DECOMPOSITION AND ABSTRACTION

*Decomposition*: large programs are partitioned into smaller pieces that are implemented by one or more people. An *abstraction* consists of just those properties essential to a purpose; details that can safely be ignored are hidden.

**Forms of Abstraction:**
- Procedures
- Modules
- Abstract data
- Objects: object-oriented programming treats an overall system as a collection of interacting objects.
PROCEDURES

*Procedures* have been in use since the earliest days of programming. Once a procedure is defined, its implementation can be abstracted away.

*Function procedures* can be thought of as extending the built-in operators of a language, and *proper procedures* can be thought of as extending built-in actions.

-----

AN EXPRESSION EVALUATOR

CHARACTER STREAM (512-487) * 2;

(512-487) * 2;

\[
\text{lparen} \number_{512} \text{minus} \number_{487} \text{rparen times} \number_{2} \text{semicolon}
\]

TOKEN STREAM

EXPRESSION VALUE

50
The idea that data and operations go together is the basis for modules. With modules, the groupings of variables and procedures are explicit in the source text.

A module is a collection of declarations, typically including both variables and procedures. The interface of a module is a subset of declarations in the module. An implementation of the module consists of everything else about the module.

We cannot create new modules or copies of existing modules dynamically as a program runs.

Programming with modules:

- Describe the roles of the modules (in general terms).
- Design the interfaces.
- Implement the interfaces, hiding design decisions in the private part.
PUBLIC AND PRIVATE VIEWS OF MODULES

MODULE Scanner

procedure scan;
type token;
var lookahead;
var lookvalue;
var ch;
var tok;
procedure getch;
function getnum;

initiation

procedure bodies

MODULE Parser

function expr;
function factor;
function term;

initiation

procedure bodies

USING MODULES

MODULE StackManager

type stack;
function pop(s : stack) : integer;
procedure push(a : integer; s : stack);
Function newstack : stack;

representation of stacks

initiation

procedure bodies

var s,t : stack;
begin
s := newstack;
t := newstack;
push(10,s)
...
end.
A Pascal Program to Remove Adjacent Duplicates

Program uniq (input, output);
var x, next : integer
begin
  read(x);
  while x <> 0 do begin
    writeln(x);
    do read(next) while next=x;
    x := next;
  end;
end.

WHEN ENTRIES ARE NOT BUILT-IN TYPES

CLASS Entry
procedure read();
procedure write();
function endmarker() : boolean;
function equal(Entry) : boolean;
function copy(Entry) : boolean;
constructor Entry;
destructor ~Entry;
representation of entries
...
procedure bodies

Entry e.f;
e.read();
while not e.endmarker() do begin
  e.write();
do f.read(); while e.equal(f);
e.copy(f);
end
DATA ABSTRACTION

A **data type** is a set of objects and operations on those objects. A **data abstraction** in a programming language is a mechanism which collects together (or **encapsulates**) the representation and the operations of a data type. The **encapsulation** forms a wall which is intended to shield the data type from improper use, and provides a “window” which allows the user a well-defined means for accessing the data type.

The term **abstract data type** is often used in the literature to refer to a class of objects which are defined by a **representation independent specification**.

```
STACK
   pop ()
   d
   c
   b
   a
```

INFORMATION HIDING

An **abstract specification** tells us the behavior of an object independent of its implementation; that is, an abstract specification tells us what an object does independent of how it works.

A **concrete representation** tells us how an object is implemented, how its data is laid out inside a machine, and how this data is manipulated by its operations.

The **implementation hiding** principle: design a program so that the implementation of an object can be changed without affecting the rest of the program.

**Scope rules**, which control the visibility of names, are the primary tool for achieving implementation hiding.
### PRIVATE AND PUBLIC VIEWS OF A BOUNDED BUFFER

<table>
<thead>
<tr>
<th>Operation</th>
<th>PUBLIC VIEW</th>
<th>PRIVATE VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(a)</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>put(b)</td>
<td>a b</td>
<td>b</td>
</tr>
<tr>
<td>get( )</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>put(c)</td>
<td>b c</td>
<td></td>
</tr>
<tr>
<td>get( )</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>put(d)</td>
<td>c d</td>
<td>d</td>
</tr>
</tbody>
</table>

### DATA INVARIANTS

A grouping of data and operations has a local state, consisting of the values of its variables.

A **data invariant** for an object is a property of its local state that holds whenever control is not in the object.

**Design an object around data invariants:**

- **Initialization of Private Variables**
  
  Since the private data of an object is inaccessible from outside, initialization of the data belongs to the code for the object. Initialization is needed to set up data invariants when the object is created.

- **Assignments to Public Variables**
  
  Assignments to public variables can change the local state of an object. It is up to the user to ensure that such assignments do not disturb the desired data invariants.
The term *class* is an abbreviation of "class of objects." A class corresponds to a type.

An *object* is a run-time entity with data on which operations can be performed. Objects can be created and deleted at run time.

**Example:**

```cpp
class Stack {
    public:
        Stack();
        void push(int a);
        int pop();
    private:
        ...
};
```

```
Stack s, t;
s.push(7);
t.push(8);
```

**Procedures** *push* and *pop* operate on private data.

In C++’s terminology, the procedure *Stack*, with the same name as the class, is a *constructor*. The constructor is called automatically when an object of the class is created, so initialization code can be put in the constructor.

