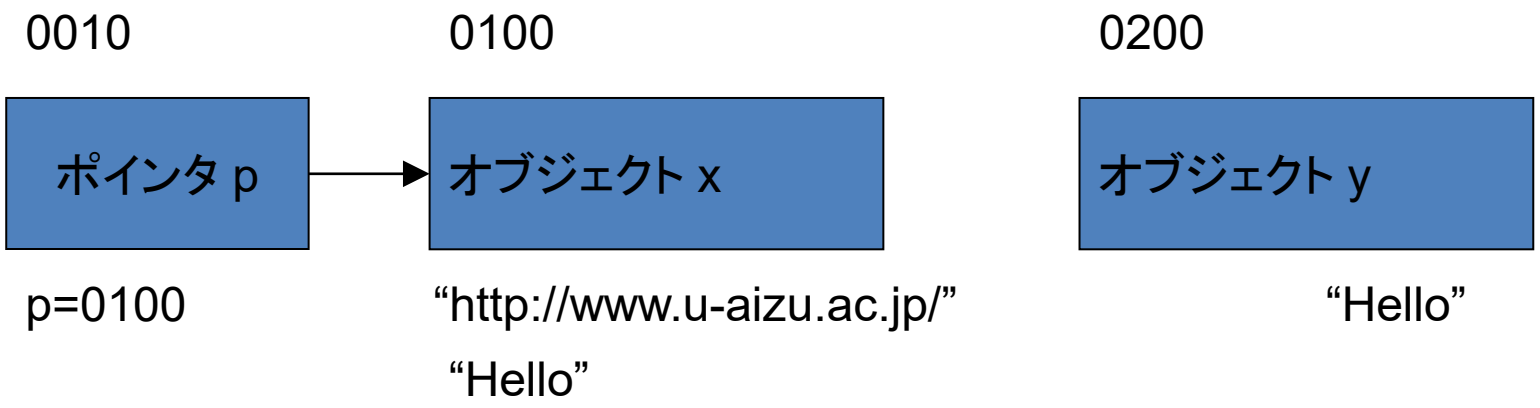
アドレス演算子

- xがオブジェクトとすると，xのアドレスは&x
  - &はアドレス演算子
  - p = &x;
    - ポインタ変数 p に x のアドレスを代入
    - p がオブジェクト x を指すようにする



デリファレンス演算子

- pがアドレスだとすると，pが指すオブジェクトは*p
  - *はデリファレンス演算子
  - *p = y;
  - ポインタ変数 p が指すオブジェクトの内容に y の内容を代入する



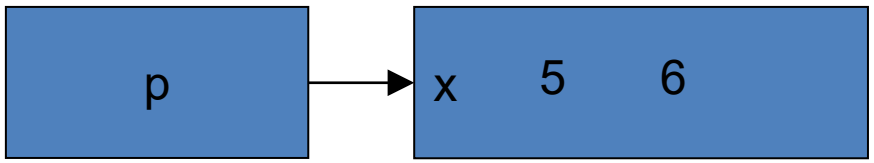
ポインタ

- int　x;
  - x は int 型のオブジェクト（変数）
- int　*p;
  - *p が int 型を意味するので，pは int 型のオブジェクトを指すポインタ
- int*　p;
  - 上の同じ意味．pがポインタを意味することを強調するため，こう書かれることが多い

ポインタを用いたプログラムの例(1)

```
int  x = 5;
int* p = &x;
cout << “x=” << x << "*p=" << *p << endl;
*p = 6;
cout << "x=" << x << "*p=" << *p << endl;
```

- 出力は,
  - x=5 *p=5
  - x=6 *p=6



ポインタを用いたプログラムの例(2)

```
int  x = 5, y = 7;
int* p = &x;
cout << “x=” << x << "*p=" << *p << endl;
p = &y;
cout << “x=” << x << "*p=" <<  *p << endl;
```

- 出力は,
  - x=5 *p=5
  - x=5 *p=7

関数へのポインタの使用例

- 呼び出し側
  - 下の第3引数は，isEvenという関数へのポインタ

```
my_partition( c.begin(), c.end(), isEven );

bool isEven( int num )
{
  if( num % 2 == 0 ){
    return( true );
  }
  return( false );
}
```

関数へのポインタの使用例

- 呼び出された側
  - Pr は，int 型の引数を，bool 型の戻り値を持つ関数へのポインタ（の型）

```
template < class Bi, class Pr >
Bi my_partition( Bi b, Bi e, Pr p)
{
    //  ここまでも省略
   if( p( *b ) == false )  {
      while( b != e )  {
    //  以下も省略
}
```

関数のポインタ

- bool (*fp)(int)
  - int 型の引数を取り，bool 型の戻り値を返す関数への，fp という名前のポインタ
- fp = &isEven;
  - fp に isEven という関数のポインタを代入し，fp が isEven を指すようにする
- fp = isEven;
  - こちらの書き方も可能．上と同じ意味
  - 関数のポインタだけは，特別
- bool a = (*fp)(i);
  - ポインタ変数を使った関数の呼び出し
  - 引数が i，戻り値が a に入る
- bool b = fp(i);
  - こちらも上と同じ意味．
  - 関数のポインタだけは，特別

