Special Classes, Functions, dan Pointers

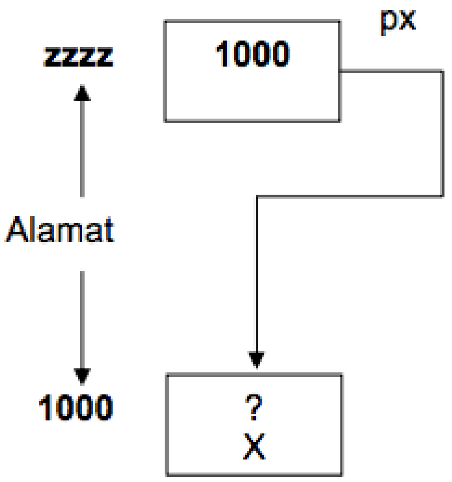
Pertemuan 12

Outline

- Special Classes, Functions, and Pointers
  - Static Member Data
  - Static Member Functions
  - Containment of Classes
  - Pointers to Functions
  - Arrays of Pointers to Functions
  - Passing Pointers to Functions to Other Functions
  - Pointers to Member Functions
  - Arrays of Pointers to Member Functions

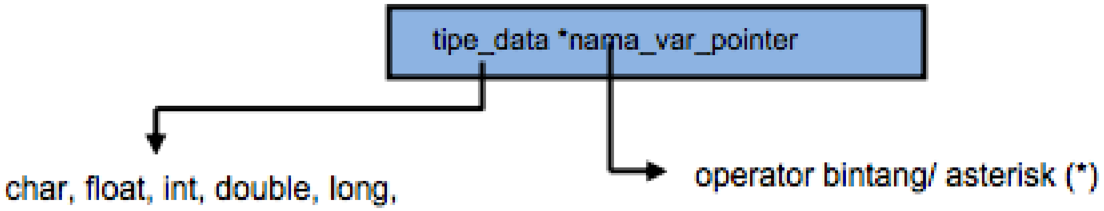
Dasar Teori

- Variabel pointer sering dikatakan sebagai variabel yang menunjuk ke obyek lain. Pada kenyataan yang sebenarnya, variabel pointer berisi alamat dari suatu obyek lain (yaitu obyek yang dikatakan ditunjuk oleh pointer).
- Contoh, px adalah variable pointer dan x adalah variabel yang ditunjuk oleh px. Kalau x berada pada alamat memori (alamat awal) 1000, maka px akan berisi 1000.



Dasar Teori

- Suatu variabel pointer dideklarasikan dengan bentuk sebagai berikut :



tipe_data *nama_var_pointer

char, float, int, double, long,

operator bintang/ asterisk (*)

- Dimana type merupakan tipe dari data yang ditunjuk, bukan tipe dari pointernya. Dengan tipe dapat berupa sembarang tipe yang sudah dibahas pada bab-bab sebelumnya.
- Dalam melakukan pemrograman menggunakan pointer, yang pertama perlu dilakukan adalah dengan melakukan inisialisasi pointer tersebut. Untuk lebih jelasnya perhatikan contoh dibawah ini:

int number; int *tommy = &number;

Dasar Teori

- C++ memperbolehkan operasi dengan pointer pada function. Kegunaan yang utama adalah untuk memberikan satu function sebagai parameter untuk function lainnya.
- Deklarasi pointer untuk function sama seperti prototype function kecuali nama function dituliskan diantara tanda kurung () dan operator asterisk (*) diberikan sebelum nama.

1. Static Member Data

StaticCat.cpp

```
#include <iostream>

class Cat
{
        public:
                Cat(int newAge = 1):age(newAge){ howManyCats++; }
                virtual ~Cat() { howManyCats—; }
                virtual int getAge() { return age; }
                virtual void setAge(int newAge) { age = newAge; }
                static int howManyCats;

        private:
                int age;
};

int Cat::howManyCats = 0;

```

...

```
int main()
{
      const int maxCats = 5;
      Cat *catHouse[maxCats];
      int i;
      for (i = 0; i < maxCats; i++)
                catHouse[i] = new Cat(i);

      for (i = 0; i < maxCats; i++)
      {
                std::cout << “There are “;
                std::cout << Cat::howManyCats;
                std::cout << “ cats left!\n”;
                std::cout << “Deleting the one which is “;
                std::cout << catHouse[i]->getAge();
                std::cout << “ years old\n”;
                delete catHouse[i];
                catHouse[i] = 0;
      }
      return 0;
}
```

