Report Number: WUCS-95-20

1995-01-01

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

Kenneth J. Goldman, T. Paul McCartney, Ram Sethuraman, and Bala Swaminathan and Todd Rogers

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

Kenneth J. Goldman, T. Paul McCartney,
Ram Sethuraman, Bala Swaminathan, and
Todd Rodgers

WUCS-95-20

July 1995

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

Contents

Preface

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 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.*

You do not need to read this entire manual in order to begin. The Programmers' Playground and this manual are designed to enable effective use of the Playground software without a thorough understanding of all of its features. It is important to read Chapter 1 to gain a basic understanding of the fundamental concepts of the Playground methodology. You should also read Chapter 2 to learn the mechanics of setting up a Playground application. More information and live demonstrations are available on the Playground world wide web site [2].

Welcome to The Programmers' Playground!

Chapter 1

The Playground Philosophy

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 the 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.

The Programmers' Playground is neither a new programming language nor a new operating system. It is a way of thinking about distributed applications and a set of software tools to support that way of thinking. This chapter describes the Playground philosophy. Later chapters describe the software tools that enable you to write applications in C++ using this philosophy. As additions are made to the suite of supported languages, your C++ applications will be able to communicate with applications written in other languages. In addition to the tools described in this manual, Playground also provides EUPHORIA, a user interface management system for end-user construction of direct-manipulation user interfaces (for more information, see the *EUPHORIA Reference Manual* [4].)

1.1 I/O abstraction

The Programmers' Playground is the embodiment of *I/O abstraction.*

Briefly, I/O abstraction is the view that each *module* in a system has a set of data structures that may be externally observed and/or manipulated. This set of data structures forms the external interface (or *presentation*) of the module. 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 structures. As published data structures are updated, I/O occurs implicitly ("under the covers") according to the logical connections created in the configuration.

I/O abstraction 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.

Declaring high-level relationships among the state components of software modules makes implicit communication possible. Once the high-level relationships between state components are declared, if a particular state change in one module should be reflected in the state of another module, then this can be recognized by the system and the necessary communication can be handled under the covers. Thus, output is essentially a *byproduct* of computation, and input is handled *passively*, treated as a *modifier* (or sometimes an *instigator*) of computation.

The result is a separation of computation from communication. Rather than writing applications programs that are peppered with explicit I/O requests, the applications programmer is concerned only with the details of the computation, and the communication is declared separately in terms of high-level relationships among the state components of different modules.

1.2 Basic Concepts

The Programmers' Playground implements I/O abstraction in terms of three basic concepts: data, control, and connections.

1.2.1 Data

Data are the units of a module's state. Data may be private to a module or they may be *published*, meaning that other modules in the system may access the data. Playground provides a library of data types that may be published. These *Playground data types* are divided into three categories: *base types* for storing integer, real, boolean, character, and string values, *tuples* for storing records with various *fields*, and *aggregates* for organizations of homogeneous collections of *elements*. Certain aggregate data types (such as mappings and arrays) are provided with the Playground implementation, and the applications programmer may define others. The fields of tuples and the elements of aggregates may be nested arbitrarily using the Playground data types.

1.2.1.1 The Presentation

Each Playground module has a *presentation* that consists of the set of data items that have been published by that application. Each published data item has certain associated information:

public name A descriptive textual name assigned by the application program when the data item is published. The name helps users of the application 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 detail in Chapter 3.

1.2.1.2 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 application. In other cases, it is more natural to allow both the environment and the application to write to the same data item.

A *behavior* of an application 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 application. Therefore, when designing a Playground application, it is helpful to first write down a *behavioral specification* of the application. 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 application.

The notion of behavior is symmetric. That is, the behavior of an application is not only the view that the environment has of the application, but it is also the view that the application 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 Section 1.4. 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.

1.2.2 Control

The *control* portion of an application defines its behavior, the sequence of changes it makes to its state. Therefore, it is really the control that determines how the application interacts with its environment.

Since an application knows nothing about the structure of its environment, Playground applications are written entirely in terms of the application's local state information, some of which may be published. Since all Playground communication takes place through the presentation, a Playground application'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 application'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 application, 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 changes in externally controllable parameters of the simulation.

