Special Classes, Functions, dan Pointers

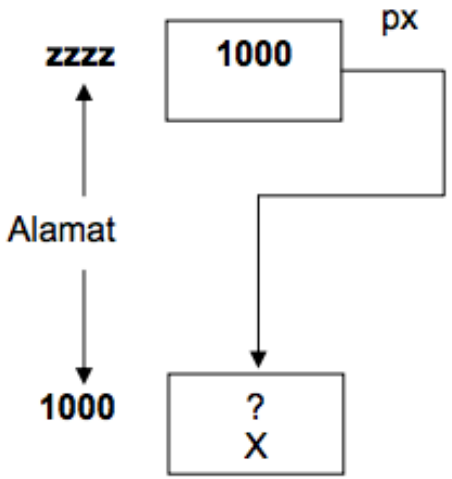
Pertemuan 12

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

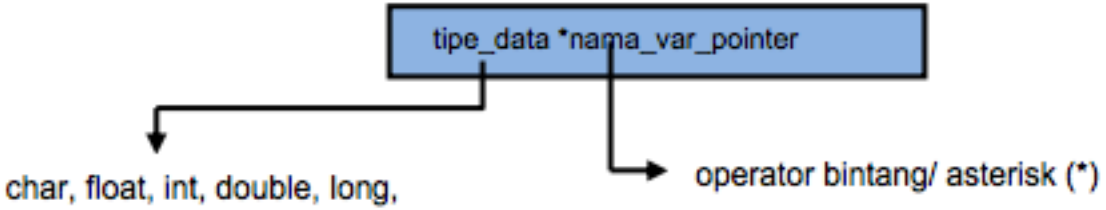
Dasar Teori

- 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 :



- 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

```
1: #include <iostream>
2:
3: class Cat
4: {
5:         public:
6:                   Cat(int newAge = 1):age(newAge){ howManyCats++; }
7:                   virtual ~Cat() { howManyCats—; }
8:                   virtual int getAge() { return age; }
9:                   virtual void setAge(int newAge) { age = newAge; }
10:                  static int howManyCats;
11:
12:        private:
13:                  int age;
14: };
15:
16: int Cat::howManyCats = 0;
17:
```

...

```
18: int main()
19: {
20:       const int maxCats = 5;
21:       Cat *catHouse[maxCats];
22:       int i;
23:       for (i = 0; i < maxCats; i++)
24:                 catHouse[i] = new Cat(i);
25:
26:       for (i = 0; i < maxCats; i++)
27:       {
28:                 std::cout << "There are ";
29:                 std::cout << Cat::howManyCats;
30:                 std::cout << " cats left!\n";
31:                 std::cout << "Deleting the one which is ";
32:                 std::cout << catHouse[i]->getAge();
33:                 std::cout << " years old\n";
34:                 delete catHouse[i];
35:                 catHouse[i] = 0;
36:       }
37:       return 0;
38: }
```

2. Static Member Functions

StaticFunction.cpp

```
1: #include <iostream>
2:
3: class Cat
4: {
5:         public:
6:                   Cat(int newAge = 1):age(newAge){ howManyCats++; }
7:                   virtual ~Cat() { howManyCats—; }
8:                   virtual int gGetAge() { return age; }
9:                   virtual void setAge(int newAge) { age = newAge; }
10:                  static int getHowMany() { return howManyCats; }
11:        private:
12:                  int age;
13:                  static int howManyCats;
14: };
15:
16: int Cat::howManyCats = 0;
17:
18: void countCats();
19:
```

...

```
20: int main()
21: {
22:       const int maxCats = 5;
23:       Cat *catHouse[maxCats];
24:       int i;
25:       for (i = 0; i < maxCats; i++)
26:       {
27:                 catHouse[i] = new Cat(i);
28:                 countCats();
29:       }
30:
31:       for (i = 0; i < maxCats; i++)
32:       {
33:                 delete catHouse[i];
34:                 countCats();
35:       }
36:       return 0;
37: }
38:
39: void countCats()
40: {
41:       std::cout << “There are “ << Cat::getHowMany()
42:       << “ cats alive!\n”;
43: }
```

