Building Interactive Distributed Applications in C++ with The Programmers' Playground

Kenneth J. Goldman, Joe Hoffert, T. Paul McCartney, Jerome Plun, Todd Rodgers

WUCS-97-14

February 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

The Programmers' Playground

Reference Manual

*Revised for **Playground C++ UNIX Veneer v3.0.3***
February 1997

Earlier version published as
Washington University technical report WUCS-95-20.

Copyright (c) 1993-1997 by

Distributed Programming Environments Group
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

The objective of The Programmers' Playground, described in this manual, is to provide a development environment and underlying support for end-user construction of distributed multimedia applications from reusable self-describing software components. Playground provides a set of software tools and a methodology for simplifying the design and construction of applications that interact with each other and with people in a distributed computer system.

This manual explains how to write interactive distributed applications using Playground. The only background necessary to get started is an understanding of basic data structures and control constructs in C++. If you already know C++, then with the tools provided by Playground, you will be able to write distributed applications *without learning a new programming language* and *without needing to learn about how communication works in a distributed system.*

Table of Contents

Chapter 1

Introduction

Distributed multimedia applications, supporting interaction among many users and system components, can facilitate effective communication and synthesis of information. This effectiveness is compounded when end-users are empowered with the ability to dynamically integrate and customize these applications in order to take advantage of new components and new information sources.

The Programmers' Playground provides a methodology and set of software tools for writing interactive distributed applications. Playground offers an abstraction that serves as an insulating layer between the programming language and low-level communication protocols, and provides a uniform approach to communication that accommodates diverse collections of both persistent and transient applications. This abstraction:

- simplifies the construction of distributed applications,
- provides end-user configuration and integration of software modules,
- is designed for high-bandwidth communication technology,
- provides uniform treatment of discrete and continuous data,
- permits a dynamically changing communication structure,
- offers protection for data and applications,
- supports existing programming languages and paradigms,
- is designed for scalability and modularity,
- rests on a formal foundation, and
- is compatible with a connection-oriented model of communication services.

Playground is neither a new programming language nor a new operating system. It is a way of thinking about distributed applications and software to support that way of thinking. This chapter describes the Playground philosophy. Later chapters describe the software library and tools that enable you to write applications in C++ using this philosophy.

1.1 Basic Concepts

A distributed application is composed of multiple processes, usually running on separate computers, that communicate. The Programmers' Playground C++ library, described in Chapter 3 through Chapter 5 allows programmers to write C++ programs to implement these processes and their associated communication without worrying about the low-level details of interprocess communication.

Playground is based on a model of distributed computing called *I/O abstraction.* Briefly, I/O abstraction is the view that each *module* (i.e., process) in a system has a set of data structures that may be externally observed and/or manipulated. This set of data structures forms the module's external interface, called the *presentation.* Each module is written independently and modules are then *configured* by establishing *logical connections* between the data structures in their presentations. The connections must respect access restrictions established by each module for its own data. As published data structures are modified within a module, communication occurs implicitly "under the covers" to other modules according to the connections created in the configuration.

Playground helps to simplify applications programming by treating communication as a high-level relationship among module states. Program I/O occurs implicitly as a result of this relationship. In this way, low level input and output activities are hidden from programmer. Programmers need not be concerned with explicitly initiating communication activities, such as sending and receiving messages, and therefore need not be concerned with the particular communication primitives provided by the operating system or the network interface.

1.1.1 Variables

Variables may be private to a module or may be *published*, meaning that other modules in the system may access the values of the variables. Playground provides a library of data types that may be published. These data types are divided into three categories: *base types* for storing integer, real, boolean, string, and memory block values, *tuples* for storing records with various fields, and *aggregates* for organizations of homogeneous collections of elements. The fields of tuples and the elements of aggregates may be nested arbitrarily using the Playground data types.

Each Playground module has a presentation that consists of published variables. Each published variable has certain associated information:

- *public name:* A descriptive textual name assigned by the module program when the data item is published. The name helps users of the module to understand the module's presentation.
- *data type:* The data type of the published data item, automatically associated with each data item. This information is used to enforce type compatibility in logical connections among data items in different modules. The type information is also an aid in understanding the module's presentation.
- *access protection:* This protection information, which may be determined by the module at run-time, provides restrictions on who may use a given data item, and how they may use it. Protection is discussed in Section 4.1.

1.1.2 Control

Playground modules are written entirely in terms of the module's local state information, some of which may be published. Since all Playground communication takes place through the presentation, a

Playground module's view of the world is that it may modify its local state, and sometimes data in its local state may change "miraculously" as the result of some activity in the module's environment.

This view of interaction suggests a natural division of the control portion of Playground modules into two components: *active control* and *reactive control*. The active control carries out the ongoing computation of the module, while the reactive control carries out activities in response to input from the environment. For example, in a simulation application, the active control would be responsible for the main loop that performs the computation for each event in the simulation, while the reactive control would handle external changes to published variables representing simulation parameters.

The active control component of a Playground module is the control defined by the *main* function and the functions that it calls. The reactive control component is described on a per data item basis. That is, one associates with certain published variables a function to be performed when a new value is received from an external module. Defining reactive control is discussed in detail in Section 4.5.

1.1.3 Connections

Relationships between the published variables of different modules are established by creating *logical connections* between the variables. The set of connections defines the pattern of communication among the corresponding modules. Connections can be formed:

1. graphically by the end-user, using a "Connection Manager" GUI (see Section 2.2),
2. programmatically within modules (see Chapter 5), or
3. from the "Application Management System," an automated application launcher.

Connections may be either unidirectional or bidirectional. A bidirectional connection might be useful for interactive or collaborative work, while a unidirectional connection with high fan-out would be appropriate for connecting a video source to multiple viewing modules.

The semantics for logical connection communication is FIFO across the connection, meaning that if a published variable has value `a` and later is assigned value `b`, then every module connected from that data item will see `a` before `b`. The default semantics is send-on-update. That is, a value is sent on a logical connection only if the published variable is modified within its module.

The Playground supports two kinds of connections: *simple connections* and *element-to-aggregate connections*. A simple connection specifies communication between published variables of the same type. An element-to-aggregate connection specifies communication between an aggregate of elements of type `T` (e.g., an array of integers) and a variable of type `T` (e.g., an integer). Element-to-aggregate connections facilitate the construction of client-server applications, allowing data from multiple client modules to each contribute individual elements to a server module's aggregate. Element-to-aggregate connections are discussed in more detail in Section 4.5.

1.2 Example Application

Maple syrup is produced by pouring maple sap into a vat and boiling away excess water until a suitable concentration level is reached. A factory producing maple syrup must adjust a number of actuators controlling properties such as the incoming sap flow rate and the burner status (i.e., on or off). The process control application automates maple syrup production, controlling the factory actuators in response to sensor values.

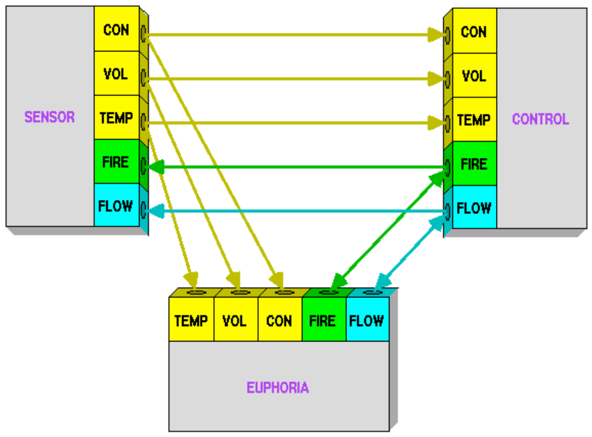

Figure 1: Process control application modules.

The process control application consists of three communicating modules. Using the Connection Manager GUI (see Section 2.2), end-users can view these modules and configure their communication (see Figure 1). Each Playground module is graphically represented as a box with a data “plug” for each published variable. The color of each variable represents its type. Logical connections are represented as arrows between pairs of published variables, and can be created by simply dragging from one published variable to another. The metaphor is that of wiring together the components of a stereo system, where the color of each cord denotes the type of information that it carries.

The SENSOR module monitors conditions of the syrup production: concentration, volume, and temperature. These are published as `CON`, `VOL`, and `TEMP` real number variables, making the values available to other modules. The CONTROL module controls the actuator settings based on these sensor values. It also has `CON`, `VOL`, and `TEMP` published variables that are used to receive updates from the SENSOR module through logical connections. Based on these values, the CONTROL module decides how to adjust the vat’s burner and flow valve (see Figure 2). Two published variables, `FIRE` and `FLOW` (boolean and integer values, respectively), are used to communicate these settings back to the SENSOR module.

