FUN WITH CLASSES

- Just as in Java! In particular,
  - A class holds member variables and member functions (hereinafter called “members” when referred to as a whole).
  - A class member can be in the private, protected, or public section.
  - There are a number of constructors (same name as the class, no return type).
    - ∗ If no constructor is defined, then an implicit one is inserted by default. The default constructor initializes the variable members with default values.
    - ∗ But if one (or more) constructors are provided, then the default constructor is no longer available.
- Things that are different from Java: we also have a destructor.
  - Its name is the name of the class, prefixed by ~.
  - It is called by the system once the extent of an object lapses, and its job is to clean up after the object.
  - A default destructor (which does nothing) is provided.

EXAMPLE: LISTS, A FIRST IMPLEMMENTATION

- Class declaration (e.g., in `list.h`)

```
#ifndef __LIST_H
#define __LIST_H

#include <iostream>
using namespace std;

struct cons_cell {
  int car;
  cons_cell* cdr;
  cons_cell(int , cons_cell* = 0);
};
```

```
class list {
  cons_cell* content;

 public:

  list(void);
  list(cons_cell*);
  list(int, cons_cell* = 0);
  ~list(void);

  int null(void) const;
  int car(void) const;
  void cdr(void);
  void cons(int);
  void rmth(int = 0);
  void print(void) const;
};

#endif /* __LIST_H */
```

MUTATORS AND ACCESSORS

- A member function that changes the state of an object (e.g., the variables therein) is a mutator.
- By contrast, a member functions which does not change the state of the object (e.g., it just returns the value of some variable) is an accessor.
- In C++, we can mark each function as accessor or mutator:
  - By default, member functions are mutators.
  - To make a function accessor, we add `const` after the closing parenthesis that ends the parameter list.
  - This is not just a comment; it has semantic implications.
    - Indeed, mutators cannot be applied to constant objects, and a good C++ compiler does enforce this.

LISTS, THE IMPLEMENTATION

```
#include "list.h"
cons_cell::cons_cell (int val, cons_cell* rest) {
  car = val;
  cdr = rest;
}
list::list (void) {
  content = 0;
}
list::list (cons_cell* c) {
  content = c;
}
list::list (int val, cons_cell* rest) {
  content = new cons_cell(val,rest);
}
```

- When implementing member functions, you have to say which class the member function pertains to.
  - You do this by using the scope operator `::`.
  - when you write *class-name*::*member* you refer to the entity *member* of class *class-name*.
  - Do not confuse `::` (refers to a class) with `.` (refers to an object).

- Alternatively, you can define a constructor by using an initializer list:

```
list::list (cons_cell* c)
  : content (c) {
}
```

- The main role of the destructor is to deallocate memory that was allocated dynamically.
  - You also do here whatever you need to do when your object is destroyed.

```
list::~list (void) {
  while (content != 0)
    cdr();
}
```

```
void list::cdr (void) {
  if (content != 0) {
    cons_cell* tmp = content;
    content = content -> cdr;
    delete tmp;
  }
}
```

Lists, the implementation (cont'd)

```
int list::null (void) const {
  return content == 0;
}

int list::car (void) const {
  return content -> car;
}

void list::cons(int c) {
  content = new cons_cell(c,content);
}

void list::print(void) const {
  cons_cell* iter = content;
  cout << "(";
  while (iter != 0) {
    cout << iter -> car;
    iter = iter -> cdr;
    if (iter != 0) cout << ",";
  }
  cout << ")";
}
```

```
void list::rmth (int which) {
  cons_cell* place = content;
  // go to element which - 1...
  for (int i = 0; i < which - 1; i++) {
    if (place == 0)
      return;  // nothing to delete,
               // we are done.
    place = place -> cdr;
  }
  if (place !=0 && place -> cdr != 0) {
    cons_cell* to_delete = place -> cdr;
    place -> cdr = place -> cdr -> cdr;
    delete to_delete;
  }
}
```

- Not as in Java! Do not use `new` when creating a normal object (i.e., as a local or global variable):