配列

- 標準ライブラリではなく，言語そのものにあるコンテナの一種
- 1つ以上の同じ型のオブジェクト（型）を保持
- 配列の要素数は，コンパイル時に決められていなければならず，途中で増やしたら減らしたりできない
- size_t
  - 配列の要素数を表す型．実は unsigned 型，参考：クラスの size_type
  - #include <cstddef>
  - const size_t NDim = 3;
  - double coords [NDim];
- 配列の名前は，配列の最初の要素を表すポインタ
  - *coords = 1.5;
    - 配列の先頭の要素，coords[0] に 1.5 が代入される
  - coords + 1
    - 配列の2番目の要素を指すポインタ

ポインタの算術

- vector<double> v;
- copy(coords, coords + NDim, back_inserter(v));
  - coordsの先頭から最後の要素までを，v の末尾にコピーする
  - coords + NDim は，一番最後の要素の次を指している．有効な要素ではない．
  - 参考：イテレータの v.end()

ポインタの算術

- p と q を配列のポインタとして，
  - p - qは p と q の指す要素間の距離
- ptrdiff_t
  - ポインタの指す要素間の距離を表す型
  - #include <cstddef>
- ポインタは，イテレータの一種とみなすことができる
  - コンテナ用の標準ライブラリの関数の引数（イテレータ）に，ポインタを使うことができる

インデックスと配列の初期化

```
const int month_length[ ] = {
        31,28,31,30,31,30,
        31,31,30,31,30,31
};
```

- 明示的に要素を与えて初期化するときは，要素数を省略できる
- p[n] は *(p + n) と同じ

main関数の引数

- 例えば, g++ -o test10 test10.cc
  - コマンドライン引数の数は4
  - “g++” “-o” “test10” “test10.cc”
- コマンドライン引数は空白で区切られる

main関数の引数

- コマンドライン引数をプログラムで使うためには
- int main(int argc, char* argv[ ])
  - argc: コマンドライン引数の数
  - argv: コマンドライン引数の内容（文字列）
  - g++ の例だと,
    - argc = 4;
    - argv[0] = “g++” argv[1] = “-o” argv[2] = “test10” argv[3] = “test10.cc”
  - argv[i]が文字列（char *型）,それが配列になっているので char*[]という型

コマンドライン引数の例

```
//  3つの引数をとるプログラム
// （コマンド名　入力ファイル名　出力ファイル名）
int main( int argc, char *argv[] )
{
    if( argc != 3)    {
            cerr << “Error” << endl;
            return -1;
    }
    ifstream infile( argv[1] );
    ofstream outfile( argv[2] );
}
```

- cerr は標準エラー出力：画面にエラーメッセージを出力するときに使われる．ファイルにリダイレクトされない

ファイルの読み書き

- #include <fstream>
- ifstream infile(argv[1]);
  - ifstream 型のinfileという名前の変数.
  - argv[1]という名前のファイルに読み込みでアクセスするときに使用
  - その後は，標準入力ストリーム cin と同じように使える
    - string FirstName, LastName, ID;
    - infile >> FirstName >> LastName >> ID;
- ofstream outfile(argv[2]);
  - ofstream 型のoutfileという名前の変数.
  - argv[2]という名前のファイルに書き込みでアクセスするときに使用
  - その後は，標準出力ストリーム cout と同じように使える
    - outfile << “|” << FirstName << “ | ” << LastName “ |” << ID << “|”<< endl;

メモリ管理

- 自動メモリ管理
  - ローカル変数に適用される
  - ブロックが終了すると自動的にメモリを開放し，その内容は無効になる

```
//  悪い例
int*  test_func1()
{
    int  x;
    return &x;
}
//  関数から出るとx は開放され，関数の戻り値は無意味
```

メモリ管理

- 静的メモリ管理
  - 関数が始めて実行されたときに初期化
  - プログラムが終了するまで，内容は保持

```
//   正しく動作する例
int*  test_func2()
{
     static int  x;
     return &x;
}
//  関数から出ても x は保持され，関数の戻り値は利用可能
```

オブジェクトの生成と破棄

- new
  - オブジェクト（メモリ）の動的な確保
- delete
  - 動的に確保したオブジェクトの破棄
- int *p = new int(42);
  - int型で，値が42のオブジェクト（メモリ）を確保
  - そのアドレスを p に代入
- delete p;
  - アドレスが p のオブジェクトを破棄（開放）

オブジェクトの生成と破棄

```
//  関数を利用した例
int main()
{
    int *p = test_func3();
    *p = 10;
    //  使い終わったら，開放すること
    delete p;
}

//  int型のオブジェクトを生成し，そのポインタを戻す関数
int test_func3()    {
    return new int(0);
}
```

配列の確保

- int *p = new int [64];
  - 要素数が64のint型のオブジェクト（配列）の確保
- delete p[];
  - 配列全体のオブジェクトの開放（破棄）
  - 参考：delete p だとpが指す要素だけ