Playground also provides a graphical user interface construction module called EUPHORIA. With EUPHORIA, end-users can create interactive graphical displays for distributed applications through the use of a graphics editor. In the process control application, the EUPHORIA module was used to create an interactive graphical display of the maple syrup production (Figure 2). In addition, this display also has published variables for the sensor and actuator values which are associated with graphics objects. Whenever the SENSOR or CONTROL modules change their published variables, these changes are communicated to EUPHORIA according to the established connections and are used to animate the display.

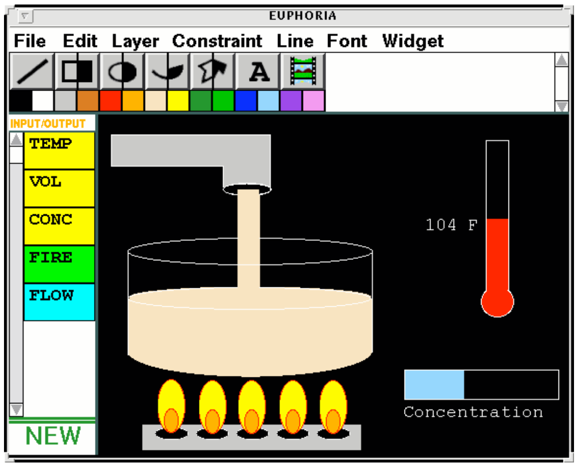

Figure 2: Process control application display.

Note that the `FIRE` and `FLOW` connections between the CONTROL and EUPHORIA modules are bidirectional, allowing user interaction in the display to override decisions of the CONTROL module, in which case the CONTROL module would report the user-specified values to the SENSOR module. For example, the factory operator can adjust the valve of the incoming sap by dragging the width of the sap flow rectangle.

1.3 Overview

The remainder of this manual is organized as follows. Chapter 2 discusses how to start using the Playground environment to develop and use distributed applications. Chapter 3 through Chapter 5 summarizes Playground's C++ library application programmer interface (API) for creating modules, variables, and connections respectively. Chapter 7 outlines a recommended methodology for designing distributed applications with The Programmers' Playground.

Chapter 2

Getting Started

This chapter describes how to start using the Playground tools and compiling modules. These instructions describe how to start applications *manually* (i.e., launching each module from the UNIX shell; graphical configuration). Playground also provides a mechanism for automating application launching.

To use Playground tools, first add Playground to your UNIX path (this only needs to be done once). For Washington University users, execute the following in the UNIX shell:

`pkgaddperm playground` *(Washington University users only)*

Other users, add the following to your `.cshrc` file:

`set path = ($path <pgbin>)` *(Other users)*

where `<pgbin>` is the directory where the Playground executables are located on your file system.

2.1 Parent File

A *parent module* is a module that has information about all of the other modules of an application. When a new module is launched, it needs to be informed of the location of its parents. This is accomplished through a file named **`.pginitrc`**. This file can be created initially as an empty file:

```
touch .pginitrc
```

In general, modules are launched in UNIX by typing the name of the module's executable in the UNIX shell. When a parent module (e.g., the Connection Manager GUI) is launched, it searches for the `.pginitrc` file, first in the current directory then in the user's home directory. If found, the parent module writes its "module ID" to the file for use by future modules. When other modules are launched, they also search for the `.pginitrc` file in the same way. If found, the modules communicate with the parents listed in the file, sending presentation information.

2.2 Connection Manager GUI

At the beginning of each Playground session, you will start up a Connection Manager GUI. The Connection Manager GUI allows you to see modules that you launch and enables you to form connections among published variables. To launch the Connection Manager GUI, type `PGcmgui` in the UNIX shell:

```
PGcmgui
```

An empty window should appear on your screen. It is empty because you have no other Playground modules running. As other modules are launched, they will appear in the window as boxes (see Section 1.2).

To create a logical connection in the Connection Manager GUI, just drag an arrow from the presentation entry of one module to the presentation entry of the other module while holding the *left mouse button*. A bidirectional connection is created the same way, but the *middle mouse button* is held. By default, connections are "send on connect," meaning that upon establishing the connection the current value of the downstream variable is communicated to the upstream variable. Holding the *shift key* while forming a connection inhibits "send on connect."

To delete a connection, click on the connection line with the *right mouse button*. A module may be rotated by clicking on its title.

2.3 Compiling Modules

You must use the CC compiler, version 4.1 or later. Application programs compiled with any other compiler may not link to the Playground library. The line:

```
#include "PG.hh"
```

at the top of every file which uses Playground data types or functions, directs the compiler to include the Playground data types and operations.

A sample `Makefile` is available to automate module compilation.

Chapter 3

Module API

This chapter discusses the Playground `C++` library functions for creating and using a module including initialization, naming, timing, and data synchronization. All module functions discussed in this manual are static members of the `PG` class, and should be preceded by `PG::`.

3.1 Initialization and Termination

Every Playground module should call `PG::initialize` before operating on Playground variables and should call `PG::terminate` upon conclusion:

`void PG::initialize(const char* name);`

`name`	Initial module name *(optional; default = <host>:<process ID>).*

`void PG::terminate();`

Although initialization and termination is done automatically by the Playground run-time system, implementors are encouraged to use `PG::initialize` and `PG::terminate` to make the behavior of the code clear. Below is an example skeleton for a Playground module.

```
#include "PG.hh"

main()
{
  PG::initialize("module name");

  // declarations and active control, including
  // publishing data items and setting up reactive control

  PG::terminate();
}
```

3.2 Synchronization

Synchronization is used to control when updates to published variables will be sent or received. By default, whenever Playground variables are accessed, the Playground system may perform internal operations associated with interprocess communication. These operations can result in communication to external modules (i.e., if a published variable is modified) or changes to the values of a module's published variables (i.e., if updates are received from external modules). At times it may be desirable to temporarily prevent this communication. This can be accomplished by using the functions `PG::shelter` and `PG::beginAtomicStep`:

`void PG::shelter();`

`void PG::unshelter();`

`PG::shelter` shields the entire presentation from external module updates. Incoming values are not seen but updates go out. When complete, updates from external modules resume. Multiple shelters and atomic steps can be nested, the appropriate updates occur when the last `PG::unshelter` or `PG::endAtomicStep` is called. Updates received while sheltered are queued and are used when the shelter is complete.

`void PG::beginAtomicStep();`

`void PG::endAtomicStep();`

`PG::beginAtomicStep` specifies that no updates should be sent or received from the module. When `PG::endAtomicStep` is called, the final values of all modified presentation entries are transmitted as an atomic unit to each connected module. When a group of updates from an atomic step is received by a module, the updates to the presentation and the associated reaction functions are completed for the entire group before the active control is allowed to resume. Thus, the active control perceives the update as an atomic step. Updates received during the atomic step are queued and are used when the atomic step is complete.

Common Bug Associated with Synchronization

`PG::shelter` or `PG::beginAtomicStep` must be matched with a corresponding `PG::unshelter` or `PG::endAtomicStep`. A common mistake is to have a function whose control flow does not always reach the end function. For shelter, this means that updates from other modules are never seen again. For atomic step, this means that updates are neither sent out or received again. For example:

```
void
foo()
{
  PG::beginAtomicStep();

  if (<condition>)
    return;                // <-- bug, exiting will skip PG::endAtomicStep

  // do some more

  PG::endAtomicStep();
}
```

3.3 Explicit Checks for Incoming Data

A module checks for updates to the presentation whenever a published variable is accessed (unless the access occurs after a `PG::beginAtomicStep` or a read occurs after a `PG::shelter`). In addition to this, it is possible to give the module a chance to process updates by using `PG::sleep` or `PG::checkInput`.

`void PG::sleep(unsigned int sec, unsigned int msec);`

`sec`	Seconds to sleep *(optional; default = 1)*.
`msec`	Milliseconds to sleep *(optional; default = 0)*.

`PG::sleep` suspends the module process for a given period of time[1], processes pending changes to a module's presentation, and returns.

☠ On the Solaris operating system, specifying small sleep values (e.g., 1 ms) may cause unpredictable results on heavily loaded systems (e.g., the program never returns from sleep).

`void PG::checkInput(unsigned long timeOut);`

`timeOut`	Maximum wait time, in milliseconds *(optional; default = 0)*.

`PG::checkInput` explicitly checks for and/or processes updates to presentation. If no parameters are supplied, `PG::checkInput` processes all currently pending updates and returns. The `timeOut` parameter specifies the maximum time to wait for an update before returning.

⚠ If a module does not access any of its published variables as part of the active control, it is important to call `PG::checkInput`, `PG::sleep`, or a synchronization method. Otherwise, updates to published variables will not be seen by the program and reactor methods (see Section 4.5) will not be called. Also, if a module needs to wait for some period of time (e.g., wait for a change to published variables) it is a good idea to call `PG::sleep` periodically. Calling `PG::sleep` periodically ensures that the module does not monopolize the CPU, since it releases its cycles during idle times.