Correct	Wrong

`list example1;`
`cons_cell* c = 0;`
`list example2(c);`
`list example3(1);`
`list example4(1,0);`
`// what do they mean?`
`list example5 = 0;`
`list example6 = c;`
	`list example1 = new list;`
	`  // correct in Java, wrong in C++!!`
	`list example2();`
	`  // why?`

- However, do use `new` when you allocate memory for your object dynamically (i.e., when you initialize a pointer):

```
list* pointer_example = new list;
```

OBJECTS, OBJECTS EVERYWHERE

- In Java, you have primitive data types (such as `int`, `float`, etc.) and classes.
- In C++, any data type is a class, including `int`, `float`, etc.
  - In other words, either of the two declarations below is correct.

```
int counter(52);          int counter = 52;
```

- Remember the initializer list?

```
list::list (cons_cell* c)
  : content (c) { }
```

  - Compare with the default constructor, which is:

```
list::list () { }
```

    * The object `content` (a pointer!) is then initialized using its default constructor, which for a pointer just initializes it with 0.
  - If we do not like the default constructor, we can ask for another one in the initializer list.
    * In this particular case, we ask for the unary constructor of a pointer, which initializes the pointer with the argument.

THE BIG THREE

- Besides the default void constructor, three more member functions are defined for you by default: a copy constructor, an $=$ operator, and a destructor.
- Copy constructor. Fires up when you write

```
int i = 0;
int j = i;  // remember, this is equivalent to int j(i);
```

  - For this to work, there has to be a constructor `int::int(int)`.
    * Well, such a constructor exists. For various purposes (which?) though, it is `int::int(const int&)` and is called the copy constructor.
    * In general, a default copy constructor `c::c(const c&)` is automatically created for any class `c` in the system. It just copies all the member variables using the respective copy constructors.

- The $=$ operator. The default such an operator does exactly what the copy constructor does.

```
list l2 = l1;  // copy constructor
list l2(l1);   // copy constructor too
l2 = l1;       // the operator =, NOT the copy constructor
```

When defaults do not work

```
cout << "lst = ";
lst.print(); cout << "\n";

cout << "We do list clone(lst);\n";
cout << "      lst.cdr(); lst.cdr();\n";
list clone(lst);
lst.cdr(); lst.cdr();
cout << "lst = ";
lst.print(); cout << "\n";
cout << "clone = ";
clone.print(); cout << "\n";

cout << "We do clone1 = lst;\n";
cout << "      lst.cdr(); lst.cdr();\n";
list clone1;
clone1 = lst;
lst.cdr(); lst.cdr();
cout << "lst = ";
lst.print(); cout << "\n";
cout << "clone1 = ";
clone1.print(); cout << "\n";
```

What we want:

```
lst = (7,5,4,3,2,1)
We do list clone(lst);
      lst.cdr(); lst.cdr();
lst = (4,3,2,1)
clone = (7,5,4,3,2,1)
We do clone1 = lst;
      lst.cdr(); lst.cdr();
lst = (2,1)
clone1 = (4,3,2,1)
```

What we actually get:

```
lst = (7,5,4,3,2,1)
We do list clone(lst);
      lst.cdr(); lst.cdr();
lst = (4,3,2,1)
clone = (7,5,4,3,2,1)
We do clone1 = lst;
      lst.cdr(); lst.cdr();
lst = (2,1)
clone1 = (4,3,2,1)
Segmentation fault
```

When defaults do not work (cont'd)

- Out class (`list`) contains a pointer. The default copy constructor and = operator just copies the pointer.
  - In effect, the defaults do shallow copying; we want deep copying.
  - Solution: roll your own member functions.

```
class list {
  ...
  cons_cell* clone_cons (cons_cell*) const;
 public:
  list(const list&);
  const list& operator=(const list&);
  ...
}
```

    * Note that the = operator returns `list&` (why?).