3. Containment of Classes

String.hpp

```
1: #include <iostream>
2: #include <string.h>
3:
4: class String
5: {
6:         public:
7:         // constructors
8:         String();
9:         String(const char *const);
10:        String(const String&);
11:        ~String();
12:
13:        // overloaded operators
14:        char& operator[](int offset);
15:        char operator[](int offset) const;
16:        String operator+(const String&);
17:        void operator+=(const String&);
18:        String& operator= (const String &);
19:
20:        // general accessors
21:        int getLen() const { return len; }
22:        const char* getString() const { return value; }
23:        // static int constructorCount;
24:
25:        private:
26:        String(int); // private constructor
27:        char* value;
28:        int len;
29: };
30:
```

...

```
31: // default constructor creates string of 0 bytes
32: String::String()
33: {
34:       value = new char[1];
35:       value[0] = '\0';
36:       len = 0;
37:       // std::cout << "\tDefault string constructor\n";
38:       // constructorCount++;
39: }
40:
41: // private (helper) constructor, used only by
42: // class functions for creating a new string of
43: // required size. Null filled.
44: String::String(int len)
45: {
46:       value = new char[len + 1];
47:       int i;
48:       for (i = 0; i < len; i++)
49:       value[i] = '\0';
50:       len = len;
51:       // std::cout << "\tString(int) constructor\n";
52:       // constructorCount++;
53: }
54:
```

...

```
55: String::String(const char* const cString)
56: {
57:         len = strlen(cString);
58:         value = new char[len + 1];
59:         int i;
60:         for (i = 0; i < len; i++)
61:         value[i] = cString[i];
62:         value[len] = '\0';
63:         // std::cout << "\tString(char*) constructor\n";
64:         // constructorCount++;
65: }
66:
67: String::String(const String& rhs)
68: {
69:         len = rhs.getLen();
70:         value = new char[len + 1];
71:         int i;
72:         for (i = 0; i < len; i++)
73:         value[i] = rhs[i];
74:         value[len] = '\0';
75:         // std::cout << "\tString(String&) constructor\n";
76:         // constructorCount++;
77: }
78:
```

...

```
79: String::~String()
80: {
81:       delete [] value;
82:       len = 0;
83:       // std::cout << "\tString destructor\n";
84: }
85:
86: // operator equals, frees existing memory
87: // then copies string and size
88: String& String::operator=(const String &rhs)
89: {
90:       if (this == &rhs)
91:                 return *this;
92:       delete [] value;
93:       len = rhs.getLen();
94:       value = new char[len + 1];
95:       int i;
96:       for (i = 0; i < len; i++)
97:                 value[i] = rhs[i];
98:       value[len] = '\0';
99:       return *this;
100:      // std::cout << "\tString operator=\n";
101: }
102:
```

...

```
103: //non constant offset operator, returns
104: // reference to character so it can be
105: // changed!
106: char& String::operator[](int offset)
107: {
108:      if (offset > len)
109:      return value[len - 1];
110:      else
111:      return value[offset];
112: }
113:
114: // constant offset operator for use
115: // on const objects (see copy constructor!)
116: char String::operator[](int offset) const
117: {
118:      if (offset > len)
119:                return value[len-1];
120:      else
121:                return value[offset];
122: }
123:
```

...

```
124: // creates a new string by adding current
125: // string to rhs
126: String String::operator+(const String& rhs)
127: {
128:      int totalLen = len + rhs.getLen();
129:      int i, j;
130:      String temp(totalLen);
131:      for (i = 0; i < len; i++)
132:                temp[i] = value[i];
133:      for (j = 0; j < rhs.getLen(); j++, i++)
134:                temp[i] = rhs[j];
135:      temp[totalLen] = '\0';
136:      return temp;
137: }
138:
139: // changes current string, returns nothing
140: void String::operator+=(const String& rhs)
141: {
142:      int rhsLen = rhs.getLen();
143:      int totalLen = len + rhsLen;
144:      int i, j;
145:      String temp(totalLen);
146:      for (i = 0; i < len; i++)
147:      temp[i] = value[i];
148:      for (j = 0; j < rhs.getLen(); j++, i++)
149:      temp[i] = rhs[i - len];
150:      temp[totalLen] = '\0';
151:      *this = temp;
152: }
153:
154: // int String::constructorCount = 0;
```

