Compositional C++ is a small set of extensions to C++ for parallel programming.

OVERVIEW OF C++

With a few exceptions, C++ is a pure extension of ANSI C. Its features:

Strong typing and memory management:

All functions must have prototypes provided. C++ enforces consistent use of function use between programs, while allowing functions to be *overloaded*.

Example:

```
/* A forward declaration to a_function */
int a_function(float b, double c);

/* The definition of a_function */
int a_function(float b, double c) {
  /* Function body */
}
```

In standard C, library routines **`malloc`** and **`free`** were used for dynamic memory allocation. C++ has two new constructs: new and delete.

Example:

```
struct S {
   /* Structure body */
};

S *sPtr = new S;          /* Allocate instance of S */
delete sPtr;              /* Delete instance of S */

int *iPtr = new int[25]; /* Allocate array of integers */
delete [] iPtr;           /* Delete array of integers */
```

Note that **`new`** is given the description of the data type to allocate.
`delete` must be told whether it deletes a single entity or an array of entities.

OVERVIEW OF C++

Classes:

A class can be seen as a generalization of a C structure.
C structure is a collection of data elements of various types;
C++ structure may also contain *member functions*, accessed in the same way as data elements – as references to an object of the appropriate type.

In C++ a class defines a scope in which names referring to functions or data can be defined.

Constructors, if defined, are called whenever an object of a given class is created. There may be a *default constructor* (without arguments) which is called implicitly.

Class mechanism also supports protection (i.e. information hiding) –

Class elements may be public or private.

EXAMPLE OF C++ CLASS DECLARATIONS

```
// Define a Datum class
class Datum {
public:
   int get_x(); // Public member functions
   Datum();
   Datum(int);
private:
   int x;        // Private variable
};

// A member function of Datum
int Datum::get_x() { return x; }

// Constructors for Datum
Datum::Datum() { x = 0; }
Datum::Datum(int arg_x) { x = arg_x; }

void test() {
   Datum a_datum;                    // Declare a datum
   Datum another_datum(23);          // Declare a second datum
   Datum *DatumPtr = new Datum(23);// Allocate third datum
   int val = a_datum.get_x();        // Access x in first datum
   int val1 = DatumPtr->get_x();     // Access x in third datum
   delete DatumPtr;                  // Delete third datum
}
```

Program 5.1 : Example of a C++ class declaration. This code fragment declares a class `Datum` and defines a member function of this class (`Datum::get_x`) and two constructor functions (`Datum::Datum`). The function `test` illustrates the use of the class.

OVERVIEW OF C++

Inheritance:

In C a structure may contain an element which itself is another structure (i.e. structures can be nested). This is an example of a *has-a* relationship.

In C++ inheritance is used to define an *is-a* relationship between classes. If a class D inherits from B, then all public members of B are also members of D. We say that D is derived from B, i.e. D is a derived class, while B is a base class.

Simply put: D is a specialized version of B, hence the *is-a* relationship.
Example of use of inheritance:

```
class B {                  // B is a base class
public:
   void base_func1();      // Public member functions, inherited
   void func2();           // by classes derived from B
};

class D : public B {       // D is derived from B
public:
   void func2();           // D redefines func2
};

void test() {
   B b;                    // Create instance of B
   D d;                    // Create instance of D
   d.base_func1();         // Call base_func1 defined in B
   b.func2();              // Call func2 defined in B
   d.func2();              // Call func2 defined in D
}
```

Program 6.2 : A program illustrating inheritance in C++ . The class D is derived from the class B. The function test creates instances of these classes and calls the functions that they define.

CC++ BASICS

CC++ is a strict superset of C++. The CC++ extensions implement six basic abstractions:

- The *processor object* is a mechanism for controlling locality. A computation may comprise one or more processor objects. Within a processor object, sequential C++ code can execute without modification. In particular, it can access local data structures. The keyword **`global`** identifies a processor object class, and the predefined class **`proc_t`** controls processor object placement.

- The *global pointer*, identified by the type modifier **`global`**, is a mechanism for linking together processor objects. A global pointer must be used to access a data structure or to perform computation (using a remote procedure call, or RPC) in another processor object.

- The *thread* is a mechanism for specifying concurrent execution. Threads are created independently from processor objects, and more than one thread can execute in a processor object. The **`par`**, **`parfor`**, and **`spawn`** statements create threads.