  - When you write: `clone1 = lst;`
    you actually mean: `clone1.operator=(lst);`

IMPLEMENTATION OF COPY CONSTRUCTOR AND = OPERATOR

```
/*
 * Does the deep copying of content.  We cannot easily do it with a
 * cycle, since a naive such a cycle will copy the list in the wrong
 * order.  So we write a recursive function.
 */
cons_cell* list::clone_cons (cons_cell* c) const {
  if (c == 0) return 0;
  return (new cons_cell(c -> car, clone_cons(c -> cdr)));
}

list::list(const list& l) {
  content = clone_cons(l.content);
}

const list& list::operator=(const list& rhs) {
  if (this != &rhs)
    content = clone_cons(rhs.content);
  return *this;  // because we may need to do a = b = c;
}
```

IMPLEMENTATION OF COPY CONSTRUCTOR AND = OPERATOR

```
/*
 * Does the deep copying of content.  We cannot easily do it with a
 * cycle, since a naive such a cycle will copy the list in the wrong
 * order.  So we write a recursive function.
 */
cons_cell* list::clone_cons (cons_cell* c) const {
  if (c == 0) return 0;
  return (new cons_cell(c -> car, clone_cons(c -> cdr)));
}

list::list(const list& l) {
  content = clone_cons(l.content);
}

const list& list::operator=(const list& rhs) {
  if (this != &rhs) // Standard alias test (when we do a = a;)
    content = clone_cons(rhs.content);
  return *this;  // because we may need to do a = b = c;
}
```

THE DESTRUCTOR

- The destructor of an object is called immediately before that object ceases to exist. In particular,
  - The destructor of a local variable is called immediately before the block that defines it returns.
  - The destructor of a global variable or of a local static variable is called at the very end of the program.
  - The destructor of a variable member of class c is automatically called by the destructor of c.

THE DESTRUCTOR (CONT'D)

```
int main () {
  int elm = -1;
  cout<<"Birth: \"lst\": ";
  list lst;
  while (elm != 0) {
    cin >> elm; if (elm != 0) lst.cons(elm); }
  lst.rmth(1); lst.rmth(10);
  cout<<"lst = "; lst.print(); cout<<"\n";
  cout<<"Birth: \"plist\", dyn.: ";
  list* plist = new list;
  cout<<"We do list clone(lst);\n";
  cout<<"      lst.cdr(); lst.cdr();\n";
  cout<<"Birth: \"clone\": ";
  list clone(lst); lst.cdr(); lst.cdr();
  cout<<"lst = "; lst.print(); cout<<"\n";
  cout<<"clone = "; clone.print(); cout<<"\n";
  cout<<"We do delete plist\n"; delete(plist);
  cout<<"We do clone1 = lst;\n";
  cout<<"      lst.cdr(); lst.cdr();\n";
  cout<<"Birth: \"clone1\": ";
  list clone1; clone1 = lst; lst.cdr(); lst.cdr();
  cout<<"lst = "; lst.print(); cout<<"\n";
  cout<<"clone1 = "; clone1.print(); cout<<"\n";
}
```

```
Birth: "lst": 0x7ffff7e8
1
2
3
4
0
lst = (4,2,1)
Birth: "plist", dyn.:
                0x10011be0
We do list clone(lst);
      lst.cdr(); lst.cdr();
Birth: "clone": 0x7ffff808
lst = (1)
clone = (4,2,1)
We now call delete plist
### Death: 0x10011be0
We do clone1 = lst;
      lst.cdr(); lst.cdr();
Birth: "clone1": 0x7ffff818
lst = ()
clone1 = (1)
### Death: 0x7ffff818
### Death: 0x7ffff808
### Death: 0x7ffff7e8
```

The rule of the Big Three

- Whenever the defaults work for everything you do not need to define anything.
- . . . however, when the default does not work for one of the big three, then the defaults won't wotk for the others
- When it comes to the Big Three,
  - You either do not need to define any, or you need to define all!
  - All the Big Three must make the same assumption about data (whether it is deep copied or shallow copied, etc.)