Employee.cpp

```
1: #include "String.hpp"
2:
3: class Employee
4: {
5:        public:
6:                  Employee();
7:                  Employee(char *, char *, char *, long);
8:                  ~Employee();
9:                  Employee(const Employee&);
10:                 Employee& operator=(const Employee&);
11:
12:                 const String& getFirstName() const { return firstName; }
13:                 const String& getLastName() const { return lastName; }
14:                 const String& getAddress() const { return address; }
15:                 long getSalary() const { return salary; }
16:
17:                 void setFirstName(const String& fName)
18:                 { firstName = fName; }
19:                 void setLastName(const String& lName)
20:                 { lastName = lName; }
21:                 void setAddress(const String& newAddress)
22:                 { address = newAddress; }
23:                 void setSalary(long newSalary) { salary = newSalary; }
24:       private:
25:                 String firstName;
26:                 String lastName;
27:                 String address;
28:                 long salary;
29: };
30:
```

...

```
31: Employee::Employee():
32: firstName(“”),
33: lastName(“”),
34: address(“”),
35: salary(0)
36: {}
37:
38: Employee::Employee(char* newFirstName, char* newLastName,
39: char* newAddress, long newSalary):
40: firstName(newFirstName),
41: lastName(newLastName),
42: address(newAddress),
43: salary(newSalary)
44: {}
45:
46: Employee::Employee(const Employee& rhs):
47: firstName(rhs.getFirstName()),
48: lastName(rhs.getLastName()),
49: address(rhs.getAddress()),
50: salary(rhs.getSalary())
51: {}
52:
53: Employee::~Employee() {}
54:
55: Employee& Employee::operator=(const Employee& rhs)
56: {
57:        if (this == &rhs)
58:                   return *this;
59:
60:        firstName = rhs.getFirstName();
61:        lastName = rhs.getLastName();
62:        address = rhs.getAddress();
63:        salary = rhs.getSalary();
64:
65:        return *this;
66: }
67:
```

...

```
68: int main()
69: {
70:       Employee edie(“Jane”, “Doe”, “1461 Shore Parkway”, 20000);
71:       edie.setSalary(50000);
72:       String lastName(“Levine”);
73:       edie.setLastName(lastName);
74:       edie.setFirstName(“Edythe”);
75:
76:       std::cout << “Name: “;
77:       std::cout << edie.getFirstName().getString();
78:       std::cout << “ “ << edie.getLastName().getString();
79:       std::cout << “.\nAddress: “;
80:       std::cout << edie.getAddress().getString();
81:       std::cout << “.\nSalary: “ ;
82:       std::cout << edie.getSalary() << “\n”;
83:       return 0;
84: }
```

4. Pointers to Functions

FunctionPointer.cpp

```
1: #include <iostream>
2:
3: void square(int&, int&);
4: void cube(int&, int&);
5: void swap(int&, int&);
6: void getVals(int&, int&);
7: void printVals(int, int);
8:
9: int main()
10: {
11:       void (*pFunc)(int&, int&);
12:       bool fQuit = false;
13:
14:       int valOne = 1, valTwo = 2;
15:       int choice;
16:       while (fQuit == false)
17:       {
18:                 std::cout << "(0) Quit (1) Change Values "
19:                 << "(2) Square (3) Cube (4) Swap: ";
20:                 std::cin >> choice;
```

...

```
21:                 switch (choice)
22:                 {
23:                           case 1:
24:                           pFunc = getVals;
25:                           break;
26:                           case 2:
27:                           pFunc = square;
28:                           break;
29:                           case 3:
30:                           pFunc = cube;
31:                           break;
32:                           case 4:
33:                           pFunc = swap;
34:                           break;
35:                           default :
36:                           fQuit = true;
37:                           break;
38:                 }
39:
40:                 if (fQuit)
41:                           break;
42:
43:                 printVals(valOne, valTwo);
44:                 pFunc(valOne, valTwo);
45:                 printVals(valOne, valTwo);
46:       }
47:       return 0;
48: }
49:
```