2. Static Member Functions

StaticFunction.cpp

```
#include <iostream>

class Cat
{
       public:
                 Cat(int newAge = 1):age(newAge){ howManyCats++; }
                 virtual ~Cat() { howManyCats–; }
                 virtual int gGetAge() { return age; }
                 virtual void setAge(int newAge) { age = newAge; }
                 static int getHowMany() { return howManyCats; }
       private:
                 int age;
                 static int howManyCats;
};

int Cat::howManyCats = 0;

void countCats();

```

...

```
int main()
{
      const int maxCats = 5;
      Cat *catHouse[maxCats];
      int i;
      for (i = 0; i < maxCats; i++)
      {
                catHouse[i] = new Cat(i);
                countCats();
      }

      for (i = 0; i < maxCats; i++)
      {
                delete catHouse[i];
                countCats();
      }
      return 0;
}

void countCats()
{
      std::cout << “There are “ << Cat::getHowMany()
      << “ cats alive!\n”;
}
```

3. Containment of Classes

String.hpp

```
#include <iostream>
#include <string.h>

class String
{
        public:
        // constructors
        String();
        String(const char *const);
        String(const String&);
        ~String();

        // overloaded operators
        char& operator[](int offset);
        char operator[](int offset) const;
        String operator+(const String&);
        void operator+=(const String&);
        String& operator= (const String &);

        // general accessors
        int getLen() const { return len; }
        const char* getString() const { return value; }
        // static int constructorCount;

        private:
        String(int); // private constructor
        char* value;
        int len;
};

```

…

```
// default constructor creates string of 0 bytes
String::String()
{
        value = new char[1];
        value[0] = ‘\0’;
        len = 0;
        // std::cout << “\tDefault string constructor\n”;
        // constructorCount++;
}

// private (helper) constructor, used only by
// class functions for creating a new string of
// required size. Null filled.
String::String(int len)
{
        value = new char[len + 1];
        int i;
        for (i = 0; i < len; i++)
        value[i] = ‘\0’;
        len = len;
        // std::cout << “\tString(int) constructor\n”;
        // constructorCount++;
}

```

...

```
String::String(const char* const cString)
{
        len = strlen(cString);
        value = new char[len + 1];
        int i;
        for (i = 0; i < len; i++)
        value[i] = cString[i];
        value[len] = ‘\0’;
        // std::cout << “\tString(char*) constructor\n”;
        // constructorCount++;
}

String::String(const String& rhs)
{
        len = rhs.getLen();
        value = new char[len + 1];
        int i;
        for (i = 0; i < len; i++)
        value[i] = rhs[i];
        value[len] = ‘\0’;
        // std::cout << “\tString(String&) constructor\n”;
        // constructorCount++;
}

```

…

```
String::~String()
{
      delete [] value;
      len = 0;
      // std::cout << “\tString destructor\n”;
}

// operator equals, frees existing memory
// then copies string and size
String& String::operator=(const String &rhs)
{
      if (this == &rhs)
                return *this;
      delete [] value;
      len = rhs.getLen();
      value = new char[len + 1];
      int i;
      for (i = 0; i < len; i++)
                value[i] = rhs[i];
      value[len] = ‘\0’;
      return *this;
      // std::cout << “\tString operator=\n”;
}

```

...

```
//non constant offset operator, returns
// reference to character so it can be
// changed!
char& String::operator[](int offset)
{
     if (offset > len)
     return value[len - 1];
     else
     return value[offset];
}

// constant offset operator for use
// on const objects (see copy constructor!)
char String::operator[](int offset) const
{
     if (offset > len)
               return value[len-1];
     else
               return value[offset];
}