SIMPLE INHERITANCE

```
#include "list.h"

class ilist: list {
 public:
  ilist(void);
  ilist(const ilist&);
  ilist(const list&);
  int operator[](int) const;
};
ilist::ilist(void)
  : list () { /* empty */ }

ilist::ilist(const ilist& l)
 : list(l) { /* empty */ }

ilist::ilist(const list& l)
 : list(l) { /* empty */ }

int ilist::operator[](int i) const {
  cons_cell* place = content;
  for (int i = 0; i < i; i++)
    place = place -> cdr;
  return place -> car;
  }
```

```
class list {
  cons_cell* content;
  cons_cell* clone_cons (cons_cell*) const;

 public:

  list(void);
  list(const list&);
  list(cons_cell*);
  list(int, cons_cell* = 0);
  ~list(void);
  const list& operator=(const list&);

  int null(void) const;
  int car(void) const;
  void cdr(void);
  void cons(int);
  void rmth(int = 0);
  void print(void) const;
};
```

- Visibility rules: With B an object of the base class, D an object of the derived class, and M a member of the base class,

Public inheritance situation	Public	Protected	Private
Base class member function accessing M	good	good	good
Derived class member function accessing M	good	good	error
`main` accessing $B.M$ or $D.M$	good	error	error
Derived class member function accessing $B.M$	good	error	error

- The default constructor for a derived class is

```
Derived() : Base () { }
```

- The copy constructor and the operator $=$ behave in the same manner:
  - they call their correspondent in the base class and then copy whatever remains using the usual assignment operator.

SIMPLE INHERITANCE (CONT'D)

```
#include "list.h"

class ilist: list {
 public:
  ilist(const list&);
  int operator[](int) const;
};

ilist::ilist(const list& l)
 : list(l) {
  /* empty */
}

int ilist::operator[](int i) const {
  cons_cell* place = content;
  for (int i = 0; i < i; i++)
    place = place -> cdr;
  return place -> car;
}
```

```
class list {
  cons_cell* clone_cons (cons_cell*) const;

 protected:
  cons_cell* content;

 public:
  list(void);
  list(const list&);
  list(cons_cell*);
  list(int, cons_cell* = 0);
  ~list(void);
  const list& operator=(const list&);

  int null(void) const;
  int car(void) const;
  void cdr(void);
  void cons(int);
  void rmth(int = 0);
  void print(void) const;
};
```

Using derived classes

```
cout << "Using copy constructor from list to ilist.\n";
ilist il(lst);
cout << "indexed lists: il[3] = " << il[3] << "\n";
cout << "Using assignment operator from list to ilist.\n";
ilist il1;
il1 = lst;
cout << "indexed lists: il1[3] = " << il1[3] << "\n";
```

Using derived classes

```
cout << "Using copy constructor from list to ilist.\n";
ilist il(lst);
cout << "indexed lists: il[3] = " << il[3] << "\n";
cout << "Using assignment operator from list to ilist.\n";
ilist il1;
il1 = lst;
cout << "indexed lists: il1[3] = " << il1[3] << "\n";
```

```
main.cc: In function ‘int main()’:
main.cc:23: no matching function for call to ‘ilist::ilist ()’
ilist.h:8: candidates are: ilist::ilist(const list &)
ilist.h:10:                ilist::ilist(const ilist &)
make: *** [main.o] Error 1
```

Indexed lists again

```
#include "list.h"

class ilist: list {
 public:
  ilist(void);
  ilist(const list&);
  int operator[](int) const;
};

ilist::ilist(const list& l)
 : list(l) {
  /* empty */
}
```

```
ilist::ilist(void)
  : list() {
  /* empty */
}
```

```
int ilist::operator[](int ix) const {
  cons_cell* place = content;
  for (int i = 0; i < ix; i++)
    place = place -> cdr;
  return place -> car;
}
```

Assigning to indices

- We would also like to do this:

```
il.print(); // il = (7,5,4,3,2,1)
il[3] = 7;
il.print(); // il = (7,5,4,7,2,1)
```