...

```
50: void printVals(int x, int y)
51: {
52:       std::cout << "x: " << x << " y: " << y << "\n";
53: }
54:
55: void square(int &rX, int &rY)
56: {
57:       rX *= rX;
58:       rY *= rY;
59: }
60:
61: void cube(int &rX, int &rY)
62: {
63:       int tmp;
64:
65:       tmp = rX;
66:       rX *= rX;
67:       rX = rX * tmp;
68:
69:       tmp = rY;
70:       rY *= rY;
71:       rY = rY * tmp;
72: }
73:
```

...

```
74: void swap(int &rX, int &rY)
75: {
76:       int temp;
77:       temp = rX;
78:       rX = rY;
79:       rY = temp;
80: }
81:
82: void getVals(int &rValOne, int &rValTwo)
83: {
84:       std::cout << "New value for valOne: ";
85:       std::cin >> rValOne;
86:       std::cout << "New value for valTwo: ";
87:       std::cin >> rValTwo;
88: }
```

5. Arrays of Pointers to Functions

ArrayFunction.cpp

```
1: #include <iostream>
2:
3: void square(int&, int&);
4: void cube(int&, int&);
5: void swap(int&, int&);
6: void getVals(int&, int&);
7: void printVals(int, int);
8:
9: int main()
10: {
11:       int valOne=1, valTwo=2;
12:       int choice, i;
13:       const int maxArray = 5;
14:       void (*pFuncArray[maxArray])(int&, int&);
15:
16:       for (i=0;i < maxArray; i++)
17:       {
18:                 std::cout << "(1) Change Values "
19:                 << "(2) Square (3) Cube (4) Swap: ";
20:                 std::cin >> choice;
```

...

```
21:                 switch (choice)
22:                 {
23:                           case 1:
24:                                     pFuncArray[i] = getVals;
25:                                     break;
26:                           case 2:
27:                                     pFuncArray[i] = square;
28:                                     break;
29:                           case 3:
30:                                     pFuncArray[i] = cube;
31:                                     break;
32:                           case 4:
33:                                     pFuncArray[i] = swap;
34:                                     break;
35:                           default:
36:                                     pFuncArray[i] = 0;
37:                 }
38:       }
39:
40:       for (i = 0; i < maxArray; i++)
41:       {
42:                 pFuncArray[i](valOne, valTwo);
43:                 printVals(valOne, valTwo);
44:       }
45:       return 0;
46: }
47:
```

...

```
48: void printVals(int x, int y)
49: {
50:       std::cout << "x: " << x << " y: " << y << "\n";
51: }
52:
53: void square(int &rX, int &rY)
54: {
55:       rX *= rX;
56:       rY *= rY;
57: }
58:
59: void cube(int &rX, int &rY)
60: {
61:       int tmp;
62:
63:       tmp = rX;
64:       rX *= rX;
65:       rX = rX * tmp;
66:
67:       tmp = rY;
68:       rY *= rY;
69:       rY = rY * tmp;
70: }
71:
```

...

```
72: void swap(int &rX, int &rY)
73: {
74:       int temp;
75:       temp = rX;
76:       rX = rY;
77:       rY = temp;
78: }
79:
80: void getVals(int &rValOne, int &rValTwo)
81: {
82:       std::cout << "New value for valOne: ";
83:       std::cin >> rValOne;
84:       std::cout << "New value for valTwo: ";
85:       std::cin >> rValTwo;
86: }
```