Not all modules have both active and reactive control. Modules with only active control can be quite elegant, since input to the application simply modifies the course of the computation without the need for

any special activity when the input occurs. For example, one could imagine a simulation in which changing some parameters would alter the course of future iterations, but the act of changing the parameter would not require any special activity. This style of handling input is called *passive observation.*

Some modules may be structured with only reactive control. This kind of application has no autonomous activity to perform. Instead it simply reacts to each input from the environment, possibly updating values of the data items in its presentation when external changes occur. For example, a data visualization application might be constructed so that each time some data element changes, the corresponding piece of the display is updated. This style of programming is called *interrupt driven.*

Typically, however, modules have both active and reactive control, although one may dominate. For example, one might design a module that for the most part uses passive observation, but must check some conditions (such as consistency violations or termination) when certain published data items are changed externally.

The active control component of a Playground module is the control defined by the `main` function of the application. The reactive control component is described on a per data item basis. That is, one associates with each data item an activity to be performed when the value of that data item is changed by the environment.

Defining active and reactive control in Playground applications is discussed in detail in Chapter 4.

1.2.3 Connections

Communication between a module and its environment cannot take place until the environment is defined. Since the environment is really nothing more than a collection of (one or more) other Playground modules, creating an environment really amounts to establishing relationships among the data items in the presentations of different modules.

Relationships between the data items in the presentation of different modules are established by creating *connections* between those data items.[1] The set of connections among published data items defines the pattern of communication among the corresponding applications. Connections are created by a special Playground module called the *connection manager.* Described in detail in Chapter 5, the connection manager enforces type compatibility among the data items involved in a connection and also guards against protection violations by creating connections only if they are compatible with the access protection defined for the relevant data items.

The same logical connection paradigm is designed for both discrete data (such as sets of integers) and continuous data (such as audio and video). Any differences in the physical communication requirements of different connections are inferred from the data type information.

Connections are created separately from the definitions of modules. This means Playground modules are designed in isolation, and then later connected together. In this way, each module need not be concerned with the structure of its environment, only with the behaviors exhibited in its presentation.

The semantics for logical connections is FIFO across the connections, meaning that if a presentation item has value a then value b, every connection from that data item will see a before b. The default semantics is send-on-update, that is send out a value on a logical connection only if the presentation item is written into.

The Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections, defined below. In general, a given data item may be involved in multiple connections of different kinds.

[1] Connections are sometimes called *logical connections* to contrast them with *physical connections* such as links in a computer network.

1.2.3.1 Simple Connections

A simple connection relates two data items of the same type. For example, an integer x in application A might be connected with a simple connection to integer y in application B. If the simple connection is unidirectional, then the semantics of the connection is that whenever A changes the value of x, then item y in application B is updated with that value as well. If the simple connection is bidirectional, then an update of y 's value by application B would also result in a corresponding update to x in A.

Arbitrary *fan-out* and *fan-in* are permitted, meaning that multiple simple connections may emanate from or converge to a given data item. For example, the integer x in the above example might also be connected to integer z in application C. Then, whenever the value of x is changed, y and z are both updated. Examples of arbitrary fan-in and fan-out are shown in Figures 5.3 and 5.4 in Chapter 5.

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 applications.

1.2.3.2 Element-to-Aggregate Connections

Recall from Section 1.2.1 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 s is a set of integers, the element type of s is integer.

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 *set(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 *set(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 x to s causes a new element to be created in the aggregate s and all interaction for that connection takes place through that distinguished element and x. In other words, it is as if there is a simple connection between x and the distinguished element of s. When the connection is broken, the distinguished element is removed. Like simple connections, these may be unidirectional or bidirectional, and permit arbitrary fan-out and fan-in. In the client/server example, arbitrarily many clients could be handled by multiple element-to-aggregate connections to the same aggregate, each with its own distinguished element.

1.3 Implementation Overview

Understanding the basic organization of the Playground implementation can be helpful in learning how to write Playground applications. The Programmers' Playground is part of an ongoing research project based on the I/O abstraction concept. Therefore, this section describes the "current implementation status." Work on providing I/O abstraction for multimedia applications on ATM networks is in progress.