ASSIGNING TO INDICES

- We would also like to do this:

```
il.print(); // il = (7,5,4,3,2,1)
il[3] = 7;
il.print(); // il = (7,5,4,7,2,1)
```

- We then change the [] operator so that it returns a reference:

```
// in class declaration:
int& operator[](int) const;

// in class implementation:
int& ilist::operator[](int ix) const {
  cons_cell* place = content;
  for (int i = 0; i < ix; i++)
    place = place -> cdr;
  return place -> car;
}
```

ASSIGNING TO INDICES

- We would also like to do this:

```
il.print(); // il = (7,5,4,3,2,1)
il[3] = 7;
il.print(); // il = (7,5,4,7,2,1)
```

- We then change the [] operator so that it returns a reference:

```
// in class declaration:
int& operator[](int) const;

// in class implementation:
int& ilist::operator[](int ix) const {
  cons_cell* place = content;
  for (int i = 0; i < ix; i++)
    place = place -> cdr;
  return place -> car;
}
```

- . . . and we get:

```
main.cc:27: fields of ‘const list’ are inaccessible in ‘ilist’ due to
            private inheritance
```

PUBLIC INHERITANCE!!

```
class ilist: public list {
 public:
  ilist(void);
  ilist(const list&);
  int& operator[](int) const;
};

ilist::ilist(void)
  : list() {
  /* empty */
}

ilist::ilist(const list& l)
 : list(l) {
  /* empty */
}

int& ilist::operator[](int which) const {
  cons_cell* place = content;
  for (int i = 0; i < which; i++)
    place = place -> cdr;
  return place -> car;
}
```

stack.h

```
#ifndef __ISTACK_H
#define __ISTACK_H
#include "list.h"

class stack: private list {
 public:
  void push(int);
  void pop(void);
  int top(void) const;
  int null(void) const;
};
#endif /* __ISTACK_H */
```

stack.cc

```
#include "stack.h"

void stack::push(int i) {
  cons(i);
}
int stack::top(void) const {
  return car();
}
int stack::null(void) const {
  return list::null();
}
void stack::pop(void) {
  cdr(); }
```

mains.cc

```
#include "stack.h"
int main () {
  int elm = -1; stack s;
  while (elm != 0) { cin >> elm; if (elm != 0)  s.push(elm); }
  // s.cdr();   --> ‘void list::cdr()’ is inaccessible within this context
  // s.print(); --> error too!
  cout << s.top() << "\n";                        }
```

Private inheritance (cont'd)

- Visibility rules: With B an object of the base class, D an object of the derived class, and M a member of the base class,

Private inheritance situation	Public	Protected	Private
Base class member function accessing M	good	good	good
Derived class member function accessing M	good	error	error
`main` accessing $B.M$	good	error	error
`main` accessing $D.M$	error	error	error
Derived class member function accessing $B.M$	error	error	error

- In general, you should avoid private inheritance...

```
class d : private b {

  ... (access b::m) ...
}
```

largely equivalent with:

```
class d {
  b o;
  ... (access o.m) ...
}
```

- ...unless it greatly simplifies the code, or simplifies coding logic, or is justified on performance grounds.

Avoiding private inheritance

stack.h

```
#ifndef __ISTACK_H
#define __ISTACK_H
#include "list.h"

class stack {
   list stk;
 public:
  void push(int);
  void pop(void);
  int top(void) const;
  int null(void) const;
};
#endif /* __ISTACK_H */
```

stack.cc

```
#include "stack.h"

void stack::push(int i) {
  stk.cons(i);
}
int stack::top(void) const {
  return stk.car();
}
int stack::null(void) const {
  return stk.null();
}
void stack::pop(void) {
  stk.cdr(); }
```

Overriding a member function

```
class worker {
  ...
 public:
  void do_work(void);
  ...
};

class workaholic: public worker {
  ...
 public:
  void do_work(void);
  ...
};

void workaholic::do_work(void) {
                        // work like a worker
  have_coffee();        // take short break
                        // work like a worker some more
}
```