```

...

```
// creates a new string by adding current
// string to rhs
String String::operator+(const String& rhs)
{
     int totalLen = len + rhs.getLen();
     int i, j;
     String temp(totalLen);
     for (i = 0; i < len; i++)
               temp[i] = value[i];
     for (j = 0; j < rhs.getLen(); j++, i++)
               temp[i] = rhs[j];
     temp[totalLen] = ‘\0’;
     return temp;
}

// changes current string, returns nothing
void String::operator+=(const String& rhs)
{
     int rhsLen = rhs.getLen();
     int totalLen = len + rhsLen;
     int i, j;
     String temp(totalLen);
     for (i = 0; i < len; i++)
     temp[i] = value[i];
     for (j = 0; j < rhs.getLen(); j++, i++)
     temp[i] = rhs[i - len];
     temp[totalLen] = ‘\0’;
     *this = temp;
}

// int String::constructorCount = 0;
```

Employee.cpp

```
#include “String.hpp”

class Employee
{
        public:
                Employee();
                Employee(char *, char *, char *, long);
                ~Employee();
                Employee(const Employee&);
                Employee& operator=(const Employee&);

                const String& getFirstName() const { return firstName; }
                const String& getLastName() const { return lastName; }
                const String& getAddress() const { return address; }
                long getSalary() const { return salary; }

                void setFirstName(const String& fName)
                { firstName = fName; }
                void setLastName(const String& lName)
                { lastName = lName; }
                void setAddress(const String& newAddress)
                { address = newAddress; }
                void setSalary(long newSalary) { salary = newSalary; }
        private:
                String firstName;
                String lastName;
                String address;
                long salary;
};
```

...

```
Employee::Employee():
firstName(“”),
lastName(“”),
address(“”),
salary(0)
{}

Employee::Employee(char* newFirstName, char* newLastName,
char* newAddress, long newSalary):
firstName(newFirstName),
lastName(newLastName),
address(newAddress),
salary(newSalary)
{}

Employee::Employee(const Employee& rhs):
firstName(rhs.getFirstName()),
lastName(rhs.getLastName()),
address(rhs.getAddress()),
salary(rhs.getSalary())
{}

Employee::~Employee() {}

Employee& Employee::operator=(const Employee& rhs)
{
       if (this == &rhs)
                  return *this;

       firstName = rhs.getFirstName();
       lastName = rhs.getLastName();
       address = rhs.getAddress();
       salary = rhs.getSalary();

       return *this;
}

```

…

```
int main()
{
      Employee edie(“Jane”, “Doe”, “1461 Shore Parkway”, 20000);
      edie.setSalary(50000);
      String lastName(“Levine”);
      edie.setLastName(lastName);
      edie.setFirstName(“Edythe”);

      std::cout << “Name: “;
      std::cout << edie.getFirstName().getString();
      std::cout << “ “ << edie.getLastName().getString();
      std::cout << “.\nAddress: “;
      std::cout << edie.getAddress().getString();
      std::cout << “.\nSalary: “ ;
      std::cout << edie.getSalary() << “\n”;
      return 0;
}
```

4. Pointers to Functions

FunctionPointer.cpp

```
#include <iostream>

void square(int&, int&);
void cube(int&, int&);
void swap(int&, int&);
void getVals(int&, int&);
void printVals(int, int);

int main()
{
        void (*pFunc)(int&, int&);
        bool fQuit = false;

        int valOne = 1, valTwo = 2;
        int choice;
        while (fQuit == false)
        {
                std::cout << “(0) Quit (1) Change Values “
                << “(2) Square (3) Cube (4) Swap: “;
                std::cin >> choice;
```

…

```
        switch (choice)
        {
            case 1:
            pFunc = getVals;
            break;
            case 2:
            pFunc = square;
            break;
            case 3:
            pFunc = cube;
            break;
            case 4:
            pFunc = swap;
            break;
            default :
            fQuit = true;
            break;
        }

        if (fQuit)
            break;

        printVals(valOne, valTwo);
        pFunc(valOne, valTwo);
        printVals(valOne, valTwo);
    }
    return 0;
}

```

…

```
void printVals(int x, int y)
{
      std::cout << “x: “ << x << “ y: “ << y << “\n”;
}

void square(int &rX, int &rY)
{
      rX *= rX;
      rY *= rY;
}

void cube(int &rX, int &rY)
{
      int tmp;

      tmp = rX;
      rX *= rX;
      rX = rX * tmp;

      tmp = rY;
      rY *= rY;
      rY = rY * tmp;
}