- The *sync variable*, specified by the type modifier **`sync`**, is used to synchronize thread execution.

- The *atomic function*, specified by the keyword **`atomic`**, is a mechanism used to control the interleaving of threads executing in the same processor object.

- *Transfer functions*, with predefined type **`CCVoid`**, allow arbitrary data structures to be transferred between processor objects as arguments to remote procedure calls.

These abstractions provide the basic mechanisms required to specify concurrency, locality, communication, and mapping.

CC++ BASICS

Illustration of many CC++ features:

Example: An implementation of the bridge construction algorithm. The program creates two tasks, foundry and bridge, and connects them with a channel. The channel is used to communicate a stream of integer values 1..100 from foundry to bridge, followed by the value --1 to signal termination.

While the concepts of task and channel are not supported directly in CC++ , they can be implemented easily by using CC++ mechanisms. Hence, the main program creates two tasks by first using the new operator to create two processor objects and then using the par construct to create two threads, one per processor object. The two tasks engage in channel communication by invoking functions defined in a Channel class. A channel is declared in the main program and passed as an argument to foundry and bridge. These processes use access functions **get_out_port** and **get_in_port** to obtain pointers to **out-port** and **in-port** objects to which can be applied functions **send** and **receive**, respectively.

```
// The Construction class is derived from the ChannelUser class.
global class Construction :  public ChannelUser {
public:         // It has two public member functions
   void foundry(Channel, int);
   void bridge(Channel);
};

void Construction::foundry(Channel channel, int cnt) {
    // Extract outport from channel
    OutPort *out = (OutPort *) channel.get_out_port();
    // Send cnt values
    for (int i=1; i<=cnt; i++)
        out->send(i);   // Invoke method in OutPort class
    // Send end-of-channel value
    out->send(-1);
}

void Construction::bridge(Channel channel) {
    int val;
    // Extract inport from channel
    InPort *in = (InPort *) channel.get_in_port();
    // Receive values until end detected
    while ((val = in->receive()) != -1)
        printf("Got value %
}

void main(int argc, char *argv[]) {
   // Create new processor objects for foundry and bridge
   Construction *global foundry_pobj = new Construction;
   Construction *global bridge_pobj  = new Construction;
   // Create a channel
   Channel channel(foundry_pobj, bridge_pobj);
   // Run foundry and bridge, passing channel
   par {
     foundry_pobj->foundry(channel, 100);
     bridge_pobj ->bridge(channel);
   }
}
```

Program 5.3 : CC++ implementation of bridge construction problem. The Channel class is defined later in the chapter.

CONCURRENCY IN CC++

A CC++ program initially executes a single thread of control. Additional threads can be created using **par**, **parfor** and **spawn** constructs.

```
par {
      statement1;
      statement2;
      ...
      statementN;
}
```

These may be single statements, or function calls, viz:

```
par {
   worker();
   worker();
   master();
}
```

A parallel blocks terminates when all its constituent threads terminate.

A parallel-for loop:

```
parfor (int i=0; i<10; i++) {
   myprocess(i);
}
```

CC++ parallel constructs can be nested arbitrarily:

```
par {
   master();
   parfor (int i=0; i<10; i++)
      worker(i);
}
```

LOCALITY IN CC++

Processor objects represent address spaces; threads represent threads of control.

Processor objects:

A processor object is defined by a C++ class declaration modified by the keyword global. A processor object is identical to a normal C++ class definition in all but two respects:

1. Names of C++ ``global'' variables and functions (that is, names with file scope) refer to unique objects within different instances of a processor object. Hence, there is no sharing between processor object instances.

2. Private members of a processor object need not be explicitly declared to be private. C++ ``global'' functions and variables are defined implicitly to be private members of the processor object in which they occur.

Processor object types can be inherited, and the usual C++ protection mechanisms apply, so private functions and data are accessible only from a processor object's member functions or from the member functions of derived objects. Hence, it is the member functions and data declared public that represent the processor object's interface.

For example, the following code creates a processor object class Construction with public member functions foundry and bridge. The class ChannelUser is specified as a base class and provides access to channel operations.

```
global class Construction : public ChannelUser {
public:
   void foundry(Channel, int);
   void bridge(Channel);
};
```

LOCALITY IN CC++

Global Pointers:

A *processor object* is a unit of locality, that is, an address space within which data accesses are regarded as local and hence cheap. A thread executing in a processor object can access data structures defined or allocated within that processor object directly, by using ordinary C++ pointers.