3.4 Module Naming

To set the name of a module, use `PG::setName`.

`void PG::setName(const char* name);`

`name`	New name of this module.

1. The sleep operation suspends the process for *at least* the specified time. Due to operating system scheduling, it is likely to actually sleep a little bit longer.

Chapter 4

Variable API

Opening up a Playground module for interaction with other modules is accomplished by declaring Playground variables and publishing them in the presentation of the module. This chapter explains how to declare and use Playground data structures.

The Playground data types are of three categories: *base types* for storing simple data values, *tuples* for storing records with fields, and *aggregates* for organizations of homogeneous collections of elements. Figure 3 shows the class hierarchy of Playground data types and commonly used member functions.

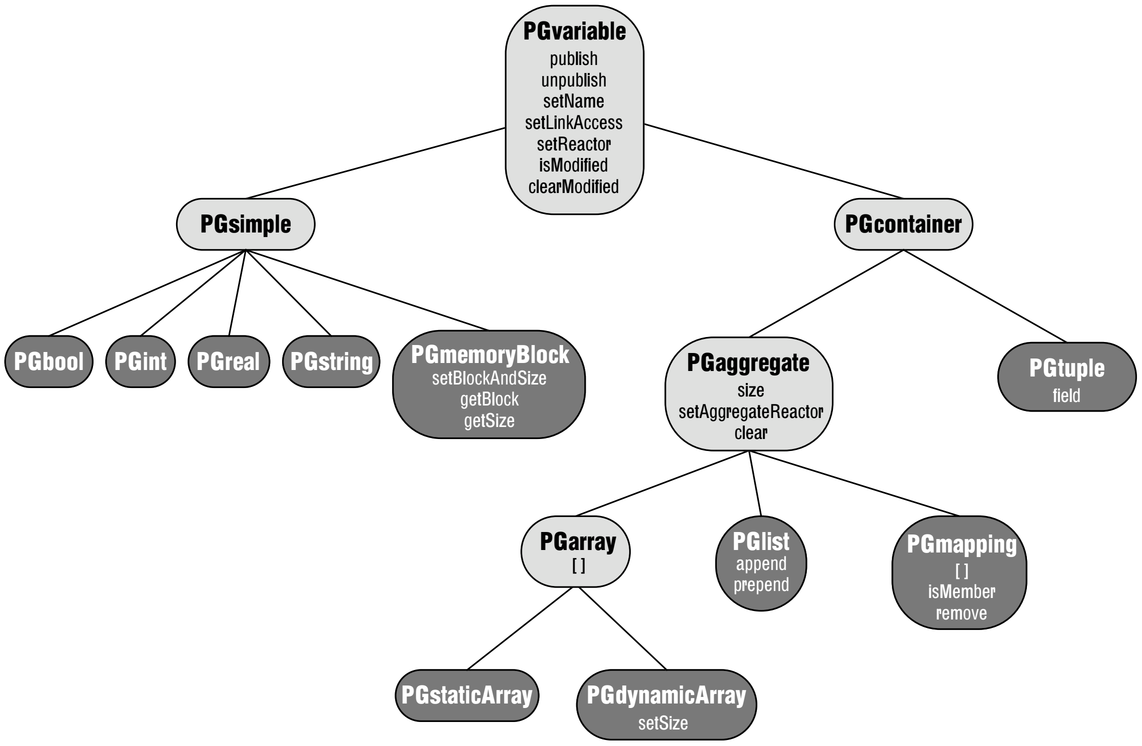

Figure 3: Playground variable class hierarchy and commonly used methods.

4.1 Using Playground Variables

Class `PGvariable` is the base class of all publishable variables; all Playground types inherit the member functions listed in this section. For the following methods, let `v` be a `PGvariable` concrete subclass.

4.1.1 Publishing

Method `publish` publishes a Playground variable in the presentation of a module, exposing it for use by external modules. Method `unpublish` removes the variable from the module's presentation.

`int v.publish();`

`void v.unpublish();`

The return value of `publish` is the variable's presentation index, which can be used in forming connections (see Chapter 5). When `v` is published, it uses either the default values or previously set values for the `v`'s name and access permissions (see Section 4.1.2). The default name of a variable is its type name; the default permission is `PG::READ` (see below). A variable may also be published with these values explicitly specified:

`int v.publish(const char* name, unsigned long access);`

`name`	Public name.
`access`	Read/write permission. <u>Possible values:</u> `PG::READ, PG::WRITE, PG::ELT2AGG,` or combination.

The `name` parameter specifies the public name of the variable. The `access` parameter is used to restrict the types of connections that can be made to the variable. For example, `PG::WRITE` permission means that connections can be directed *to* the variable, allowing external modules to write updates to the variable. Access permissions can be combined with the | operator (e.g., `PG::READ | PG::WRITE`).

In addition, the order of a published variable can be specified relative to other published with `publishBefore` and `publishAfter`.

`int v.publishBefore(PGvariable* var);`

`int v.publishAfter(PGvariable* var);`

`var`	variable in which to publish before/after

4.1.2 Property Accessors

`void v.setPublishedName(const char* name);`

`name`	New public name.

`const char* v.getPublishedName();`

Method `setPublishedName` is used to change a variable's public name.

`void v.setLinkAccess(unsigned long access);`

`access`	New read/write permission. Possible values: `PG::READ, PG::WRITE, PG::ELT2AGG,` or combination.

`unsigned long v.getLinkAccess();`

Method `setLinkAccess` change permission of a variable, as described in Section 4.1.1. A variable's permission may only be changed when it is not published.

4.1.3 Checking for Updates

As described in Chapter 1, there are two approaches for handling external updates to published variables: *active* and *reactive*. Methods `isModified` and `clearModified` provide support to aid in making modules that poll the presentation for external updates more efficient. Method `setReactor` provides support for reactive control.

`bool v.isModified();`

`void v.clearModified();`

When a published variable[1] is modified by an external update, a "modified" bit is set to `true` within the variable. Calling method `isModified` returns the value of this bit, allowing an active control module to periodically poll the variable to check for changes. This approach frees a module's implementor from keeping a copy of the published variables' state in order to determine whether a modification occurred (particularly useful when aggregates are involved). To set this bit to `false`, use method `clearModified`. Method `clearModified` does not clease the modified bit for constituent published variables.

`void v.setReactor(PGreactor* reactor);`

`reactor`	Reactor object (see Section 4.5.1).

`PGreactor* v.getReactor();`

Method `setReactor` sets a reactor object to be used when an update to the variable is received from an external environment. Sending a null pointer as the parameter clears the variable's reactor object. Reactor objects are explained in detail in Section 4.5.1.

4.2 Playground Base Types

A Playground base type can be used wherever a variable of the corresponding C++ data type can be used as an *rvalue*. All Playground base types support operators `=`, `==`, and `!=`. The following sections describe each of the Playground base types.

1. Published variable or any constituent published variables, in the case of tuples or aggregates.

4.2.1 PGint

Objects of type `PGint` carry C++ integer values. All C++ integer operators are supported. Values in C++ integer types may be assigned directly to `PGint` objects, and vice versa. The standard casting rules apply. A `PGint` can be assigned from a `PGreal` or a `double` and so on.

Syntax:

`PGint i;`

`PGint i(long);`

`PGint i(PGint&);`

Example: The following demonstrates using `PGint` with other C data types.

```
void
PGint_example()
{
  PGint x(20);
  PGint y = 10.3;

  x = y - 5;         // x now has a value of 5
  y = 2.5 * x;       // y now has a value of 12
}
```

4.2.2 PGreal

Objects of type `PGreal` carry `C++ double` precision floating point values. All `C++` floating point operations are supported. Values in `C++ double` types may be assigned directly to `PGreal` objects, and vice versa.

Syntax:

```
PGreal r;

PGreal r(double);

PGreal r(PGreal&);
```

Example: The following swaps the values in `x` and `y`:

```
void swap()
{
  PGreal x = 3.7;
  PGreal y(5.2);
  double t = x;
  x = y;           // x is now 5.2
  y = t;           // y is now 3.7
}
```

4.2.3 PGbool

Objects of type `PGbool` carry the boolean values `false` or `true`. An object of type `PGbool` can be used wherever a `bool` can be used. All boolean operators in C++ apply to objects of type `PGbool`.

Values in C++ integer types may be assigned directly to `PGbool` objects. When an integer value 0 is assigned to a `PGbool`, the `PGbool` takes on the value `false`; otherwise, it takes on the value `true`. If a `PGbool` with value `false` is assigned to a C++ integer variable, the value 0 will be assigned; if a `PGbool` with value `true` is assigned to a C++ integer variable, the value 1 will be assigned.