6. Passing Pointers to Functions to Other Functions

FunctionPasser.cpp

```
1: #include <iostream>
2:
3: void square(int&,int&);
4: void cube(int&, int&);
5: void swap(int&, int&);
6: void getVals(int&, int&);
7: void printVals(void (*)(int&, int&),int&, int&);
8:
9: int main()
10:{
11:       int valOne=1, valTwo=2;
12:       int choice;
13:       bool fQuit = false;
14:
15:       void (*pFunc)(int&, int&);
16:
17:       while (fQuit == false)
18:       {
19:                 std::cout << "(0) Quit (1) Change Values "
20:                 << "(2) Square (3) Cube (4) Swap: ";
21:                 std::cin >> choice;
```

...

```
22:                 switch (choice)
23:                 {
24:                           case 1:
25:                                     pFunc = getVals;
26:                                     break;
27:                           case 2:
28:                                     pFunc = square;
29:                                     break;
30:                           case 3:
31:                                     pFunc = cube;
32:                                     break;
33:                           case 4:
34:                                     pFunc = swap;
35:                                     break;
36:                           default:
37:                                     fQuit = true;
38:                                     break;
39:                 }
40:                 if (fQuit == true)
41:                           break;
42:                 printVals(pFunc, valOne, valTwo);
43:       }
44:
45:       return 0;
46: }
47:
48: void printVals(void (*pFunc)(int&, int&),int& x, int& y)
49: {
50:       std::cout << "x: " << x << " y: " << y << "\n";
51:       pFunc(x, y);
52:       std::cout << "x: " << x << " y: " << y << "\n";
53: }
54:
```

...

```
55: void square(int &rX, int &rY)
56: {
57:         rX *= rX;
58:         rY *= rY;
59: }
60:
61: void cube(int &rX, int &rY)
62: {
63:         int tmp;
64:
65:         tmp = rX;
66:         rX *= rX;
67:         rX = rX * tmp;
68:
69:         tmp = rY;
70:         rY *= rY;
71:         rY = rY * tmp;
72: }
73:
74: void swap(int &rX, int &rY)
75: {
76:         int temp;
77:         temp = rX;
78:         rX = rY;
79:         rY = temp;
80: }
81:
82: void getVals(int &rValOne, int &rValTwo)
83: {
84:         std::cout << "New value for valOne: ";
85:         std::cin >> rValOne;
86:         std::cout << "New value for valTwo: ";
87:         std::cin >> rValTwo;
88: }
```

7. Pointers to Member Functions

MemberPointer.cpp

```
1: #include <iostream>
2:
3: enum BOOL {FALSE, TRUE};
4:
5: class Mammal
6: {
7:         public:
8:                   Mammal():age(1) { }
9:                   virtual ~Mammal() { }
10:                  virtual void speak() const = 0;
11:                  virtual void move() const = 0;
12:        protected:
13:                  int age;
14: };
15:
16: class Dog : public Mammal
17: {
18:        public:
19:                  void speak() const { std::cout << "Woof!\n"; }
20:                  void move() const { std::cout << "Walking to heel ...\n"; }
21: };
22:
```

...

```
23: class Cat : public Mammal
24: {
25:       public:
26:                 void speak() const { std::cout << "Meow!\n"; }
27:                 void move() const { std::cout << "Slinking...\n"; }
28: };
29:
30: class Horse : public Mammal
31: {
32:       public:
33:                 void speak() const { std::cout << "Winnie!\n"; }
34:                 void move() const { std::cout << "Galloping ...\n"; }
35: };
36:
37: int main()
38: {
39:       void (Mammal::*pFunc)() const = 0;
40:       Mammal* ptr = 0;
41:       int animal;
42:       int method;
43:       bool fQuit = false;
44:
45:       while (fQuit == false)
46:       {
47:                 std::cout << "(0) Quit (1) Dog (2) Cat (3) Horse: ";
48:                 std::cin >> animal;
```

...

```
49:                     switch (animal)
50:                     {
51:                               case 1:
52:                               ptr = new Dog;
53:                               break;
54:                               case 2:
55:                               ptr = new Cat;
56:                               break;
57:                               case 3:
58:                               ptr = new Horse;
59:                               break;
60:                               default:
61:                               fQuit = true;
62:                               break;
63:                     }
64:                     if (fQuit)
65:                               break;
66:
67:                     std::cout << "(1) Speak (2) Move: ";
68:                     std::cin >> method;
69:                     switch (method)
70:                     {
71:                               case 1:
72:                                         pFunc = &Mammal::speak;
73:                                         break;
74:                               default:
75:                                         pFunc = &Mammal::move;
76:                                         break;
77:                     }
78:
79:                     (ptr->*pFunc)();
80:                     delete ptr;
81:          }
82:          return 0;
83: }
```