Processor objects are linked together using *global pointers*. A global pointer is like an ordinary C++ pointer except that it can refer to other processor objects or to data structures contained within other processor objects. It represents data that are potentially nonlocal and hence more expensive to access than data referenced by ordinary C++ pointers.

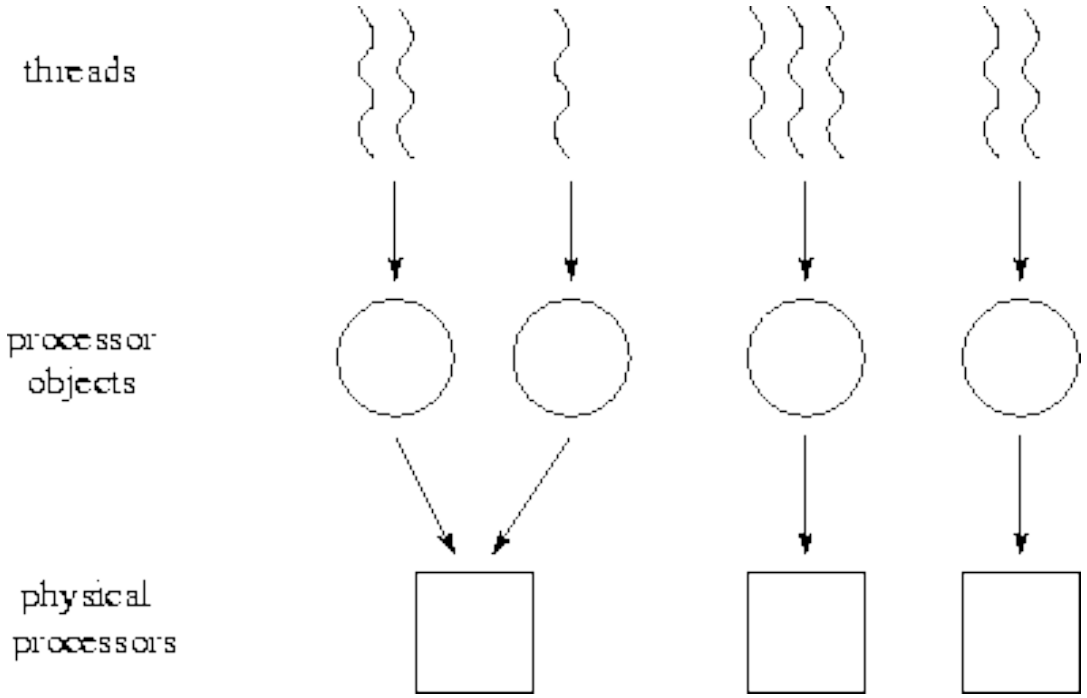
A global pointer is distinguished by the keyword **`global`**. For example:

```
float *global gpf;   // global pointer to a float
char * *global gppc; // global pointer to pointer of type char
C *global gpC;       // global pointer to an object of type C
```

When the **`new`** statement is used to create an instance of a processor object, it returns a global pointer. For example, the statement

```
Construction *global foundry_pobj = new Construction;
```

PLACEMENT ISSUES IN CC++



THREAD PLACEMENT IN CC++

By default, a CC++ thread executes in the same processor object as its parent. Computation is placed in another processor object via an RPC. A thread needs only a global pointer to another processor object to be able to invoke any of its public member functions. For example, in the following line **`bridge_pobj`** is a global pointer to the processor object on which the consumer is to execute, and **`bridge`** is a public member function of that object.

```
bridge_pobj->bridge();
```

THREAD PLACEMENT IN CC++

Example: Parallel search of a tree. Note the use of a parallel block to search branches of the tree simultaneously:

```
global class Tree { // Processor object: one member function
public:
   int search(int);
};

int Tree::search(int A) {
   int ls, rs;
   if(leaf(A)) {    // Leaf node: check whether a solution
      if solution(A)
         return(1);
      else
         return(0);
   }
   else {           // Nonleaf node: explore subtrees
      Tree *global lobj = new Tree;
      Tree *global robj = new Tree;
      par {         // Create processes to search subtrees
         ls = lobj->search(left_child(A));
         rs = robj->search(right_child(A));
      }
      delete(lobj); delete(robj);
      return(ls+rs);
   }
}
 void main(int argc, char *argv[]) {
   int total;
   // Create new processor object for search
   Tree *global searcher = new Tree;
   // Initiate search
   total = searcher->search(1);
   printf("There were %
}
```