Syntax:

```
PGbool b;

PGbool b(bool);

PGbool b(PGbool&);
```

Example: The following demonstrates assignment between integer and boolean variables.

```
void
PGbool_assign()
{
  PGint i = -2;
  PGbool b(false);
  b = i;            // b now has a value of true
  i = b;            // i now has a value of 1
}
```

4.2.4 PGstring

Objects of type `PGstring` carry string values (i.e., character sequences terminated by a null character). A `PGstring` object does not have a fixed length; its length is dynamically set during copies and external updates.

A `PGstring` may be used where ever type `const char*` is expected (e.g., as the second parameter to `strcpy`). In that case, it is actually casted to a `char*` character sequence. Be careful, if the `PGstring` is null, this casting will return 0.

Syntax:

`PGstring s;`

`PGstring s(const char* s1);`

`PGstring s(PGstring& s1);`

In addition, `PGstring` supports the + operator for concatenation:

`PGstring& s.operator+(PGstring& s1);`

`PGstring& s.operator+(const char* s1);`

Example: The following would swap `PGstring r` and `PGstring s`, using a local character string:

```
#include <string.h>

void
swap()
{
  PGstring r = "This is string 'r'.";
  PGstring s("This is string 's'.");

  char t[30];
  strcpy(t, r);     // strcpy is required since we are copying to a char array
  r = s;            // <- PGstring supports operator =, strcpy is not required here
  s = t;            // ditto
}
```

4.2.5 PGmemoryBlock

An object of type `PGmemoryBlock` is used to store a chunk of memory. `PGmemoryBlock` objects differ from `PGstring` objects in that a `PGmemoryBlock` can contain null characters.

Syntax:

`PGmemoryBlock m;`

`PGmemoryBlock m(size_t size, unsigned char* block=0);`

`PGmemoryBlock m(PGmemoryBlock& m1);`

`PGmemoryBlock` supports the following methods:

`void m.setBlockAndSize(const unsigned char* block, const size_t size);`

`block`	Pointer to the start of a block of memory to be copied.
`size`	Number of bytes in the memory block.

Method `setBlockAndSize` is used to set the value of the memory block. Given `block` and `size`, the specified memory is copied into the `PGmemoryBlock`.

`size_t m.getSize();`

`unsigned char* m.getBlock();`

`void m.updateBlock();`

Method `getSize` returns the number of bytes in the memory block. Method `getBlock` returns a pointer to `PGmemoryBlock`'s data, as set by `setBlockAndSize`. Given the memory block's data, it is possible to perform arbitrary computation on the data without the intervention of the Playground run-time system. Method `updateBlock` is used to inform the Playground run-time system that operations on the memory block's data have finished, and an update of the data should be sent out to connected modules.

Example: In the following example, a memory block is used to store a sequence of 8 bit positive integers. This example normalizes each integer by a scaling factor ($0 \le$ scale ≤ 1), sending the value out to connected modules.

```
void
scaleBlock(PGmemoryBlock& b, double scale)
{
  PG::beginAtomicStep();          // atomic step shields against unexpected updates

  size_t size = b.getSize();
  unsigned char* block = b.getBlock();
  for (size_t i=0; i<size; i++)
    block[i] = (unsigned char)(block[i] * scale);
  b.updateBlock();

  PG::endAtomicStep();
}
```

4.3 Tuples

Tuples provide a mechanism for treating a logically related, fixed sized collection of values (of possibly different data types) as a single unit. A tuple is functionally similar to a C++ `struct`, consisting of a number of *fields*. The data type of a field may be any Playground type, allowing arbitrary nesting (e.g., tuples containing tuple or aggregate fields are permitted).

A tuple is defined by subclassing the type `PGtuple` and calling its `field` method for each tuple field. Let `t` be a subclass of `PGtuple`.

`size_t t.field(PGvariable& pgVar);`

`pgVar`	Playground variable to append to the tuple.

Method `field` appends the given Playground variable to the end of the tuple's list of fields, returning the index of the newly added field. The order and types of fields are important since they are used by the run-time system to determine the type compatibility of two tuples. The `field` method may not be called while the tuple is published.

`PGvariable& operator[](const size_t index);`

index	Index of element to access.

The `[]` operator can be used to access a tuple field given its index.

<u>Example:</u> The following declares a tuple type representing an (x, y) Cartesian coordinate and an example of how it can be used.

```
class Point : public PGtuple {
public:
  Point()
   { field(x); field(y); };                 // add the x and y fields upon creation
  PGreal x, y;
};

Point p;
p.publish("location", PG::READ);
p.x = 5;
((PGreal&)p[1]) = 10;                       // x is index 0, y is index 1
```

When using tuples as a part of some other type (e.g., a `PGarray` of tuples), it is necessary to define a *copy constructor* for the type. If the copy constructor is not defined, it is likely that type incompatibities will occur when making connections. Also, overloading the *assignment operator* can be useful for defining atomic assignment operations. The following example is the `Point` class with these methods.

```
class Point : public PGtuple {
public:
  Point()
   { field(x); field(y); };

  Point(const Point& p)                    // COPY CONSTRUCTOR
   { field(x); field(y); *this = p; };


  Point& operator=(const Point& p)         // ASSIGNMENT OPERATOR
   { PG::beginAtomicStep();
     x = p.x; y = p.y;
     PG::endAtomicStep();                  // atomic step sends x and y together
     return *this; };

  PGreal x, y;
};
```

4.4 Aggregates

Playground provides *aggregates* for building data structures that consist of homogeneous collections called *elements*. The element type of a Playground aggregate must be a Playground type. Currently, array, list, and mapping data types are supported. Each aggregate type allocates and maintains its own element storage; elements inserted into an aggregate are copied rather than used directly. All aggregates support the following methods (let `a` be an aggregate subclass):

`size_t a.size();`

`void a.clear();`

Method `size` returns the number of elements in the aggregate. Method `clear` removes and deallocates all elements of an aggregate (`clear` cannot be used with a `PGstaticArray`).

`void a.setAggregateReactor(PGaggregateReactor* aggReactor);`

`aggReactor`	Aggregate reactor object.

Method `setAggregateReactor` is used to register an aggregate reactor object in order to customize the behavior of element-to-aggregate connections to the aggregate. Sending a null pointer as the parameter clears the aggregate reactor. Aggregate reactors are described in more detail in Section 4.5.2.

⚠ When the values stored in a published aggregate are accessed, either a shelter or atomic step operation (see Section 3.2) should usually be used. Since the number of elements stored within the aggregate can be changed from an external module, a published aggregate may change during traversal if shelter or atomic step is not used. Also, if iterators are used on the aggregate, the iterators may become invalid as a result of an external change.

Using `PG::shelter` ensures that the aggregate does not change while iterating over its values. Using `PG::beginAtomicStep` ensures that all changes to aggregate elements are sent out all at once, which is likely to make communication more efficient.

4.4.1 PGarray

A `PGarray` is a single dimension array of Playground data types. PGarrays can be either *static*, having a fixed size, or *dynamic*, having an adjustable size. The size of static arrays is used for type checking when a connection is requested. Two static arrays cannot be connected if they are of different sizes.

Syntax:

```
PGstaticArray<Element,size_t> a;

PGdynamicArray<Element> a;
```

`PGstaticArray` specifies the array's size as its second template argument. The specified size must be greater than 0. Both static and dynamic arrays support the `[]` operator:

`Element& a.operator[](size_t index);`

`index`	Array index to access.

The `[]` operator is used to access an element of the array, given its index (type `Element` represents the array's element type, as defined by the template). For dynamic arrays, if the supplied index is larger than the size of the array, the array is automatically resized to accommodate the index. For static arrays, supplying an index larger than the array's size generates an exception.

`void a.setSize(size_t size);` *(dynamic arrays only)*

`size`	New size.

The `setSize` method is used to change the size of a dynamic array to a specified size. The size of a dynamic array is not used in determining type compatibility.

Example: The following example reads a series of points from a file and stores them in a dynamic array.

```
#include “PG.hh”
#include <fstream.h>
/* Include the Point tuple example from Section 4.3 */

typedef PGdynamicArray<Point> Points;


void
readFile(Points& points)
{
  PG::beginAtomicStep();

  int n;
  ifstream file(“points.txt”, ios::in);     // open file called “points.txt”
  file >> n;                               // read the first number (# of points)
  points.setSize(n);                          // set the array’s size

  double x, y;
  for (int i=0; i<n; i++) {
    file >> x >> y;                   // read points into temporary variables
    points[i].x = x;                  // <- assign to PGreals
    points[i].y = y;                  // <- ditto
  }

  PG::endAtomicStep();                // send all points together
}


main()
{
  PG::initialize(“Coordinates”);

  Points pts;                              // instantiate array of Points
  pts.publish(“points”, PG::READ);

  readFile(pts);

  while (1)
    PG::sleep(1);

  PG::terminate();
}
```

4.4.2 PGlist

`PGlist` is a singly linked list of homogeneous elements. Used in conjunction with `PGlistIterator`, one can insert, remove, or iterate over list elements.