A class can also have a *destructor* procedure, which is called automatically just before the object disappears.
Classes in C++ are a generalization of records, called \textit{structures} in C and C++. A structure is traditionally a grouping of data; C++ allows both data and functions to be structure members. Below is an example:

```cpp
struct Stack {
    int top;
    char elements[101];
    char pop();
    void push(char);
    Stack() { top = 0; }
};
```

```cpp
char Stack::pop() {
    top = top - 1;
    return elements[top+1];
}
```

```cpp
void Stack::push(char c) {
    top = top + 1;
    elements[top] = c;
}
```

```cpp
#include <stdio.h>
main() {
    Stack s;
    s.push('!'); s.push('@'); s.push('#');
    printf("%c %c %c\n", s.pop(), s.pop(), s.pop());
}
```
OVERLOADED FUNCTION NAMES

The same name can be given to more than one function in a class, provided we can tell the overloaded functions apart by looking at the number and types of their parameters. Constructors are functions, so they too can be overloaded.

Example:
```c++
struct Complex {
    float re;
    float im;
    Complex(float r) { re = r; im = 0; }
    Complex(float r, i) { re = r; im = i; }
};
```

PUBLIC, PRIVATE, AND PROTECTED MEMBERS

Privacy and access control in C++ are class based. That is, access to members is restricted through keywords in a class declaration.

C++ has three keywords—public, private, and protected—for controlling the accessibility of member names in a class declaration:

- **Public members** are accessible to outside code.
- **Private members** are accessible to the member functions in this class declaration. They are accessible to all objects of this class.
- **Protected members** behave like private members, except for derived classes. Protected members are visible through inheritance to derived classes but not to other code.
DYNAMIC ALLOCATION IN C++

C++ objects can be created in three ways:
- through variable declarations,
- dynamically through `new`, and
- as static objects whose lifetime is the entire life of the program.

EXAMPLE: A CIRCULARLY LINKED LIST

class Cell {
    int info;
    Cell *next;
    Cell(int i) { info = i; next = this; }
    Cell(int i, Cell *n) { info = i; next = n; }
    friend class List;
};

class List {
    Cell *rear;
public:
    void put(int);
    void push(int);
    int pop();
    int empty() { return rear==rear->next; }
    List() { rear = new Cell(0); }
    ~List() { while (!empty()) pop(); }
};
A CIRCULARLY LINKED LIST (CONT.)

```cpp
void List::push(int x) {
    rear->next = new Cell(x, rear->next);
}

void List::put(int x) {
    rear->info = x;
    rear = rear->next = new Cell(0, rear->next);
}

int List::pop() {
    if (empty()) return 0;
    Cell *front = rear->next;
    rear->next = front->next;
    int x = front->info;
    delete front;
    return x;
}
```

TEMPLATES: PARAMETERIZED TYPES

```cpp
template<class T> class Stack {
    int top;
    int size;
    T *elements;
public:
    Stack(int n) {
        size = n; elements = new T[size]; top = 0;
    }
    ~Stack() { delete elements; }
    void push(T a) { top++; elements[top] = a; }
    T pop() { top--; return elements[top+1]; }
};

Usage:
Stack<int> s(99);
Stack<char> t(80);
```
A C++ IMPLEMENTATION

```c++
struct Stack {
    int top;
    char elements[101];
    char pop();
    void push(char);
    Stack() { top = 0; }
};
char Stack::pop() {
    top = top - 1;
    return elements[top+1];
}
void Stack::push(char c){
    top = top + 1;
    elements[top] = c;
}
```

A C++ IMPLEMENTATION (cont.)

```c++
#include <stdio.h>
main() {
    Stack s;
    s.push('!');
    s.push('@');
    s.push('#');
    printf("%c %c %c\n", s.pop(), s.pop(), s.pop());
}
```
**IN-LINE EXPANSION**

Implementation hiding can result in lots of little functions that manipulate the data in an object. C++ implements such functions efficiently by using *in-line expansion*, which replaces a call by the function body. In-line expansion in C++ preserves the semantics of call-by-value parameter passing.

Suppose a public function `isempty` is added to class `stack`:

```cpp
int isempty() { return top == 0; }
```

With in-line expansion, the following conditional statement

```cpp
if ( s isempty() )...
```

expands to

```cpp
if ( (s.top == 0) ) ...
```

In-line expansion eliminates the overhead of function calls at run time, so it encourages free use of functions. It also encourages data hiding.

---

**MODULES VS. CLASSES**

A module partitions the next of a program into manageable pieces.

Modules are static; we cannot create new modules or copies existing modules dynamically as a program runs.

A module serves as a black box with which the rest of the program interacts through an interface.

The interface of a module is a collect of declarations of types, variables, procedures, and so on.

An implementation of the module consists of everything else about the module.

Interfaces and implementations are also referred to as the public and private views, respectively, of the modules.

A module is said to have a local state because its variables retain their values when control is not in the module.
MODULES VS. CLASSES (cont.)

A class corresponds to a type. “Class” is an abbreviation of “class of objects”.

Objects are dynamic; we can create and delete objects at run time.

A module sets up a single object. The preceding remarks about modules extend to classes.

Ex. Airport check-in counters.

Objects in this example are: ticket agents, passengers, and queues.

---

THE OBJECT-ORIENTED FAMILY

---