COMMUNICATION IN CC++

In CC++ there are no low-level primitives for directly sending and receiving data between threads. Instead, threads communicate by operating on shared data structures. We need to explain here:

1. How global pointers are used to transfer data between processor objects
2. How sync variables and atomic functions are used to provide synchronization and mutual exclusion
3. How data transfer functions are used to communicate arbitrary data structures

Here we go:

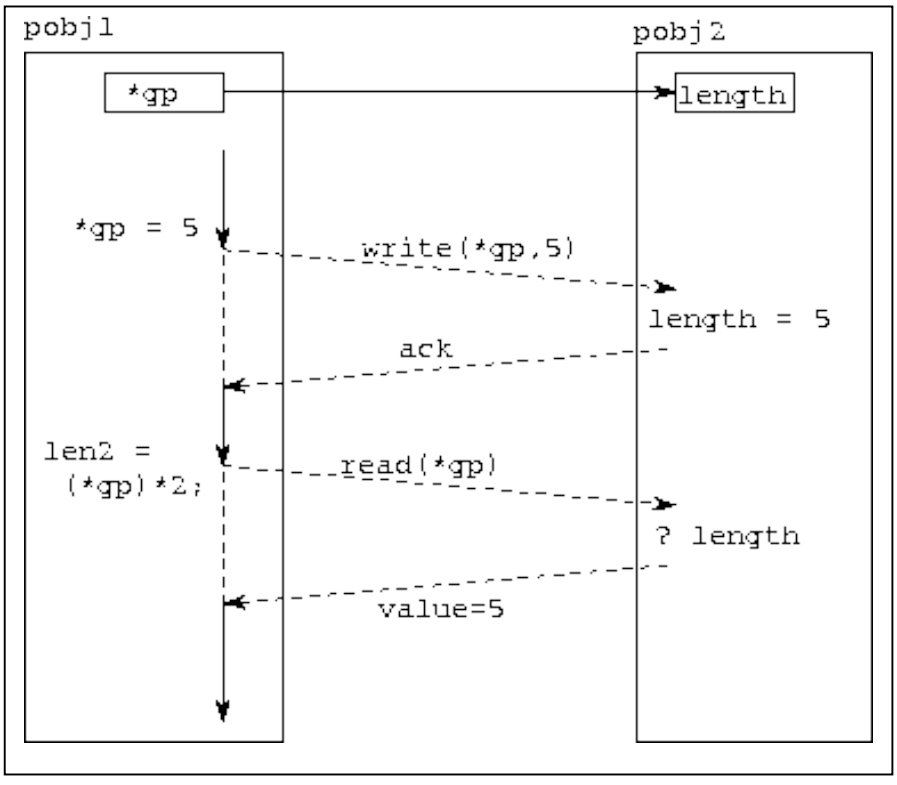
Remote operations:

CC++ global pointers refer to processor objects, but are used exactly like the C++ pointers.

The following code:

```
global int *gp;
int len2;
*gp = 5;
len2 = (*gp) * 2;
```

results in the communications shown beside:



A RPC general form:

```
<type> *global gp;
result = gp->p(...);
```

COMMUNICATION IN CC++

A RPC general form:

```
<type> *global gp;
result = gp->p(...);
```

A RPC progresses in three stages:

1. The arguments to the function `p(...)` are packed into a message, communicated to the remote processor object, and unpacked. The calling thread suspends execution.

2. A new thread is created in the remote processor object to execute the called function.

3. Upon termination of the remote function, the function return value is transferred back to the calling thread, which resumes execution.