Since one of the goals of this work is to support heterogeneous distributed computing, The Programmers' Playground is implemented not as a new programming language but instead as a software library and run-time system that insulates the applications programmer from the operating system and the network. The version of the system described here supports applications written in C++ on top of the Solaris(UNIX) operating system with sockets as the underlying communication mechanism.

A *module* is the basic unit of computation in Playground. Each module is written in C++ using I/O abstraction, as described in the previous section. The application publishes a set of data structures as its presentation. The presentation is managed by the *veneer,* a software layer between the application and

the Playground communication protocol. The veneer defines the Playground data types and maintains the documentation and protection information published with each data structure. Reactive control information is also registered in the veneer. The protocol is responsible for communication between the module and other Playground modules according to the logical connections.

1.4 Application Design Methodology

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.

1.4.1 Designing Playground Modules from Scratch

1. Write down a *behavioral specification* of the module. This will include the following three parts.
   (a) The set of data items to appear in the presentation, with annotations for whether they are externally readable, writable, or both.
   (b) 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.
   (c) The set of allowable behaviors that may be exhibited by the module, as long as the environment respects the conditions given in (b). 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 function* 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.

1.4.2 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 the Playground user interface management system.

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 Playground data types provided in the veneer. Following this, it is a simple matter to publish these data structures and make them available for interaction with the environment.

1.4.3 Discussion

In the Playground methodology described above, concepts like "sockets," "message passing," and "remote procedure calls" are conspicuously absent. This results from the separation of computation from communication that is a fundamental principle of the I/O abstraction concept. With I/O abstraction each program's computation is expressed in terms of its local data structures, some of which may be published. The published data structures form a well-defined data interface with an abstract environment that is allowed to observe and modify those data structures. Logical connections between the published data structures are configured separately and may be changed dynamically. The application need not be concerned with explicitly sending data to and receiving data from other modules, and need not be concerned with coordinating its activities with specific processes. Access protection is provided so that changes occur only to those published data structures that are expected to change.

I/O abstraction exploits high speed network technology in two ways. First, it recognizes the fact that communication is becoming much less expensive, and so the programmer no longer needs such tight low-level control on the exchange of data between modules. This low-level control is relinquished in favor of a high-level communication abstraction that makes use of efficient "behind the scenes" data transmission, where the granularity of transmission is flexible. Second, it takes advantage of high-speed networks by accommodating both discrete and continuous data types within the same high-level configuration mechanism.

An important benefit of I/O abstraction is the potential for integrating discrete data and continuous data within one communication model. The testbed for this work will be the high speed packet-switched network that is being deployed on the Washington University campus. The network, called *Zeus*, is based on fast packet switching technology that has been developed at Washington University over the past several years and is designed to support port interfaces at up to 2.4 Gb/s. The Zeus network will support the

implementation of multimedia Playground applications that communicate using real-time digital video and audio, as well as symbolic data. [1]

I/O abstraction encourages the use of proven software engineering techniques. Since each Playground module is written as an encapsulated unit, it is possible to reason about each module locally in terms of its well-defined data interface. Since I/O abstraction makes portions of the state of a computation externally available and hides the structure of the environment from the application, it is possible to build a user interface for an application in a way that is completely decoupled from the application itself. Furthermore, like other coordination languages, I/O abstraction encourages a certain level of portability. An application written using I/O abstraction in a given programming language can be readily ported to another platform provided that the veneer and protocol supporting I/O abstraction exist for the target platform.

A discussion of the relationship between I/O abstraction and other communication models is contained elsewhere [3].

Chapter 2

Getting Started

2.1 Setting up Playground

To run Playground you will need a file called `.pginitrc` in your home directory containing the two lines:

PGLMHOST *hostname*
PGLMPORT *portnumber*

where *hostname* is the name of the machine on which you will run the connection manager and *portnumber* is a number that you pick, in the range 10,000 to 30,000. This is the port number that the connection manager will use to communicate with other Playground modules. If you pick a number that is used by another connection manager running on your machine, then you will have a conflict and should change the number. The port number should also be changed when a module terminates abnormally.