Syntax:

```
PGlist<Element> lst;

PGlistIterator<Element> it(PGlist<Element> lst);

PGlistIterator<Element> it(PGlistIterator<Element> it2);
```

An iterator may be initialized using either a PGlist or another iterator. If a PGlist is used, the iterator initially points to the first list element. If another iterator is supplied, the iterator is initially points to the same list element as the supplied iterator.

`PGlist` supports the following methods:

```
Element& lst.append(Element el);

Element& lst.prepend(Element el);
```

`el`	Element to add.

Method `append` is used to add an element to the end of the list. The parameter `el` is copied into the newly created last list element. Similarly, method `prepend` is used to add an element to the beginning of the list. A reference to the newly created element is returned.

`PGlistIterator` support the following methods:

```
void it.begin();

void it.next();

bool it.atEnd();
```

Method `begin` moves the iterator to the first list element. Method `next` advances the iterator forward one element from its current position. Method `atEnd` returns true if the iterator has advanced past the last list element or the list is empty.

```
Element& it.getElement();
```

Method `getElement` returns a reference to the list element referred to by the iterator. The `Element` type is defined by the template definition of the list.

```
Element& it.insertBefore(Element el);

Element& it.insertAfter(Element el);
```

`el`	Element to insert.

Methods `insertBefore` and `insertAfter` are used to insert list elements before or after the current iterator position, respectively. The parameter `el` is copied into the newly created last list element. A reference to the newly created element is returned.

`void it.remove();`

The `remove` method removes the current element associated with the iterator. The iterator becomes invalidated once the remove operation is performed.

Example: The following function inserts a string into a list of strings in alphabetical order.

```
#include <string.h>

void
insert(PGstring word, PGlist<PGstring>& list)
{
  PGlistIterator<PGstring> it(list);            // create an iterator
  for (it.begin(); !it.atEnd(); it.next()) {    // iterate over each element
    if (strcmp(word, it.getElement()) < 0) {    // if word comes before current..
      it.insertBefore(word);                    //   insert before current
      return;
    }
  }
  list.append(word);                            // word comes last...
}                                              // put it at the end


PGlist<PGstring> lst;
insert("foo", lst);
insert("bar", lst);
insert("foo", lst);
...
```

4.4.3 PGmapping

A mapping is a one-to-one and onto relation from a *domain* type to a *range* type. That is, for each domain value stored in the mapping, there is an associated range value. Stored range values can be accessed by supplying the associated domain value (there cannot be duplicate domain values). In addition, all domain/range mapping elements can be accessed through the use of an iterator, `PGmappingIterator`.

Syntax:

```
PGmapping<Domain,Range> map;

PGmappingIterator<Domain,Range> it(map);
```

Where `Domain` and `Range` are Playground types. `PGmapping` supports the following operations:

```
Range& map.operator[](Domain domain);

bool map.isMember(Domain domain);

void map.remove(Domain domain);
```

The [] operator is used to access range values given a domain value. This can be used for both storing and retrieving values to/from the mapping. If there is no associated domain/range value, one is created automatically. Method `isMember` returns true if and only if there is an associated domain/range already stored in the mapping. The `remove` method is used to remove a domain value and its associated range value from the mapping.

`PGmappingIterator` supports the following operations:

```
void it.begin();

void it.next();

bool it.atEnd();
```

Method `begin` moves the iterator to the first mapping element. Method `next` advances the iterator forward one element from its current position. Method `atEnd` returns true if and only if the iterator has advanced past the last mapping element.

```
Domain& it.getDomain();

Range& it.getRange();
```

Method `getDomain` returns the domain value of the element associated with the iterator. Method `getRange` returns the range value of the element associated with the iterator.

Example: The following example creates a mapping with domain `PGstring` and range `Stock` (tuple containing two `PGstring` fields, name and price). The mapping is used to store stock information associated with a given stock symbol.

```
// find the associated Stock tuple, create if necessary.
// set the associated stock information.
void
setPrice(PGmapping<PGstring,Stock>& m, PGstring symbol, char* name, double price)
{
  Stock& s = m[symbol];
  s.name = name;
  s.price = price;
}


// use an iterator to print out the stocks.  A shelter is used to ensure
// that the contents of the mapping do not change during iteration.
void
printMap(PGmapping<PGstring,Stock>& m)
{
  PG::shelter();
  PGmappingIterator<PGstring,Stock> iter(m);
  for (iter.begin(); !iter.atEnd(); iter.next()) {
    cout << "D: " << iter.getDomain() << "  R: [";
    Stock& s = iter.getRange();
    cout << s.name << ", " << s.price << "]" << endl;
  }
  PG::unshelter();
}


void
foo()
{
  PGmapping<PGstring,Stock> m;
  m.publish("stocks", PG::READ | PG::WRITE);

  // add some elements to the mapping
  setPrice(m, "SAN", "Sandboxes, Inc.", 37.28);
  setPrice(m, "TRE", "Treehouses, Inc.", 28.43);
  setPrice(m, "SEE", "Seesaws, Inc.", 87.32);
  setPrice(m, "SLI", "Slides, Inc.", 12.64);
  setPrice(m, "SWI", "Swings, Inc.", 67.11);

  cout << "Starting elements:" << endl;
  printMap(m);

  if (m.isMember("SWI"))
    m.remove("SWI");

  cout << "Remaining elements:" << endl;
  printMap(m);
}
```

4.5 Reactor Objects

The purpose of *reactor objects* is to enable the programmer to define an operation (i.e., a method) to be performed when a particular event occurs. The following subsections describe reaction objects that enable reactive control in response to updates (see Section 1.1.2) and element-to-aggregate connections (see Section 1.1.3).

4.5.1 Reacting to Updates

The purpose of reactive control is to allow a module to take a specific course of action whenever a published variable is modified by an external module. This is accomplished by defining a subclass of `PGreactor` and associating it with the variable through the use of `PGvariable`'s `setReactor` method (see Section 4.1.3). Let `r` be a subclass of class `PGreactor`.

`virtual void r.react(PGvariable& reactVar);`

`reactVar`	Published variable that has `r` as its reaction object.

The `react` method is a virtual method of `PGreactor`. It is called whenever the run-time system receives an external update for its associated published variable, supplying the variable as a parameter. Execution of the `react` method is sheltered by the run-time system (see Section 3.2), so other external updates are not processed during `react`'s execution.

In order for reactive control to occur, the program must give the Playground run-time system a chance to run. This is accomplished by either accessing a published variable or calling one of the module functions listed in Section 3.3. Otherwise, reaction methods will never be called.

<u>Example:</u> The following program prints the value of a `PGint` whenever it is modified by an external module. This is achieved by defining a reactor object, associating it with the `PGint`, and entering a reaction loop.

```
#include “PG.hh”

class IntReactor : public PGreactor {
public:
  virtual void react(PGvariable& v) {    // param type is always PGvariable
    PGint& i = (PGint&)v;                // casting is usually necessary...
    cout << “Value: “ << i << endl; };
};

main()
{
  PG::initialize(“Printer”);
  PGint num;
  IntReactor r;
  num.publish(“NUM”, PG::WRITE);
  num.setReactor(&r);

  while (1)                           // infinite loop, calls reaction fn as needed
    PG::sleep(0, 200);                // checks for updates about 5 times per second

  PG::terminate();
}
```

4.5.2 Element-to-Aggregate Connections

Recall from Section 4.4 that an aggregate data type is an organization of a homogeneous collection of elements, such as a set of integers or an array of tuples. The *element type* of an aggregate is the data type of its elements. For example, if `q` is a list of `PGint`, the element type of `q` is `PGint`.

An element-to-aggregate connection results when a connection is formed between a data item of type T and an aggregate data item with element type T. For example, a client/server application could be constructed by having the server publish a data structure of type *list(T)* and having each client publish a data structure of type *T*. If an element-to-aggregate connection is created between each client's type *T* data structure and the server's *list(T)* data structure, then the server program will see a set of client data structures, and each client may interact with the server through its individual data structure.

The creation of an element-to-aggregate connection from `t` to `aggr` causes an element `elt` of the `aggr` to be used as the element associated with the connection and `t`. In other words, it is as if there is a simple connection between `t` and `elt`. When the connection is broken, `elt` can be removed from `aggr`. Like simple connections, element-to-aggregate connections may be unidirectional or bidirectional, and permit arbitrary fan-out and fan-in.

An element-to-aggregate reactor serves as mechanism for controlling how elements are used from an aggregate based on established element-to-aggregate connections. The program is informed when an element-to-aggregate connection is created or removed from an aggregate, giving the programmer the ability to define the relationship between the connection and the aggregate. For instance, if the aggregate is a fixed size array, one might associate each new element-to-aggregate connection with a consecutive element of the array, starting at element 0. If the aggregate is a list, one might wish to *create* a new list element in response to each new element-to-aggregate connection and *destroy* the element when the connection is removed.

