# CHAPTER 22

# FROM C TO C++

22.1 INTRODUCTION TO C++
- Data Abstraction And Encapsulation
- Inheritance
- Polymorphism

22.2 C++ PROCEDURAL EXTENSIONS TO C
- References
- Function and Operator Overloading
- Input and Output Operators
- Default Function Arguments
- Inline Functions
- Storage Allocation and Deallocation

22.3 OBJECT-ORIENTED PROGRAMMING
- Defining a Stack Class
- Defining the Stack Operations
- Using the Stack
- An Improved Stack Class

22.4 USING BUILT-IN CLASSES—FILE I/O

22.5 INHERITANCE

# swap.C

(Figure 22.1 – Page 556)

```
//
// C++ program using a swap function to exchange two int values.
//
#include <stdio.h>

int main()
{
  void swap(int &xptr, int &yptr);
  int s = 10, t = 20;

  printf("Before swap, s=%i, t=%i\n", s, t);
  swap(s, t);
  printf("After swap, s=%i, t=%i\n", s, t);
  return 0;
}

void swap(int &x, int &y)
{
  int temp;

  temp = x;  x = y;  y = temp;
}
```

# prstruct.C

(Figure 22.2 – Page 557)

```
//
// C++ program to display employee information.
//
#include <stdio.h>

struct employee
{
  long number;                            // employee number
  char *name, *phone;                     // name and phone
  int age;                                // employee age
};

int main()
{
  void print_emp(const employee &emp);
  employee emp;

  emp.number = 1001;                      // fill in employee info
  emp.age    =  23;
  emp.name   = "borromeo, daphne";
  emp.phone  = "310-555-2042";
  print_emp(emp);                         // write our employee
  return 0;
}

void print_emp(const employee &e)        // reference to employee
{
  printf("Employee: %li\n", e.number);
  printf("Name:     %s\n", e.name);
  printf("Age:      %i\n", e.age);
  printf("Phone:    %s\n", e.phone);
}
```

# prstrct2.C

(Figure 22.3 – Page 558)

```
//
// C++ program using overloaded functions to display employee info.
//
#include <stdio.h>

struct employee
{
  long number;                          // employee number
  char *name, *phone;                   // name and phone
  int age;                              // employee age
};

void display(const employee &e);        // display prototypes
void display(char *s);
void display(int i);
void display(long l);

int main()
{
  employee emp;

  emp.number = 1001;                    // fill in employee info
  emp.age    =  23;
  emp.name   = "borromeo, daphne";
  emp.phone  = "310-555-2042";
  display(emp);                         // write our employee
  return 0;
}
```

# prstrct2.C

(Figure 22.3 – Page 558)

```
void display(const employee &e)            // display an employee
{
  display("Employee: ");  display(e.number);  display("\n");
  display("Name:     ");  display(e.name);    display("\n");
  display("Age:      ");  display(e.age);     display("\n");
  display("Phone:    ");  display(e.phone);   display("\n");
}

void display(char *s)                       // display a string
{
  printf("%s", s);
}

void display(int i)                         // display an integer
{
  printf("%i", i);
}

void display(long l)                         // display a long
{
  printf("%li", l);
}
```

# power.C

(Figure 22.4 – Page 559)

```
//
// C++ Program using the math library to compute exponents.
//
#include <iostream.h>
#include <math.h>

int main()
{
  double x,y;                        // user-supplied base and exponent

  while (cin >> x && cin >> y)
    cout << x << '^' << y << " = " << pow(x,y) << '\n';
  return 0;
}
```

# prstrct3.C

(Figure 22.5 – Page 560)

```
//
// C++ program using an overloaded operator to display employee info.
//
#include <iostream.h>

struct employee
{
  long number;                            // employee number
  char *name, *phone;                     // name and phone
  int  age;                               // employee and age
};

ostream& operator<<(ostream& output, const employee &e);

int main()
{
  employee emp;

  emp.number = 1001;                      // fill in employee info
  emp.age    =  23;
  emp.name   = "borromeo, daphne";
  emp.phone  = "310-555-2042";
  cout << emp;                            // write our employee
  return 0;
}

ostream& operator<<(ostream& output, const employee &e)
{
  output << "Employee: " << e.number << "\n";
  output << "Name:     " << e.name   << "\n";
  output << "Age:      " << e.age    << "\n";
  output << "Phone:    " << e.phone  << "\n";
  return output;
}
```