In the future, if you need to change the location where you are running the connection manager, or if you need to change the port id for the connection manager, you can use the command

setpg *portnumber hostname*

to change the corresponding values in your `.pginitrc` file. The *hostname* can be omitted if only the *portnumber* is to be changed.

Starting the Connection Manager: At the beginning of each Playground session, you will start up a connection manager, `PGcm` for your own private use, on the host named in your `.pginitrc` file. Most likely, you will also want to start up a connection manager user interface, `PGcmgui`, not necessarily on the same host. See Chapter 5 for an explanation of the connection manager.

An empty window should appear on your screen. It is empty because you have no other Playground modules running. If it's not empty, the number you picked earlier probably was not unique – this is a feature allowing multiple users to share a connection manager, but if this happens unintentionally, you should use **setpg** to pick another number and try again.

To create a logical connection in the graphical front-end to the connection manager, 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. To delete a connection, click on the connection line with the right mouse button. See Chapter 5 for more information on connections.

2.2 Starting to write Playground Modules

This section explains the basic housekeeping that must be done to initialize and terminate a Playground application.

Below is an example skeleton for a Playground application.

Example:

```
#include "PG.hh"

// declarations for global Playground objects

main() {
   PGinitialize("module name");

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

   PGterminate();
}
```

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 defined in the veneer.

The file containing the `main` function provides, using the function `PGinitialize`, an external name for the application for configuration purposes. The application above has the name *module name.*

The `main` function begins with `PGinitialize` and ends with `PGterminate`. `PGterminate` destroys the protocol process and therefore should be called only after the application's computation is complete. Note that while all files using Playground data types must include `PG.hh` , only the file containing `main` should call `PGinitialize`.

We include an annotated Playground program just to give you a feel for what it looks like.

Example: This module has a single read-write integer presentation It increments the value once per second until the counter hits 1800. Therefore, it will run for about 30 minutes starting from 0.

```
#include "PG.hh"

int
main()
{
  PGint c = 0;                  // declaration of Playground integer

  PGinitialize("COUNTER");      // Start protocol process and assign name ''COUNTER'' to module
  PGpublish(c,"num",RW_WORLD); // Publish variable c, that is make it available in
                                // the presentation . Variable c has name ''num'' in the
                                // presentation . RW_WORLD means that the environment can
                                // read and write c.

  while (c < 1800) {
```

```
    PGsleep(1);                 // delay execution for 1 second
    c = c + 1;
  }

  PGterminate();                // clean up
}
```

Chapter 3

Creating a Playground Presentation

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

Data may either be private to a module or may be *published.* Other modules in the environment can access a published variable subject to access restrictions enforced by the system. Only the Playground data types can be published. 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. Certain types are provided in the Playground library and the applications programmer can define others as needed using the provided types.

3.1 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.*

3.1.1 PGint

Objects of type `PGint` carry C++ integer values.

Sample Declaration: `PGint i;`

Supported Operators: All C++ integer operators are supported.

Example: Values in C++ integer types may be assigned directly to `PGint` objects, and vice versa. The standard casting rules apply. A `PGint` can be cast to a `PGreal` or a `double` and so on.

```
void PGint_example()
{
  PGint x,y;
  int t;
  double d;

  x = 20;
  y = 10.3;
  t = 5;
  d = 2.5;
```

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

3.1.2 PGreal

Objects of type `PGreal` carry C++ `double` precision floating point values.

Sample Declaration: `PGreal x;`

Supported Operators: All C++ floating point operations are supported except for bit operations (e.g. `<<`, `&`, `|`, etc.) and `%`.

Example: Values in C++ `double` types may be assigned directly to `PGreal` objects, and vice versa. The following swaps the values in `x` and `y`:

```
void swap()
{
  PGreal x,y;
  double t;

  x = 3.7;
  y = 5.2;
  t = x;
  x = y;  // x is now 5.2
  y = t;  // y is now 3.7
}
```

3.1.3 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.

Sample Declaration: `PGbool b;`

Supported Operartors: All boolean operators in C++ apply to objects of type `PGbool`.

Example: 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.

```
void PGbool_assign()
{
  PGbool b;
  PGint i;

  i = -2;
  b = false;
  b = i;  // b now has a value of true
  i = b;  // i now has a value of 1
}
```

3.1.4 PGchar

Objects of type PGchar carry C++ character values.