Element-to-aggregate reactors are created by subclassing `PGaggregateReactor`, overriding its methods, and associating the reactor with its aggregate through the `setAggregateReactor` method (see Section 4.4). In addition, the aggregate should have the `PG::ELT2AGG` permission set (see Section 4.1) in order to accept element-to-aggregate connections. Let `r` be a subclass of `PGaggregateReactor`.

`virtual PGvariable* r.newConnection(PGaggregate* aggr);`

`aggr`	Aggregate to react upon.

Method `newConnection` is a virtual function of the aggregate reactor that is called when its associated aggregate receives a new element-to-aggregate connection. This method should return a pointer to the element that is to be associated with the connection. For `PGmapping`, `newConnection` should return a pointer to the new domain value.

`virtual void r.removeConnection(PGaggregate* aggr, PGvariable* elt);`

`aggr`	Aggregate to react upon.
`elt`	Element associated with removed connection.

Method `removeConnection` is a virtual function of the aggregate reactor that is called when a connection associated with its aggregate is removed. This method takes as a parameter `elt`, which is the element associated with the removed connection as returned by `newConnection`.

Chapter 5

Connection API

As described in Section 1.1.3, logical connections among published variables can be made in three ways:

1. graphically by the end-user, using a "Connection Manager" GUI,
2. programmatically within modules, or
3. from the "Application Management System," an automated application launcher.

This chapter discusses the second approach: how to form connections programmatically from within modules. Note that this functionality is not required in constructing applications. Indeed, most of the existing applications as of this writing *do not* make connections programmatically. This functionality is provided to support applications that have fairly advanced configuration requirements.

5.1 Connection Requests

A module can make requests to establish or remove connections between the published variables of two modules using `PG::connectRequest` and `PG::disconnectRequest`.

```
void PG::connectRequest(PGmoduleCommID* fromMod, char* fromName,
                        PGmoduleCommID* toMod, char* toName,
                        unsigned long properties);

void PG::disconnectRequest(PGmoduleCommID* fromMod, char* fromName,
                           PGmoduleCommID* toMod, char* toName,
                           unsigned long properties);
```

`fromMod`	ID of upstream module.
`fromName`	Public name of upstream module's variable.
`toMod`	ID of downstream module.
`toName`	Public name of downstream module's variable.
`properties`	Attributes of the connection. Combination of `PG::UNIDIRECTIONAL`, `PG::BIDIRECTIONAL`, `PG::ELEM2AGGREGATE`, `PG::ELEMFROMAGGREGATE` and `PG::SENDONCONNECT`.

Given information about two modules and two published variables, `PG::connectRequest` sends a request to each module to initiate a connection between the variables. Similarly, `PG::disconnectRequest` sends a request to each module to remove a specified connection type between the variables. If a request cannot be sent for some reason, an exception is thrown.

`PGmoduleCommID` contains all of the appropriate information in order to contact a module. It is a subclass of `PGtuple`, and thus can be published. Section 5.3 describes automated mechanisms that allow modules to obtain this information from other modules. In addition, a module can get its own ID through method `PG::getCommunicationID`.

`PGmoduleCommID* PG::getCommunicationID();`

The indices required for a connect/disconnect request represent the presentation indices associated with the published variables. As described in Section 4.1.1, method `publish` returns this value. In addition, Section 5.3 describes how to view the presentation information of external modules.

5.2 Connection Reactor

`PG::connectRequest` and `PG::disconnectRequest` send *requests* for connections to be created or removed. However, this does not guarantee that the connections will actually be created or removed. Modules perform type and access compatibility checking, and may reject a connection request.

When modules complete the processing of a request, a status message is sent back to the module that initiated the request. A *connection reactor* is used to register a method to be performed in response to the status message.

`void PG::setConnectionReactor(ConnectionReactor* reactor);`

`reactor`	Connection reactor object.

`ConnectionReactor* PG::getConnectionReactor();`

`PG::setConnectionReactor` registers a connection reactor object with the Playground run-time system. Sending a null pointer as the parameter clears the run-time system's reactor object. A reactor object is defined by subclassing `ConnectionReactor` and overriding its `reactToStatus` virtual method. Let `r` be a subclass of `ConnectionReactor`.

`virtual void r.reactToStatus(ConnectStatusMessage* message);`

`message`	Connection status message.

Whenever a connection status message is received by a module that has registered a connection reactor, the `reactToStatus` method is called on the reactor object. Let `s` be a `ConnectStatusMessage`.

`int s.getStatus();`

Method `getStatus` returns the status of the connection message. Possible return values are `INITIALIZATION`, `TYPEMISMATCH`, `SUCCESSFULCONNECT`, `SUCCESSFULDISCONNECT`, `MESSAGENOTSENT`, `FROMVARNOTPUBLISHED`, `FROMPERMISSIONDENIED`, `NOFROMINCOMINGLINK`, `NOFROMOUTGOINGLINK`, `TOVARNOTPUBLISHED`, `TOPERMISSIONDENIED`, `NOTOINCOMINGLINK`, `NOTOOUTGOINGLINK`, `NODIRECTIONSPECIFIED`, `INVALIDFROMELEM2AGG`, `NOTRANSPORTSPECIFIED`, or `INVALIDENCODING`.

5.3 Designating a Module as a Parent

A *parent* module is one that automatically receives information about other modules (i.e., "children") when they are started and (possibly) when their presentations change. The Connection Manager GUI (Section 2.2) is one example of a parent; it receives information about other modules and uses this information to create a graphical display.

There are two types of parent modules: *dad* modules and *mom* modules. A dad module receives only basic information about its children. For each child module, the dad module receives the child's module ID. A mom module receives detailed information about its children. For each child module, the mom module receives the child's complete presentation information.

To designate a module as a dad or a mom, use `PG::addDadToStartup` or `PG::addMomToStartup`:

`void PG::addDadToStartup();`

`void PG::addMomToStartup();`

These methods append the parent module's ID to the `.pginitrc` file as either a mom or a dad module, if the `.pginitrc` file is being used (see Chapter 2). When child modules are started, they use the IDs in this file to form element-to-aggregate connections from their presentation information to each of the parents. For this reason, parent modules should publish as their first variable an aggregate to store information about child modules:

- *dad* modules must have an aggregate of `PGmoduleCommID` as the first published variable.
- *mom* modules must have an aggregate of `PGveneerInfo` as the first published variable.

`PGmoduleCommID` is the ID of a module, as described in Section 5.1. `PGveneerInfo` is a `PGtuple` containing complete presentation information about a module. Let `v` be a `PGveneerInfo` tuple:

`const char* v.getModuleName();`

`PGmoduleCommID* v.getCommunicationID();`

Methods `getModuleName` and `getCommunicationID` are used to access the name and ID of a child module given its `PGveneerInfo`.

`size_t v.getNumOfItems();`

`PublishableInfo& v.getItem(size_t nth);`

`nth`	Element number of variable to get (e.g., 0=first, 1=second, ...).

Methods `getNumOfItems` and `getItem` are used to iterate over the list of presentation entries. Information about each published variable is stored in a `PublishableInfo` tuple. Let `p` be a `PublishableInfo` tuple:

`const char* p.getNameMember();`

`const char* p.getTypeMember();`

```
unsigned long p.getAccessMember();
```

Methods `getNameMember`, `getTypeMember`, and `getAccessMember` are used to access the name, type, and access permissions of a published variable (see Section 4.1).

Example Parent Module

The following example shows how a simple server module can be constructed which automatically forms connections to its client modules (client code is not given). The job of the server is to broadcast textual messages received from each client to all clients. Figure 4 shows the relationship between the server module (Mother) and the clients (Child).

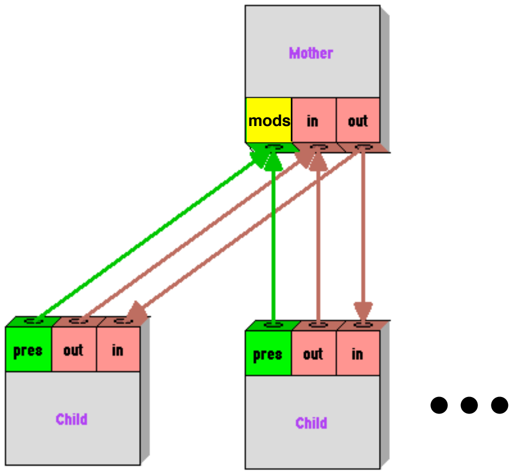

Figure 4: Configured client/server modules.