# array.C
(Figure 22.6 – Page 561)

```
//
// C++ program to print an array, several elements per line.
//
#include <iostream.h>

int a[10] = {81, 78, 45, 78, 91, 45, 76, 87, 54, 99};

main()
{
  void display(const int a[], int n, int count = 1);

  cout << "Here's the array, one per line.\n";
  display(a, 10);          // one per line
  cout << "Here's the array, three per line.\n";
  display(a, 10, 3);       // three per line
  return 0;
}

void display(const int a[], int n, int count)
{
  int i;

  for (i = 1; i <= n; i++)
  {
    cout << a[i - 1];
    if (i % count == 0 || i == n)
      cout << "\n";
    else if (i != 0)
      cout << " ";
  }
}
```

# stack.h

(Figure 22.7 – Page 563)

```
//
// Simple stack definition, with operations.
//
const int MAXSTACK = 100;          // default stack size

struct stack
{
  private:
    int count;                     // # of elements on stack
    int items[MAXSTACK];           // the stack itself
  public:
    stack();                       // initialize stack!
    void push(int item);           // add element to the stack
    void pop();                    // delete top element from stack
    int top();                     // return top element of stack
    int isempty();                 // is stack empty?
};
```

# stack.C

(Figure 22.8 – Page 565)

```
//
// Simple stack definition, with operations.
//
#include "stack.h"

stack::stack()
{
  count = 0;                      // stack starts off empty
}

void stack::push(int item)        // add element to stack
{
  items[count++] = item;
}

void stack::pop()                 // delete top element from stack
{
  count--;
}

int stack::top()                  // return top element of stack
{
  return items[count-1];
}

int stack::isempty()              // is stack empty?
{
  return count == 0;
}
```

# rev.C

(Figure 22.9 – Page 566)

```
//
// Use stack to reverse stream of integers, no error checking.
//
#include <iostream.h>
#include "stack.h"

main()
{
  stack s;                              // create a stack
  int   i;

  while (cin >> i)                      // push inputs onto stack
    s.push(i);
  for (; !s.isempty(); s.pop())         // display stack
    cout << s.top() << "\n";
  return 0;
}
```

# stack2.h

(Figure 22.10 – Page 567)

```
//
// Revised stack definition, with operations.
//
const int MAXSTACK = 100;       // default stack size

class stack
{
    int count;                  // # of elements on stack
    int *items;                 // the stack itself
    int size;                   // the size of a stack
  public:
    stack(int n = MAXSTACK);    // create stack
    ~stack();                   // get rid of a stack
    void push(int item);        // add element to the stack
    void pop();                 // delete top element from stack
    int top();                  // return top element of stack
    int isempty();              // is stack empty?
};
```

# stack2.C

(Figure 22.11 – Page 568)

```
//
// Simple stack definition, with operations.
//
#include "stack2.h"

stack::stack(int n)
{
  items = new int[n];               // make stack
  count = 0;                        // stack starts off empty
  size = n;                         // elements in stack
}

stack::~stack()                     // free up stack
{
  delete [] items;
}

void stack::push(int item)          // add element to stack
{
  if (count < size)
    items[count++] = item;
}

void stack::pop()                   // delete top element from stack
{
  if (count != 0)
    count--;
}

int stack::top()                    // return top element of stack (or 0)
{
  return count > 0 ? items[count-1] : 0;
}

int stack::isempty()                // is stack empty?
{
  return count == 0;
}
```

# filecopy.C

(Figure 22.12 – Page 570)

```
//
// Copy one file to another.  First argument is source file, and
// second argument is destination.
//
#include <iostream.h>                          // For basic I/O
#include <fstream.h>                           // For file I/O

inline void usage_error(char *progname, char *arguments);
inline void open_error(char *filename, char *access_type);

inline void usage_error(char *progname, char *arguments)
{          // error message for bad program usage
  cerr << "Usage: " << progname << " " << arguments << "\n";
}

inline void open_error(char *filename, char *access_type)
{          // error message for failure to open files
  cerr << "Can't open " << filename << " for " << access_type << "\n";
}
```