Sample Declaration: `PGchar i;`

Supported Operators: All character operators in C++ apply to objects of type PGchar.

Example: Values in C++ characters may be assigned directly to PGchar objects, and vice versa. The following would swap the values in a and b:

```
void swap()
{
  PGchar a,b;
  char t;

  a = 'z';
  b = 'w';
  t = a;
  a = b;
  b = t;
}
```

3.1.5 PGstring

Sample Declaration: `PGstring s;`

Supported Operators: `=, ==, +` (concatenation)

Supported Functions: strcat(), strcpy(), strcmp() and strlen() can be applied to PGstring objects.

Example: Values in C++ string types may be copied to PGstring objects using the function strcpy, and vice versa. A PGstring object does not have a fixed length. The following would swap PGstring r and PGstring s:

```
void swap()
{
  PGstring r,s;
  char t[30];
  strcpy(r,"This is string 'r'.");
  strcpy(s,"This is string 's'.");

  strcpy(t,r);
  strcpy(r,s);
  strcpy(s,t);
}
```

3.1.6 PGmemoryBlock

An object of type PGmemoryBlock is used to store a sequence of bytes (in a more efficient way then a PGarray of characters.) PGmemoryBlock objects differ from PGstring objects in that a PGmemoryBlock can contain null characters.

Sample Declaration: `PGmemoryBlock b;`

Supported Operators: `=, ==`

Supported Functions: The following functions can be used with `PGmemoryBlock` objects (Let `b` be a `PGmemoryBlock`.):

- `int b.write(const char* s, int n, int delete_old=0)`
  Writes the first *n* characters of the block pointed to by *s* into *b*.
  The optional *delete_old* argument determines whether the previous contents of *b* should be overwritten. If *delete_old* is 0 (the default), the block *s* will be appended to the contents of *b*; otherwise, *s* will overwrite the contents of *b*.
  Returns 1 on success, -1 on failure.
- `int b.read(char* s, int n, int start=0)`
  Reads the *n* characters (starting from index *start*) from *b* into *s*.
  Returns the actual number of characters read.
- `int b.length()`
  Returns the length of the memory block contained in *b*.

3.2 Tuples

Tuples provide a mechanism for treating a logically related collection of values (of possibly different data types) as a single unit. Each component of a tuple is a *field*, the data type of which may be arbitrary, except that tuples cannot be defined within other tuples. Tuples can be static, meaning that the tuple definition is fixed at compile time like a C++ `struct` or they can be dynamic, meaning that fields can be added to the tuple at run-time. Static and dynamic tuples also differ in that declaration of a static tuple involves creation of a struct, whereas dynamic tuples are declared as instances of the dynamic tuple class. This means that static tuples can have methods as well as data, while dynamic tuples only contain a collection of data.

3.2.1 PGtuple (static)

`PGtuple` objects consist of a collection of named fields. Declaration is accomplished using a set of macros that create a C++ `struct`. In order to make members of a `PGtuple` publishable, they must be declared using the macro `PUBLIC_FIELDS`. Only Playground variables can be included in the `PUBLIC_FIELDS` macro. If the tuple is published only these fields will have their values sent out or updated by the environment. Furthermore, when one tuple is assigned to another only these fields are copied. It is permissible to use non-Playground variables as fields in a `PGtuple`, provided that they are not included as `PUBLIC_FIELDS`. Since they are not `PUBLIC_FIELDS`, they will not be published and they are not used in equality tests and tuple assignment. There is an optional user-defined initializiation routine `initialize` which is called by the default `PGtuple` constructor (see `Stock` example below.)