Presentation information about each module is automatically published by the run-time system. Normally, this information is not shown in the Connection Manager GUI. For illustration purposes, it is shown as variable `pres` in Figure 4's child modules. The following code implements the Mother module. Publishing a `mods` list and calling `PG::addMomToStartup` means that each child will automatically connect from their `pres` variable to the `mods` aggregate (see Section 5.3). The Mother module can use this presentation to request connections between its `in` and `out` variables and each child's.

```
#include "PG.hh"

////////////////////////////////// Children Class ///////////////////////////////////

// This class is both a list of client modules and an aggregate reactor...
// bootstrapping of child modules causes methods of this class to be
// called, resulting in addition/deletion of children
//
class Children : public PGlist<PGveneerInfo>, public PGaggregateReactor {
public:
  virtual PGvariable* newConnection(PGaggregate*);
  virtual void removeConnection(PGaggregate*, PGvariable* elt);
};
```

```
// Add tuple for a new child module, used to store module info.
PGvariable*
Children::newConnection(PGaggregate*)
{
  PGveneerInfo info;
  return &append(info);
}


// Remove information about a child.
void
Children::removeConnection(PGaggregate*, PGvariable* elt)
{
  PGlistIterator<PGveneerInfo> it(*this);
  for (it.begin(); !it.atEnd(); it.next())
    if (elt == &it.getElement())
      it.remove();
}


////////////////////////////////// Mother Class ///////////////////////////////////

// This class is a reactor that manages the module’s published variables.
// It makes connections from its inMsg, outMsg variables to corresponding
// variables in each child.  Through reaction method, it forwards incoming
// messages from each child back out to all children.
//
class Mother : PGreactor {
public:
  Mother();
  virtual void react(PGvariable& v);
  void makeMsgConnections();
private:
  Children kids;                    // list of child presentation info
  PGstring inMsg, outMsg;           // in message and out message strings
};


// publish each variable and set reactors
Mother::Mother()
{
  kids.publish(“mods”, PG::WRITE | PG::ELT2AGG);
  kids.setAggregateReactor(&kids);
  inMsg.publish(“in”, PG::WRITE);
  inMsg.setReactor(this);
  outMsg.publish(“out”, PG::READ);
}


// when a new value of inMsg is received, this function is called.
// assigning to outMsg broadcasts the message out to all connected modules.
void
Mother::react(PGvariable&)
{
  outMsg = inMsg;
}
```

```
// iterate over all child modules, making message connections to each
// module named “Child”
void
Mother::makeMsgConnections()
{
  PG::beginAtomicStep();

  PGmoduleCommID* myID = PG::getCommunicationID();
  PGlistIterator<PGveneerInfo> it(kids);
  for (it.begin(); !it.atEnd(); it.next()) {
    PGveneerInfo& info = it.getElement();
    PGmoduleCommID* otherID = info.getCommunicationID();
    if (strcmp(“Child”, info.getModuleName()) == 0) {
      PG::connectionRequest(otherID, “out”, myID, “in”, PG::UNIDIRECTIONAL);
      PG::connectionRequest(myID, “out”, otherID, “in”, PG::UNIDIRECTIONAL);
    }
  }

  PG::endAtomicStep();
}


main()
{
  PG::initialize(“Mother”);

  Mother mom;                    // create mom object, with associated variables
  PG::addMomToStartup();         // register mom so that others will connect
  PG::sleep(5);                  // wait for child modules
  mom.makeMsgConnections();      // make connections to all children

  while (1)                      // reactive loop, forward messages of children
    PG::sleep(0, 200);

  PG::terminate();
}
```

Chapter 6

Types API

Instead of statically defining the type of a Playground variable using one of the types described in Chapter 4, there is the possibility of dynamically building a type and instantiating variables from that type.

As of this writing, the types API described in this chapter are not supported in the current release. This functionality will be released with future versions.

6.1 Specifying Types

Class `PGtype` is the container for the type information, which is kept as an internal structure of nodes derived from the class `PGtypeNode`. These nodes are never created directly but can be accessed for extracting appropriate information (See "Accessing Node Attributes" on page 37.).

6.1.1 Basic Type Creation

`PGtype` contains a set of static methods to create a type corresponding to the different Playground static types. These methods are:

Basic Types:

```
PGtypeNode * aPGint();

PGtypeNode * aPGreal();

PGtypeNode * aPGbool();

PGtypeNode * aPGstring();

PGtypeNode * aPGmemoryBlock();
```

Aggregates:

`PGtypeNode * aPGXstaticArray(size_t arraySize,`
`                              PGtypeNode * element);`

`arraySize`	size of the array
`element`	type of the elements of the array

`PGtypeNode * aPGXdynamicArray(PGtypeNode * element);`

`element`	type of the elements of the array

`PGtypeNode * aPGXlist(PGtypeNode * element);`

`element`	type of the elements of the list

`PGtypeNode * aPGXmapping(PGtypeNode * domain,`
`                          PGtypeNode * range);`

`domain`	type of the domain values
`range`	type of the range values

For the aggregate types, the type arguments are specified using one of the methods described above, from another `PGtype` (which will be automatically converted to a `PGtypeNode *`), or from a Playground variable (See "Accessing the Type of a Variable" on page 40.) The following shows examples of type creation:

```
PGtype t1 = PGtype::aPGint();
PGtype t2 = PGtype::aPGXlist(PGtype::aPGint());
PGtype t3 = PGtype::aPGXlist(t1); // i.e., a PGlist of PGint, as is t2
PGtype t4 = t3;
```

6.1.2 Creating Tuple Types

Creating a type representing or containing a tuple can be performed using a combination of two methods: directly by specifying the types of the fields, or indirectly by adding field types to an existing tuple type. Two static methods similar to the ones described in the previous section are available to create a tuple type:

`PGtypeNode * aPGtuple();`

`PGtypeNode * aPGtuple(size_t nbFields, ...);`

`nbFields`	number of field information specified after that argument
`...`	'*nbFields*' pairs of field information, each consisting of a (possibly `NULL`) `const char *` specifying the name of the field, and a `PGtypeNode *` indicating the type of the field

Furthermore, given a `PGtype` `t` representing a tuple, one can add a field to `t` by using

```
PGtypeNode * t.addField(const char * fieldName
                         PGtypeNode * fieldType);
```

`fieldName`	(possibly `NULL`) name of the field
`fieldType`	type of the field

Here several possible methods to create a tuple with two `PGint` fields named ‘x’ and ‘y’, and a third unnamed `PGreal` field:

```
PGtype i = PGtype::aPGint();
PGtype r = PGtype::aPGreal();

// specify all at once
PGtype tuple1 = PGtype::aPGtuple(3,“x”,i,”y”,i,NULL,r);

// add one field at a time
PGtype tuple2 = PGtype::aPGtuple();
tuple2.addField(“x”,i);
tuple2.addField(“y”,i);
tuple2.addField(NULL,r);

// chain the field additions
PGtype tuple3
  = ((PGtype::aPGtuple()->addField(“x”,i))->addField(“y”,i))->addField(NULL,r)

// hybrid creation
PGtype tuple4 = PGtype::aPGtuple(2,”x”,i,”y”,i);
tuple4.addField(NULL,r);
```

A tuple type can be extended as long as no Playground variable has been instantiated from that type (See “Creating a Variable from a PGtype” on page 40.) Once a variable has been instantiated, one must clone the type (using the assignment operator or copy constructor for `PGtype`) to be able to modify the clone. Trying to modify a type that has been instantiated will throw an exception. The method

```
bool t.canBeModified() const;
```

will indicate if a tuple type `t` can be safely modified.

6.1.3 Accessing Node Attributes

While the actual structure of a type is not directly available, it is possible to extract some information from the nodes composing the structure. Each node is provided with a identifier indicating the kind of the node, which can be accessed using

```
PGtype::Types_t t.getType_t() const;
```

where `PGtype::Types_t` is an `enum` containing the following values:

```
PGbool_t, PGint_t, PGreal_t, PGstring_t, PGmemoryBlock_t, PGtuple_t,
PGXstaticArray_t, PGXdynamicArray_t, PGXlist_t, PGXmapping_t
```

There is a common attribute-accessing interface for all node types, also some methods are meaningful only for specific kinds of nodes. Trying to access attributes which are not meaningful to a given node type will return a null value (`NULL` pointer or 0). The interface is the following

For tuple nodes

```
size_t nbFields() const;

PGtypeNode * getFieldType(size_t fieldIndex) const;

const char * getFieldName(size_t fieldIndex) const;
```

For array and list nodes

```
PGtypeNode * getComponentType() const;
```

For static array nodes

```
size_t getArraySize() const;
```

For mapping nodes

```
PGtypeNode * getDomainType() const;

PGtypeNode * getRangeType() const;
```

6.2 Instantiating Variables

Chapter 4 describes the high-level interface to create Playground variables. This section presents the underlying mechanisms: generic aggregates and instantiating from a `PGtype` structure.