```

…

```
void swap(int &rX, int &rY)
{
      int temp;
      temp = rX;
      rX = rY;
      rY = temp;
}

void getVals(int &rValOne, int &rValTwo)
{
      std::cout << “New value for valOne: “;
      std::cin >> rValOne;
      std::cout << “New value for valTwo: “;
      std::cin >> rValTwo;
}
```

5. Arrays of Pointers to Functions

ArrayFunction.cpp

```cpp
#include <iostream>

void square(int&, int&);
void cube(int&, int&);
void swap(int&, int&);
void getVals(int&, int&);
void printVals(int, int);

int main()
{
        int valOne=1, valTwo=2;
        int choice, i;
        const int maxArray = 5;
        void (*pFuncArray[maxArray])(int&, int&);

        for (i=0;i < maxArray; i++)
        {
                std::cout << “(1) Change Values “
                << “(2) Square (3) Cube (4) Swap: “;
                std::cin >> choice;
```

…

```
        switch (choice)
        {
            case 1:
                pFuncArray[i] = getVals;
                break;
            case 2:
                pFuncArray[i] = square;
                break;
            case 3:
                pFuncArray[i] = cube;
                break;
            case 4:
                pFuncArray[i] = swap;
                break;
            default:
                pFuncArray[i] = 0;
        }
    }

    for (i = 0; i < maxArray; i++)
    {
        pFuncArray[i](valOne, valTwo);
        printVals(valOne, valTwo);
    }
    return 0;
}

```

…

```
void printVals(int x, int y)
{
      std::cout << “x: “ << x << “ y: “ << y << “\n”;
}

void square(int &rX, int &rY)
{
      rX *= rX;
      rY *= rY;
}

void cube(int &rX, int &rY)
{
      int tmp;

      tmp = rX;
      rX *= rX;
      rX = rX * tmp;

      tmp = rY;
      rY *= rY;
      rY = rY * tmp;
}

```

…

```
void swap(int &rX, int &rY)
{
      int temp;
      temp = rX;
      rX = rY;
      rY = temp;
}

void getVals(int &rValOne, int &rValTwo)
{
      std::cout << “New value for valOne: “;
      std::cin >> rValOne;
      std::cout << “New value for valTwo: “;
      std::cin >> rValTwo;
}
```

6. Passing Pointers to Functions to Other Functions

FunctionPasser.cpp

```
#include <iostream>

void square(int&,int&);
void cube(int&, int&);
void swap(int&, int&);
void getVals(int&, int&);
void printVals(void (*)(int&, int&),int&, int&);

int main()
{
        int valOne=1, valTwo=2;
        int choice;
        bool fQuit = false;

        void (*pFunc)(int&, int&);

        while (fQuit == false)
        {
                std::cout << “(0) Quit (1) Change Values “
                << “(2) Square (3) Cube (4) Swap: “;
                std::cin >> choice;
```

…

```
        switch (choice)
        {
            case 1:
                pFunc = getVals;
                break;
            case 2:
                pFunc = square;
                break;
            case 3:
                pFunc = cube;
                break;
            case 4:
                pFunc = swap;
                break;
            default:
                fQuit = true;
                break;
        }
        if (fQuit == true)
            break;
        printVals(pFunc, valOne, valTwo);
    }

    return 0;
}

void printVals(void (*pFunc)(int&, int&),int& x, int& y)
{
    std::cout << “x: ” << x << “ y: ” << y << “\n”;
    pFunc(x, y);
    std::cout << “x: ” << x << “ y: ” << y << “\n”;
}

```

...

```
void square(int &rX, int &rY)
{
       rX *= rX;
       rY *= rY;
}

void cube(int &rX, int &rY)
{
       int tmp;

       tmp = rX;
       rX *= rX;
       rX = rX * tmp;

       tmp = rY;
       rY *= rY;
       rY = rY * tmp;
}

void swap(int &rX, int &rY)
{
       int temp;
       temp = rX;
       rX = rY;
       rY = temp;
}

void getVals(int &rValOne, int &rValTwo)
{
       std::cout << “New value for valOne: “;
       std::cin >> rValOne;
       std::cout << “New value for valTwo: “;
       std::cin >> rValTwo;
}
```