Syntax:

PGtuple(*typename*){
    typename1 *fieldname1*
    typename2 *fieldname2*
    ⋮
    PUBLIC_FIELDS(*typename*,
        field(*fieldname1*);
        field(*fieldname1*); ...)
};

Example: The following type definition, declaration, and assignments would result in a `Rectangle rect` with `height` 1.974 and `width` 1.888.

```
PGtuple(Rectangle){          // define the rectangle type
   PGreal h;
   PGreal w;

   PUBLIC_FIELDS (Rectangle,
                  field(h);
                  field(w))
};

Rectangle rect;

rect.h = 1.974;
rect.w = 1.888;
```

Example: The following `PGtuple` declaration defines a `Stock PGtuple`, with public `price`, `symbol` and `name` fields, an initialization routine, a method for entering trades of the `Stock`, methods for setting and accessing the private `Stock` ID number and ostream overloading. Note that since `ID` is private and not declared in the `PUBLIC_FIELDS` macro, the ID number is only accessible using `set_ID` and `get_ID` and is not available outside the module in which the `Stock` is declared. Usage of the `Stock` tuple is demonstrated in the routine `test_Stock`.

```
PGtuple(Stock){
  PGreal price;      // recent trading price of Stock
  PGstring symbol;   // symbol of Stock
  PGstring name;     // name of Stock

  PUBLIC_FIELDS(Stock, field(price); field(symbol); field(name));

  void initialize();                    // initializer
  void set_ID(int i) { ID = i; }        // set Stock's ID number
  int get_ID() { return ID; }           // get Stock's ID number
  void trade(double trade_price);       // enter a trade
  friend ostream& operator<<(ostream &os, Stock &s);
private:
  int ID;                               // Stock ID number
};

void
Stock::initialize()
{
  price = 0;
  strcpy(symbol,NULL);
  strcpy(name,"no name");
  ID = 0;
}

void
Stock::trade(double trade_price)
{
  price = trade_price;
}
```

```
ostream &
operator<<(ostream& os, Stock& s)
{
  os << "SYMBOL: " << s.symbol << " NAME: " << s.name
    << " CUR_PRICE: " << s.price << endl;
  return(os);
}

void
test_Stock()
{
  Stock s;     // declare Stock s, initializer called

  // at this point the price and ID of s are 0, the symbol is
  //  NULL, and the name is ''no name''

  s.set_ID(2235);
  strcpy(s.symbol,"SSI");
  strcpy(s.name,"Sandboxes & Seesaws, Inc.");
  s.trade(378.9);

  // at this point the price of s is 378.9, the ID is
  //  2235, the symbol is "SSI", and the name is
  //  "Sandboxes & Seesaws, Inc."
}
```

3.2.2 PGdynamicTuple

A `PGdynamicTuple` contains a collection of data fields of arbitrary Playground data types. The tuple can be based on another `PGdynamicTuple` or `PGtuple` and fields can be added at run-time using the operations described below. All fields in a `PGdynamicTuple` are published. Fields cannot be added when a `PGdynamicTuple` is published and fields cannot be removed at any time.

Syntax:

PGdynamicTuple *t*;
Builds a dynamic tuple *t* with no fields defined at declaration.

PGdynamicTuple *t2*(*t1*);
Builds a dynamic tuple *t2* containing the public fields defined in *t1*. The tuple *t1* may be either a `PGtuple` or a `PGdynamicTuple`. If it is a `PGtuple` then only the fields declared in the `PUBLIC_FIELDS` macro will be included in *t2*.

Operations: The following operations can be used with `PGdynamicTuple` objects (let `t` be a `PGdynamicTuple`.):

- `void t.add_field(PGobj *f, const char* n)`
  Adds the field `f` to the `PGdynamicTuple` `t`. The new field is given the name `n`. The pointer `f` holds the address of the actual storage inside the tuple, so modifications through `f` are shared by the tuple. This operation cannot be performed when `t` is published.

- `PGobj& t.get_field(const char* n)`
  Returns a reference to the field `n`. The return value must be explicitly cast. If the field with name `n` is not in `t`, the result is undefined.
- `PGobj* t.get_field_ptr(const char* n)`
  Returns a pointer to the field `n`. The return value must be explicitly cast. If the field with name `n` is not in `t`, NULL is returned.
- `t[const char* n]`
  Returns a reference to the field `n` (this is equivalent to the call `t.get_field(n)`.) The return value must be explicitly cast.
- `t[const char* n] = obj`
  Assigns a reference to `obj` to the field `n` in the tuple `t`. It is necessary to explicitly cast the left hand side to be a reference to the type of `obj`.