6.2.1 “Non-Templatized” Aggregate Variables

The set of template classes described in Section 4.4 are wrappers around an equivalent set of “generic” classes for which the types of the elements are specified using `PGtype` information, instead of actual Playground types. All generic aggregate class names start with ‘`PGX`’ instead of ‘`PG`’.

6.2.1.1 Creating Generic Aggregates

While the templatized aggregate classes get all the necessary information from the template structure, generic aggregates need to be provided with appropriate information whenever an instance is created. Thus, the constructors for the different generic aggregates are:

```
PGXstaticArray(size_t size, const PGtype & elementType);
```

`size`	number of elements in the array
`elementType`	type of the elements

`PGXdynamicArray(const PGtype & elementType);`

`elementType`	type of the elements of the array

`PGXlist(const PGtype & elementType)`

`elementType`	type of the elements of the list

`PGXmapping(const PGtype & domainType, const PGtype & rangeType);`

`domainType`	type of the domain values
`rangeType`	type of the range values

Generic iterators are also provided for lists and mappings:

`PGXlistIterator(PGXlist & list);`

`PGXlistCyclicIterator(PGXlist & list);`

`list`	the list to iterate through

`PGXmappingIterator(PGXmapping & mapping);`

`mapping`	the mapping to iterate through

The template aggregates are actually derived from their respective generic versions. Thus one can perfectly use a `PGXlistIterator` to iterate through a `PGlist<...>`.

6.2.1.2 Accessing Generic Aggregates

The generic aggregate classes provide the same set of methods as their templatized versions. But because generic aggregate classes do not have any type information until created, the return values which were typed based on the template information are replaced by `PGvariable &` types, as follows:

<u>`PGXstaticArray` and `PGXdynamicArray`:</u>

`PGvariable & operator [] (size_t index);`

<u>`PGXmapping`</u>:

`PGvariable & operator [] (PGvariable &);`

<u>`PGXlistIterator`, `PGXlistCyclicIterator`, and `PGXmappingIterator`</u>:

`PGvariable & getElement();`

Thus casting is always required to manipulate the values returned from these methods:

```
PGXstaticArray array(4,PGtype::aPGint());
for (size_t i=0;i<4;i++) {
  (PGint &)array[i] = 2*i;
}
```

6.2.2 Creating a Variable from a `PGtype`

One can create instances of `PGvariable` based on a `PGtype` structure `t` using:

```
PGvariable * t.createInstance();
```

The resulting variable is created using an appropriate combination of basic Playground types, `PGtuple`, and generic aggregates.

```
PGtype type = PGtype::aPGXstaticArray(4,PGtype::aPGint());
PGvariable * var = type.createInstance();
PGXarray & array = *var;
for (size_t i = 0; i<4; i++) {
  (PGint &)array[i] = 2*i;
}
```

6.2.3 Accessing the Type of a Variable

Every `PGvariable` includes a representation of their `PGtype` structure, which can be accessed using

```
PGtype & v.getType() const;
```

Note: using a copy constructor for one of the Playground types will result in a "clone" variable being created, with a duplicate of the original variable's value, as in

```
PGlist<PGint> list1;
// setting the content of list1
PGlist<PGint> list2(list1); // list2 now has a duplicate of list1's content
```

On the other hand, one can simply clone the variable structure without duplicating the original's content using the `PGtype` of the original variable:

```
PGlist<PGint> list1;
// setting the content of list1;
PGXlist * list2 = (PGXlist *)(list1.getType().createInstance());
// list2 is now an empty list of PGint
```

Warning: In the case of tuple classes derived from `PGtuple`, "cloning" a variable will result in a `PGtuple` with the same set of internal fields, but **without** any of the additional data members that are part of the derived types:

```
class PGpoint : public PGtuple {
public:
  PGpoint() : { field(x); field(y); }
  PGint x, y;
};

PGpoint p;
PGtuple * p2 = (PGtuple *)(p.getType().createInstance());
(*p2)[0] = 3; // OK, assign to 1st field
PGpoint * p3 = (PGpoint *)(p.getType().createInstance()); // DANGEROUS
p3->x = 3; // WRONG, 'p3' doesn't point to a structure containing storage for 'x'
(*p3)[0] = 3; // OK, assign to 1st field
```

Chapter 7

Application Design Methodology

This chapter discusses a suggested approach for designing modules from scratch and upgrading existing C++ programs to be Playground modules.

7.1 Behaviors and the Environment

It is helpful to think about a Playground module as interacting with an *environment*, an external collection of modules that may observe and modify the data items in its presentation (as permitted by the access protection defined for the data items). Sometimes, the data items in a presentation fall naturally into *input data items* written only by the environment and *output data items* written only by the module. In other cases, it is more natural to allow both the environment and the module to write to the same data item.

A *behavior* of a module is a sequence of values held by the data items in its presentation. One should think about the behavior as the view that the environment has of the module. Therefore, when designing a Playground module, it is helpful to first write down a *behavioral specification* of the module. Such a specification would include the set of data items that should appear in the presentation, as well as a description of the allowable behaviors that may be exhibited by the module.

The notion of behavior is symmetric. That is, the behavior of an module is not only the view that the environment has of the module, but it is also the view that the module has of the environment. In fact, as part of the behavioral specification, it is useful to state any assumptions being made about the allowable behaviors of the environment. Behavioral specifications are part of the Playground application design methodology presented in the next section. Dividing the presentation into input data items and output data items can help simplify the task of constructing a behavioral specification. Such a division can be enforced by assigning the appropriate protection information to the data items.

7.2 Designing Playground Modules from Scratch

This section contains a step-by-step recommended approach to constructing Playground applications. Beginners may prefer to work based on the examples provided and not following a formal methodology as given below, but it is recommended that experienced Playground programmers use such a methodology especially for non-trivial projects. As you develop experience with Playground, you will, no doubt, develop your own methodology that may differ from the one presented here. The guide is written from the viewpoint of designing Playground applications from scratch. However, it is followed by some hints on modifying existing C++ applications to interact in the Playground environment.

1. Write down a *behavioral specification* of the module. This will include the following three parts.
   - The set of data items to appear in the presentation, with annotations for whether they are externally readable, writable, or both.
   - A set of restrictions on how the environment is allowed to interact with your module. For example, if you publish an integer variable that is writable by the environment, you may want to require that the environment writes only positive integers into that variable. If you want the module to be very robust, you may specify that the environment be allowed to do anything it wishes.
   - The set of allowable behaviors that may be exhibited by the module, as long as the environment respects the conditions given in above. You are not likely to list all of the allowable behaviors (there may be infinitely many of them!) Instead, you should characterize the set by stating a set of necessary conditions on behaviors that, when met, imply that the sequence is legal. An important kind of condition is an *invariant assertion*, a predicate on data items that must remain true at all times during execution.
2. Identify which activities of the module will be under active control, and which will be under reactive control. As a rule of thumb, ongoing activities will be part of the active control, while those that occur in response to an external change to a published data item will be part of the reactive control. A good starting point is to consider each externally writable data item as a potential point of reactive control.
3. Define the local (internal) state of the module.
4. For each reactive control activity, design a *reaction method* to be associated with the corresponding data item. This function may modify the local state and/or data items in the presentation. Verify that each reaction function cannot violate any conditions in the behavioral specification (provided that the environment meets its obligations).
5. For the active control, there are typically three parts: an initialization phase (that includes some necessary Playground initialization and typically publishes data items in the presentation), one or more loops that control the ongoing part of the computation, and a termination phase. Verify that all three phases are consistent with the behavioral specification.
6. Write the module based on the above design. Once the presentation, local state, and active and reactive control have been identified, writing the code for the module should be a standard programming task.

7.3 Upgrading Existing Modules for Playground

The essential difference between an ordinary C++ application and a Playground application is the way that I/O is accomplished. Normally, the reason for upgrading an existing application to work in Playground is so that it may be configured to work interactively with other Playground applications, or so that a new user interface can be constructed for the application using EUPHORIA.

To upgrade an existing module, you should begin by identifying the data structures of the module that are of external interest, and by identifying the parameters or input data that direct the computation. The former are prime candidates for externally readable data items in the presentation, and the latter are likely to be externally writable data items. Writing a behavioral specification is likely to be helpful for upgraded modules as well as for new ones.

Note that it is not a requirement that all I/O take place through the presentation in an upgraded module. For example, if you want to retain your old user interface, but allow the "back-end" to communicate with other Playground modules, it may not be necessary to publish the user-controllable data in the presentation. Alternatively, you may want to retain the same "back-end" but redefine the user interface.

Once the presentation has been determined, the data types for those data structures must be redefined in terms of the provided Playground data types. Following this, it is a simple matter to publish these data structures and make them available for interaction with the environment.

Index