8. Arrays of Pointers to Member Functions

MPFunction.cpp

```
1: #include <iostream>
2:
3: class Dog
4: {
5:         public:
6:                   void speak() const { std::cout << "Woof!\n"; }
7:                   void move() const { std::cout << "Walking to heel ...\n"; }
8:                   void eat() const { std::cout << "Gobbling food ...\n"; }
9:                   void growl() const { std::cout << "Grrrrr\n"; }
10:                  void whimper() const { std::cout << "Whining noises ...\n"; }
11:                  void rollOver() const { std::cout << "Rolling over ...\n"; }
12:                  void playDead() const
13:                  { std::cout << "Is this the end of Little Caesar?\n"; }
14: };
15:
16: typedef void (Dog::*PDF)() const;
17:
```

…

```
18: int main()
19: {
20:         const int maxFuncs = 7;
21:         PDF dogFunctions[maxFuncs] =
22:         {           &Dog::speak,
23:                     &Dog::move,
24:                     &Dog::eat,
25:                     &Dog::growl,
26:                     &Dog::whimper,
27:                     &Dog::rollOver,
28:                     &Dog::playDead
29:         };
30:
31:         Dog* pDog =0;
32:         int method;
33:         bool fQuit = false;
34:
35:         while (!fQuit)
36:         {
37:                     std::cout << "(0) Quit (1) Speak (2) Move (3) Eat (4) Growl";
38:                     std::cout << " (5) Whimper (6) Roll Over (7) Play Dead: ";
39:                     std::cin >> method;
40:                     if (method == 0)
41:                     {
42:                                 fQuit = true;
43:                                 break;
44:                     }
45:                     else
46:                     {
47:                                 pDog = new Dog;
48:                                 (pDog->*dogFunctions[method - 1])();
49:                                 delete pDog;
50:                     }
51:         }
52:         return 0;
53: }
```

9. LinkedList

LinkedList.cpp

```
1: // Demonstrates an object-oriented approach to
2: // linked lists. The list delegates to the node.
3: // The node is an abstract data type. Three types of
4: // nodes are used, head nodes, tail nodes and internal
5: // nodes. Only the internal nodes hold data.
6: //
7: // The Data class is created to serve as an object to
8: // hold in the linked list.
9: //
10: #include <iostream>
11:
12: enum { kIsSmaller, kIsLarger, kIsSame };
13:
14: // Data class to put into the linked list
15: // Any class in this linked list must support two
16: // functions: show (displays the value) and compare
17: // (returns relative position)
18: class Data
19: {
20:       public:
21:                 Data(int newVal):value(newVal) {}
22:                 ~Data() {}
23:                 int compare(const Data&);
24:                 void show() { std::cout << value << "\n"; }
25:       private:
26:                 int value;
27: };
28:
```

...

```
29: // Compare is used to decide where in the list
30: // a particular object belongs.
31: int Data::compare(const Data& otherData)
32: {
33:       if (value < otherData.value)
34:                 return kIsSmaller;
35:       if (value > otherData.value)
36:                 return kIsLarger;
37:       else
38:                 return kIsSame;
39: }
40:
41: // forward declarations
42: class Node;
43: class HeadNode;
44: class TailNode;
45: class InternalNode;
46:
47: // ADT representing the node object in the list.
48: // Every derived class must override insert and show.
49: class Node
50: {
51:       public:
52:                 Node() {}
53:                 virtual ~Node() {}
54:                 virtual Node* insert(Data* data) = 0;
55:                 virtual void show() = 0;
56:       private:
57: };
58:
```