Example: The following example demonstrates usage of the `PGdynamicTuple` object. A `box` tuple is declared and fields are added and modified.

```
void make_box()
{
  PGreal width;
  PGreal height;
  PGreal length;
  PGreal *width_ptr;
  PGdynamicTuple Box;

  Box.add_field(&width, "width");
  Box.add_field(&height, "height");
  Box.add_field(&length, "length");

  (PGreal&)Box["height"] = 1.7;
  (PGreal&)Box["length"] = 3.7;
  (PGreal&)Box["width"] = 1.1;

  // Box has height of 1.7, length of 3.7, and width of 1.1

  height = (PGreal&)Box.get_field("length");          // height is changed to 3.7
  width = (PGreal&)Box["length"];                     // width is changed to 3.7
  width = 8.8;                                        // width is changed to 8.8
  width_ptr = (PGreal*)Box.get_field_ptr("width");
  length = *width_ptr;                                // Box now has height of 3.7 and
                                                      //  length and width of 8.8
}
```

Take note of how fields added at run time are assigned values. They must be explicitly cast to references before being used.

3.3 Aggregates

Playground provides *aggregates* for building data structures that consist of homogeneous collections of objects called *elements*. Some commonly used aggregates are provided in the Playground library. In addition, programmers may define their own aggregates with the general purpose aggregate *PGgrouping*. Connections to and from aggregates are discussed in Section 5.7.

3.3.1 PGmapping

A mapping is a relation from a *domain* type to a *range* type. It is content addressed. The mapping maintains a list of tuples of type `PGmappingItem` with two fields, `domain` and `range`. This provides a way to iterate through the entries of the mapping using the built-in `PGmappingIterator`.

Syntax:

PGmapping<D,R>*m*;
Define a mapping *m* with domain type `D` and range type `R`. `D` and `R` must be Playground types.

PGmappingItem<D,R>*i*;
Define a mapping item *i* with domain type `D` and range type `R`.

PGmappingIterator<D,R>*iter* (*m*, *d*);
Define an iterator *iter* for the mapping *m* of domain type `D` and range type `R`. *d* is an optional parameter (a pointer to an object of domain type `D`) - if included, the iterator will only iterate through entries with domain of *d*.

Operations: The following operations may be performed on mappings (Let `m` be a mapping from domain type `D` to range type `R`.):

- `m[a]`
  Returns the first range value assigned to domain value `a` in the mapping `m`. If `a` is not a member of `m`, then a new entry is created, with domain equal to `a` and range equal to the default range value.
- `m[a]=b`
  Creates a new entry in the mapping `m` with a copy of `a` for the domain value and a copy of `b` for the range value. If there is already an entry with domain `a`, it will be overwritten.
- `void m.add(D &d, R &r, int how=DUPLICATES_OK)`
  Adds an entry to the mapping `m` with a copy of `d` for the domain value and a copy of `r` for the range value. The third parameter is optional and defaults to allow duplicates, i.e there can be more than one range value for the same domain. In this case any one of the entries can be returned when requested. If the second parameter to add is `NO_DUPLICATES`, then add will overwrite the old range value. `m.add(d, r, NO_DUPLICATES)` is the same as `m[d]=r`.
- `void m.add(D *d, R *r, int how=DUPLICATES_OK)`
  Adds an entry to the mapping `m` with a pointer to `d` for the domain value and a pointer to `r` for the range value. The `how` parameter is treated the same way as in the copy semantics version of add, defined above.
- `void m.remove(const D &d)`
  `void m.remove(const D *d)`
  Removes and deallocates the first entry with domain value `d`.
- `void m.remove(const PGmappingItem<D, R> *pmi)`
  `void m.remove(const D &d, R const &r)`
  Removes and deallocates a specific entry by specifying both domain and range values.
- `int m.removeRangeByAddress(const R* r),`
  `int m.removeRangeByContent(const R& r)`
  Removes and deallocates the first entry with range `r` from the mapping. Returns 1 on success, 0 otherwise.