Overriding a member function

```
class worker {
  ...
 public:
  void do_work(void);
  ...
};

class workaholic: public worker {
  ...
 public:
  void do_work(void);
  ...
};

void workaholic::do_work(void) {
  worker::do_work();    // work like a worker
  have_coffee();        // take short break
  worker::do_work();    // work like a worker some more
}
```

- `cons_cell` is not used outside the classses `list` and `ilist`.
  - We would therefore like to disallow access to its members (all of them, including its constructor!!) for anybody else than the classes `list` and `ilist`.
  - We could declare it in the protected area of class list.
    - Nobody will then be able to access its members outside the class we define it in.
    - But then nobody will know about its existence either.
- We would also like to be able to print lists just by doing something like this:

```
list lst;
cout << lst << "\n";
```

FRIENDS

- First, we make `cons_cell` a class instead of a struct (i.e., all of its members are private by default).
- Given a class C, a friend class of C is allowed to access the private members of C just as C does.
  - So we declare class `list` to be a friend of our class `cons_cell`.
  - "Friendliness" is not inherited, so we must do the same thing with `ilist`.

```
class cons_cell {
  int car;
  cons_cell* cdr;
  cons_cell(int , cons_cell* = 0);

  friend class list;
  friend class ilist;
};
```

I/O FRIENDS

- Operators >> and << normally do shifts.
- However, they are also redefined to do I/O.
  - So we could also redefine them to do I/O for our class.
  - But we cannot define them as members of class `list` (why?).

I/O FRIENDS

- Operators >> and << normally do shifts.
- However, they are also redefined to do I/O.
  - So we could also redefine them to do I/O for our class.
  - But we cannot define them as members of class `list`:
    - If << were a member function of `list` it would take an object of type `list` and an object of type `ostream`. We would then write `lst << cout`.
    - What we want is the other way around, because we want to write `cout << lst`.
    - So we declare << as

```
... operator<< (ostream& out, const list& value);
```

  - Conclusion: we make I/O operators functions, and we declare them friends of our class:

I/O FRIENDS (CONT'D)

In list.h:

```
class list {
  ...
  friend ostream& operator<< (ostream& out, const list& value);
};
```

In list.cc:

```
ostream& operator<< (ostream& out, const list& value) {
  list iter(value);  out << "(";
  while (!iter.null()) {
    out << iter.car(); iter.cdr();
    if (!iter.null()) out << ",";
  }
  out << ")";
  return out;
}
```

Avoiding friendly functions

- In general, printing can be done using accessors, which are public anyway.
- If you need a friend function, you can always write an equivalent public member function and then just call that function from within the friend function.
  - Then the function does not need to be friend anymore:
    In list.h:

```
class list {
  ...
  /* no friends necessary */
};

ostream& operator<< (ostream& out, const list& value);
```

In list.cc:

```
ostream& operator<< (ostream& out, const list& value) {
  value.print();
  return out;
}
```

Operator overloading

- You can overload almost any operator you like.
  - However, you cannot create new operators (stick with overloading the existing ones).
    - This include changing the arity of some operator.
  - The following operators cannot be overloaded: `.`, `::`, `?:`, and `->`.
- Recommendations for operator overloading:
  - Use similar meaning: use overloaded operators to do operations as close as possible to those they already do.
  - Be consistent: if you overload one arithmetic operator, it is a good idea to overload all of them.
  - Do not abuse: sometimes an operator is easier to understand than a function (e.g., indexing using `[]`), sometimes it is not (e.g., getting the prefix of a string using `-`). When in doubt, use a function.

STATIC CLASS MEMBERS

- Exactly as in Java, a static class member is a global variable visible only to class members (if declared private).
  - There is one static member per class instead of one per instance.
  - You access a static member by using the scope operator (`::`), not the member access operator (`.`).

```
class list {
  static int active_instances;
  ...
};

int list::active_instances = 0;
```

```
list::list (void) {
  active_instances++;
  content = 0;
}
```

```
list::~list (void) {
  active_instances--;
  while (content != 0)
    cdr();
}
```