Example:

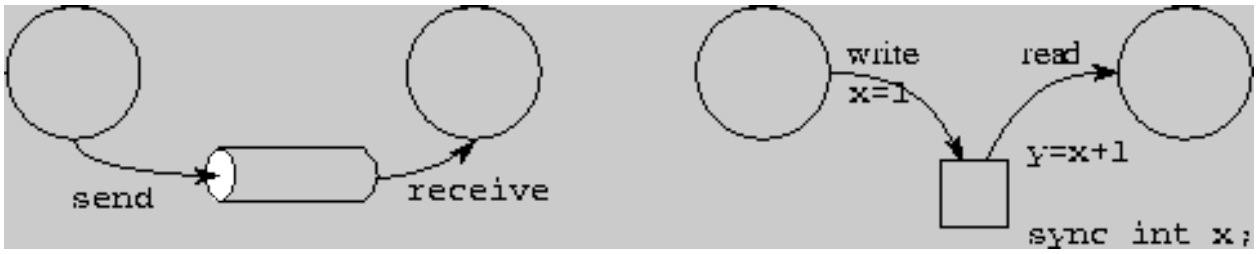
```
int length;          // Global variable: implicitly private

global class Length {
public:
   int read_len()              { return(length); }
   void write_len(int newval) { length = newval; }
};

// Test program: create and operate on Length objects
void test() {
   int len, len2;
   // Allocate new processor object
   Length *global lp = new Length;
   // Write the private variable length
   lp->write_len(5);
   // Read the private variable length
   len  = lp->read_len();
   len2 = len*2;
}
```

SYNCHRONIZATION IN CC++

Possible synchronization mechanisms: using a channel (left), or using a **sync** variable (right):



A **sync** variable is defined by type modifier "**sync**". Its properties:

1. It initially has a special value, ``undefined.''
2. It can be assigned a value at most once, and once assigned is treated as a constant (ANSI C and C++ const).
3. An attempt to read an undefined variable causes the thread that performs the read to block until the variable is assigned a value.

A **sync** variable can be thought of as an empty box with glued interior: an object, once placed there, cannot be removed.

All C++ types and CC++ global pointers can be declared sync. Examples:

```
sync int i;          // i is a sync integer
sync int *j;         // j is a pointer to a sync integer
int *sync k;         // k is a sync pointer to an integer
sync int *sync l;    // l is a sync pointer to a sync integer
```

SYNCHRONIZATION IN CC++

Example of use: A queue with two operations: enqueue and dequeue.
Note the use of the sync variable in IntQData, used to synchronize the enqueue and dequeue operations:

```
struct IntQData {                  // A list element contains:
  sync int value;                  //   sync variable (message), &
  struct IntQData *next;           //   pointer to next list element
};

class Queue {
public:
   void enqueue(int);
   int dequeue();
private:
   // Initialize: allocate single element
   void Queue() { head = tail = new IntQData; }
   IntQData *head, *tail;
};

void Queue::enqueue(int msg) {  // Enqueue a value:
   tail->next  = new IntQData;  //   allocate new element,
   tail->value = msg;           //   set message value, &
   tail        = tail->next;    //   advance tail pointer
}

int Queue::dequeue() {          // Dequeue a value:
   int retval = head->value;    //   access message value,
   IntQData *newh = head->next; //   get next list item,
   delete head;                 //   delete old list head,
   head = newh;                 //   advance head pointer, &
   return retval;               //   return message value
}
```

MUTUAL EXCLUSION IN CC++

The sync variable prevented retrieval of data from the queue before the data were placed there, but what to do if two producers were to try to deposit info in the queue simultaneously?

Methods of a given objects may be declared **atomic**. This guarantees that an the execution of an atomic method will not be interleaved with an execution of another atomic method of the same object. Consider:

```
atomic void Queue::enqueue(int msg) {
   tail->next  = new IntQData;
   tail->value = msg;
   tail        = tail->next;
}
```

PROCESSOR PLACEMENT IN CC++

The issue here: How do we allocate processor objects to physical processors?

By default, a newly created object is placed on the same physical processor as its creator. The following code fragment places a processor object on a CPU named “mymachine”:

```
MyClass *global G;
proc_t location(node_t("mymachine"));
G = new (location) MyClass;
```

The following code creates 32 processor objects, placing each on a different processor of a multicomputer with nodes named sp#0, sp#1, ..., sp#31. Notice how parfor is used to create the different processor objects concurrently.

```
MyClass *global G[32];
parfor (int i=0; i<31; i++) {
   char node_name[256];
   sprintf(node_name,"sp#%
   proc_t location(node_t(node_name));
   G[i] = new (location) MyClass;
}
```