# filecopy.C

(Figure 22.12 – Page 570)

```
int main(int argc, char *argv[])
{
  char ch;
  int  status = 1;                        // default is failure!

  if (argc != 3)                          // two files required
    usage_error(argv[0], "file1 file2");
  else
  {
    ifstream  f1(argv[1]);                // open input file

    if (!f1)
      open_error(argv[1], "reading");
    else
    {
      ofstream  f2(argv[2]);              // open output file

      if (!f2)
        open_error(argv[2], "writing");
      else
        while (f1.get(ch))                // do the copy
          f2.put(ch);
      status = 0;
    }
  }
  return status;
}
```

# lists.h

(Figure 22.13 – Page 571)

```
//
// Simple linked list class.
//
#include <iostream.h>

struct node
{
  node *next;
  int item;
};

class list
{
  protected:
    node *first, *last;                          // ptrs to front/back
    int items;                                   // number of values
  public:
    list() {first = last = 0; items = 0;}       // initialize list
    ~list();
    void insert_front(int value);
    void insert_rear(int value);
    int  isempty() { return first == 0;}         // is list empty?
    int  length() { return items; }              // # of items in list
    friend ostream& operator<<(ostream &output, const list &l);
};
```

 

# lists.C

(Figure 22.14 – Page 573)

```
//
// Simple linked list class.
//   ~list - free up list by deleting all elements.
//   insert_front - insert new item at front of list.
//   insert_rear - insert new item at end of list.
//
#include <iostream.h>
#include "lists.h"

list::~list()
{
  node *ptr;

  for (ptr = first; ptr != 0; ptr=ptr->next)
    delete ptr;
}

void list::insert_front(int value)
{
  node *temp = new node;

  temp->next = first;                       // create and fill node
  temp->item = value;
  first = temp;                             // hook into list
  if (!last)                                // only last if empty
    last = temp;
  items++;                                  // new node
}
```

# lists.C
(Figure 22.14 – Page 573)

```
void list::insert_rear(int value)
{
  node *temp = new node;

  temp->next = 0;                          // create and fill node
  temp->item = value;
  last->next = temp;
  last = temp;
  items++;
}

ostream& operator<<(ostream& output, const list &l)
{
  node *ptr;

  for (ptr = l.first; ptr != 0; ptr = ptr->next)
    output << ptr->item << "\n";
  return output;
}
```

# stack3.h
(Figure 22.15 – Page 574)

```
//
// Stack class implemented as derived class.
//
#include "lists.h"

class stack : public list
{
  public:
    void push(int value) { insert_front(value); }
    void pop();
    int  top()  { return first ? first->item : 0; }
};
```

# stack3.C

(Figure 22.16 – Page 575)

```
//
// Definitions of stack operations in terms of lists.
//
#include "stack3.h"

void stack::pop()
{
  node *temp = first;

  if (first)
  {
    first=first->next;
    delete temp;
  }
}
```

# stack4.h

(Figure 22.17 – Page 576)

```
//
// Stack class implemented as derived class.
//
#include "lists.h"

class stack : public list
{
  private:                              // stack users can't get at these
    void insert_front(int value);
    void insert_rear(int value);
    int length();
    friend ostream& operator<<(ostream& output, const list &l);
  public:
    void push(int value) { list::insert_front(value); }
    void pop();
    int  top() { return first ? first->item : 0; }
};
```

# CHAPTER SUMMARY

- C++ is one implementation of object-oriented programming, which has been around for quite some time and is here to stay.[1]

- C++ is built on C. It provides a variety of features that are nice to have, even without knowing all of its object-oriented capabilities. But C++'s powerful object-oriented features allow us to easily write clear, reuseable code.

- C++ provides references, which we can use to simulate call-by-reference parameter passing and to efficiently pass large structures.

- C++ provides function and operator overloading, which allows us to have the same functions and operators work with a variety of different types.

- C++ provides **`inline`** functions, which allow us to trade size for speed with small functions.

- C++ allows us to provide functions as structure and class members. These functions then become the operations that we can apply to objects of that class.



[1] The language is still evolving. There is now an ANSI standard for the C language; no such standard exists for C++, and the differences between AT&T's versions 1.2, 2.0, 2.1, and 3.0 have been substantial. At this time, the language is actively being worked on, and there could be major changes in new versions.

# CHAPTER SUMMARY

- C++ lets us define classes that are derived from other classes. These derived classes automatically inherit instance data and member functions from their base classes.

- We really like C++. It's an elegant extension to C that opens the door to the strengths, advantages, and conveniences of object-oriented programming. But like C before it, it is hardly a panacea for all programming woes, and it brings along its own potential problems and pitfalls.