7. Pointers to Member Functions

MemberPointer.cpp

```
#include <iostream>

enum BOOL {FALSE, TRUE};

class Mammal
{
        public:
                Mammal():age(1) { }
                virtual ~Mammal() { }
                virtual void speak() const = 0;
                virtual void move() const = 0;
        protected:
                int age;
};

class Dog : public Mammal
{
        public:
                void speak() const { std::cout << “Woof!\n”; }
                void move() const { std::cout << “Walking to heel ...\n”; }
};
```

…

```
class Cat : public Mammal
{
      public:
                void speak() const { std::cout << “Meow!\n”; }
                void move() const { std::cout << “Slinking...\n”; }
};

class Horse : public Mammal
{
      public:
                void speak() const { std::cout << “Winnie!\n”; }
                void move() const { std::cout << “Galloping ...\n”; }
};

int main()
{
      void (Mammal::*pFunc)() const = 0;
      Mammal* ptr = 0;
      int animal;
      int method;
      bool fQuit = false;

      while (fQuit == false)
      {
                std::cout << “(0) Quit (1) Dog (2) Cat (3) Horse: “;
                std::cin >> animal;
```

…

```
        switch (animal)
        {
            case 1:
            ptr = new Dog;
            break;
            case 2:
            ptr = new Cat;
            break;
            case 3:
            ptr = new Horse;
            break;
            default:
            fQuit = true;
            break;
        }
        if (fQuit)
            break;

        std::cout << “(1) Speak (2) Move: ”;
        std::cin >> method;
        switch (method)
        {
            case 1:
                pFunc = &Mammal::speak;
                break;
            default:
                pFunc = &Mammal::move;
                break;
        }

        (ptr->*pFunc)();
        delete ptr;
    }
    return 0;
}
```

8. Arrays of Pointers to Member Functions

MPFunction.cpp

```
#include <iostream>

class Dog
{
        public:
                void speak() const { std::cout << “Woof!\n”; }
                void move() const { std::cout << “Walking to heel ...\n”; }
                void eat() const { std::cout << “Gobbling food ...\n”; }
                void growl() const { std::cout << “Grrrrr\n”; }
                void whimper() const { std::cout << “Whining noises ...\n”; }
                void rollOver() const { std::cout << “Rolling over ...\n”; }
                void playDead() const
                { std::cout << “Is this the end of Little Caesar?\n”; }
};

typedef void (Dog::*PDF)() const;

```

…

```
int main()
{
        const int maxFuncs = 7;
        PDF dogFunctions[maxFuncs] =
        {       &Dog::speak,
                &Dog::move,
                &Dog::eat,
                &Dog::growl,
                &Dog::whimper,
                &Dog::rollOver,
                &Dog::playDead
        };

        Dog* pDog =0;
        int method;
        bool fQuit = false;

        while (!fQuit)
        {
                std::cout << “(0) Quit (1) Speak (2) Move (3) Eat (4) Growl”;
                std::cout << “ (5) Whimper (6) Roll Over (7) Play Dead: “;
                std::cin >> method;
                if (method == 0)
                {
                        fQuit = true;
                        break;
                }
                else
                {
                        pDog = new Dog;
                        (pDog->*dogFunctions[method - 1])();
                        delete pDog;
                }
        }
        return 0;
}
```

9. LinkedList

LinkedList.cpp

```
// Demonstrates an object-oriented approach to
// linked lists. The list delegates to the node.
// The node is an abstract data type. Three types of
// nodes are used, head nodes, tail nodes and internal
// nodes. Only the internal nodes hold data.
//
// The Data class is created to serve as an object to
// hold in the linked list.
//
#include <iostream>

enum { kIsSmaller, kIsLarger, kIsSame };

// Data class to put into the linked list
// Any class in this linked list must support two
// functions: show (displays the value) and compare
// (returns relative position)
class Data
{
      public:
                Data(int newVal):value(newVal) {}
                ~Data() {}
                int compare(const Data&);
                void show() { std::cout << value << “\n”; }
      private:
                int value;
};