…

```
59: // This is the node that holds the actual object.
60: // In this case the object is of type Data.
61: // We'll see how to make this more general when
62: // we cover templates.
63: class InternalNode : public Node
64: {
65:       public:
66:                 InternalNode(Data* data, Node* next);
67:                 virtual ~InternalNode() { delete next; delete data; }
68:                 virtual Node* insert(Data* data);
69:                 virtual void show()
70:                 { data->show(); next->show(); } // delegate!
71:
72:       private:
73:                 Data* data; // the data itself
74:                 Node* next; // points to next node in the linked list
75: };
76:
77: // All the constructor does is to initialize
78: InternalNode::InternalNode(Data* newData, Node* newNext):
79: data(newData), next(newNext)
80: {
81: }
82:
```

...

```
83: // The meat of the list.
84: // When you put a new object into the list
85: // it is passed to the node which figures out
86: // where it goes and inserts it into the list
87: Node* InternalNode::insert(Data* otherData)
88: {
89:       // is the new guy bigger or smaller than me?
90:       int result = data->compare(*otherData);
91:
92:       switch(result)
93:       {
94:                 // by convention if it is the same as me it comes first
95:                 case kIsSame: // fall through
96:                 case kIsLarger: // new data comes before me
97:                 {
98:                           InternalNode* dataNode =
99:                           new InternalNode(otherData, this);
100:                          return dataNode;
101:                }
102:
103:                // it is bigger than I am so pass it on to the next
104:                // node and let IT handle it.
105:                case kIsSmaller:
106:                next = next->insert(otherData);
107:                return this;
108:      }
109:      return this; // appease the compiler
110: }
111:
```

...

```
112: // Tail node is just a sentinel
113: class TailNode : public Node
114: {
115:      public:
116:                TailNode() {}
117:                virtual ~TailNode() {}
118:                virtual Node* insert(Data* data);
119:                virtual void show() {}
120:      private:
121: };
122:
123: // If data comes to me, it must be inserted before me
124: // as I am the tail and NOTHING comes after me
125: Node* TailNode::insert(Data* data)
126: {
127:      InternalNode* dataNode = new InternalNode(data, this);
128:      return dataNode;
129: }
130:
131: // Head node has no data, it just points
132: // to the very beginning of the list
133: class HeadNode : public Node
134: {
135:      public:
136:                HeadNode();
137:                virtual ~HeadNode() { delete next; }
138:                virtual Node* insert(Data* data);
139:                virtual void show() { next->show(); }
140:      private:
141:                Node* next;
142: };
143:
```

...

```
144: // As soon as the head is created
145: // it creates the tail
146: HeadNode::HeadNode()
147: {
148:       next = new TailNode;
149: }
150:
151: // Nothing comes before the head so just
152: // pass the data on to the next node
153: Node* HeadNode::insert(Data* data)
154: {
155:       next = next->insert(data);
156:       return this;
157: }
158:
159: // I get all the credit and do none of the work
160: class LinkedList
161: {
162:       public:
163:                  LinkedList();
164:                  ~LinkedList() { delete head; }
165:                  void insert(Data* data);
166:                  void showAll() { head->show(); }
167:       private:
168:                  HeadNode* head;
169: };
170:
171: // At birth, i create the head node
172: // It creates the tail node
173: // So an empty list points to the head which
174: // points to the tail and has nothing between
175: LinkedList::LinkedList()
176: {
177:       head = new HeadNode;
178: }
179:
```

…

```
180: // Delegate, delegate, delegate
181: void LinkedList::insert(Data* pData)
182: {
183:      head->insert(pData);
184: }
185:
186: // test driver program
187: int main()
188: {
189:      Data* pData;
190:      int val;
191:      LinkedList ll;
192:
193:      // ask the user to produce some values
194: // put them in the list
195: while (true)
196: {
197: std::cout << "What value (0 to stop)? ";
198: std::cin >> val;
199: if (!val)
200: break;
201: pData = new Data(val);
202: ll.insert(pData);
203: }
204:
205: // now walk the list and show the data
206: ll.showAll();
207: return 0; // ll falls out of scope and is destroyed!
208: }
```

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.