```

...

```
// Compare is used to decide where in the list
// a particular object belongs.
int Data::compare(const Data& otherData)
{
      if (value < otherData.value)
                return kIsSmaller;
      if (value > otherData.value)
                return kIsLarger;
      else
                return kIsSame;
}

// forward declarations
class Node;
class HeadNode;
class TailNode;
class InternalNode;

// ADT representing the node object in the list.
// Every derived class must override insert and show.
class Node
{
      public:
                Node() {}
                virtual ~Node() {}
                virtual Node* insert(Data* data) = 0;
                virtual void show() = 0;
      private:
};

```

…

```
// This is the node that holds the actual object.
// In this case the object is of type Data.
// We’ll see how to make this more general when
// we cover templates.
class InternalNode : public Node
{
      public:
                InternalNode(Data* data, Node* next);
                virtual ~InternalNode() { delete next; delete data; }
                virtual Node* insert(Data* data);
                virtual void show()
                { data->show(); next->show(); } // delegate!

      private:
                Data* data; // the data itself
                Node* next; // points to next node in the linked list
};

// All the constructor does is to initialize
InternalNode::InternalNode(Data* newData, Node* newNext):
data(newData), next(newNext)
{
}

```

…

```
// The meat of the list.
// When you put a new object into the list
// it is passed to the node which figures out
// where it goes and inserts it into the list
Node* InternalNode::insert(Data* otherData)
{
      // is the new guy bigger or smaller than me?
      int result = data->compare(*otherData);

      switch(result)
      {
                // by convention if it is the same as me it comes first
                case kIsSame: // fall through
                case kIsLarger: // new data comes before me
                {
                          InternalNode* dataNode =
                          new InternalNode(otherData, this);
                          return dataNode;
                }

                // it is bigger than I am so pass it on to the next
                // node and let IT handle it.
                case kIsSmaller:
                next = next->insert(otherData);
                return this;
      }
      return this; // appease the compiler
}
```

...

```
// Tail node is just a sentinel
class TailNode : public Node
{
     public:
               TailNode() {}
               virtual ~TailNode() {}
               virtual Node* insert(Data* data);
               virtual void show() {}
     private:
};

// If data comes to me, it must be inserted before me
// as I am the tail and NOTHING comes after me
Node* TailNode::insert(Data* data)
{
     InternalNode* dataNode = new InternalNode(data, this);
     return dataNode;
}

// Head node has no data, it just points
// to the very beginning of the list
class HeadNode : public Node
{
     public:
               HeadNode();
               virtual ~HeadNode() { delete next; }
               virtual Node* insert(Data* data);
               virtual void show() { next->show(); }
     private:
               Node* next;
};

```

...

```
// As soon as the head is created
// it creates the tail
HeadNode::HeadNode()
{
      next = new TailNode;
}

// Nothing comes before the head so just
// pass the data on to the next node
Node* HeadNode::insert(Data* data)
{
      next = next->insert(data);
      return this;
}

// I get all the credit and do none of the work
class LinkedList
{
      public:
                 LinkedList();
                 ~LinkedList() { delete head; }
                 void insert(Data* data);
                 void showAll() { head->show(); }
      private:
                 HeadNode* head;
};

// At birth, i create the head node
// It creates the tail node
// So an empty list points to the head which
// points to the tail and has nothing between
LinkedList::LinkedList()
{
      head = new HeadNode;
}
```

...

```
// Delegate, delegate, delegate
void LinkedList::insert(Data* pData)
{
     head->insert(pData);
}

// test driver program
int main()
{
     Data* pData;
     int val;
     LinkedList ll;

     // ask the user to produce some values
// put them in the list
while (true)
{
std::cout << “What value (0 to stop)? “;
std::cin >> val;
if (!val)
break;
pData = new Data(val);
ll.insert(pData);
}

// now walk the list and show the data
ll.showAll();
return 0; // ll falls out of scope and is destroyed!
}
```

Tugas

- Berikan comment pada program String.cpp yang berisi code program, kemudian jalankan kembali program Employee untuk menunjukkan seberapa sering fungsi ini dipanggil!
- Modifikasi program ArrayFunction.cpp untuk menolak input yang tidak sesuai!
- Tuliskan kembali program LinkedList.cpp untuk memegang object Tricycle lebih besar dari integer
- Tambahkan fungsi pada class LinkedList agar dapat menampilkan hitungan bilangan dari isi dari tiap node list.