- `void m.clear()`
  Removes and deallocates all entries of a mapping. If a member has been published, then it will be unpublished before being removed.
- `PGmappingItem<D,R>* m.getItem(const D &d)`
  Returns a pointer to the first `PGmappingItem<D,R>` tuple associated with domain `d`. If the item is not present in the mapping, returns NULL.
- `PGmappingItem<D,R>* m.getItemByAddress(const D* d, const R* r, unsigned int n=1)`
  `PGmappingItem<D,R>* m.getItemByContent(const D* d, const R* r, unsigned int n=1)`
  Returns a pointer to the nth `PGmappingItem<D,R>` with domain `d` and range `r`. Either domain, range, or both can be specified in order to locate a desired mapping item. If no such entry exists then NULL is returned. `getItemByContent` uses the contents of the domain and/or range parameters to find the matching entry. `getItemByAddress` uses the address of the domain and/or range parameters. If neither domain nor range is specified, then the n^{th} entry is returned (or NULL if there are less then `n` entries.)
- `int m.numElements()`
  Returns the number of entries in the mapping.
- `int m.isEmpty()`
  Returns 1 if the mapping is empty, otherwise 0.
- `int m.isMember(D &d)`
  Returns 1 if `d` is a member of the mapping, otherwise 0.

The following operations may be performed on iterators (Let `iter` be a `PGmappingIterator` for the mapping `m` with domain type `D` and range type `R`):

- `PGmappingItem<D,R>* iter.first()`
  Returns the first entry in the mapping.
- `PGmappingItem<D,R>* iter.same()`
  Returns the mapping entry pointed to by the iterator.
- `PGmappingItem<D,R>* iter.next()`
  Moves the iterator to the next entry in the mapping and returns that entry.
- `void iter.reset()`
  Resets the iterator to the first entry in the mapping.

When using an iterator on a mapping it is important to be careful of removing items from the mapping. If the item that the iterator is pointing to is removed, it will no longer be possible to use the `next` operator safely. Therefore, it is necessary to use the `next` operator first, and then remove the item (see the end of the `main` routine in the example below.)

Example: The following example creates a mapping using the `Stock` tuple defined in Section 3.2.1. The domain is the `symbol` member of the `Stock` tuple and the range is the `Stock` object.

```
void
add_stock(PGmapping<PGstring,Stock> &m, PGstring symbol, PGstring name)
{
  Stock s;
  strcpy(s.symbol,symbol);
  strcpy(s.name,name);
  m.add(s.symbol,s);
}
```

```
void
print_map(PGmapping<PGstring,Stock> m)
{
  // use an iterator to print out the stocks
  PGmappingIterator<PGstring,Stock> iter(m);
  PGmappingItem<PGstring,Stock>* item;
  for(item=iter.first();item;item=iter.next())
    cout << item->range;
  cout << endl;
}

int
main()
{
  PGmapping<PGstring,Stock> m;

  // add some elements to the mapping
  add_stock(m,"SAN","Sandboxes, Inc.");
  add_stock(m,"TRE","Treehouses, Inc.");
  add_stock(m,"SEE","Seesaws, Inc.");
  add_stock(m,"SLI","Slides, Inc.");
  add_stock(m,"SWI","Swings, Inc.");
  add_stock(m,"SLI","Slides and Swings, Inc.");

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

  // several different ways of retrieving the range of the same element:
  Stock s;
  s = m.getItem("SLI")->range;                    // by domain
  cout << m.getItemByContent(0,&s)->range;        // by range
  PGstring str("SLI");
  cout << m.getItemByContent(&str,0)->range;      // by domain
  cout << m.getItemByContent(&str,&s)->range;     // both
  cout << m.getItemByContent(&str,&s,1)->range;   // the first matching entry
  cout << endl;

  // remove some elements from the mapping
  if (m.isMember((PGstring)"SWI")) m.remove((PGstring)"SWI");     // remove SWI
  m.remove(m.getItemByContent(0, 0, 1));                          // remove the first entry
  s = m.getItem("SEE")->range;
  m.removeRangeByContent(s);                                      // remove the entry with
                                                                  //  range s

  // delete elements with domain "SLI" using an iterator
  str = "SLI";
  PGmappingIterator<PGstring,Stock> iter(m,&str);
  PGmappingItem<PGstring,Stock>* item;
  item = iter.first();
  while(item) {
    iter.next();                  // move the iterator to the next item before removing
    m.remove(item);               // remove item
    item = iter.same();
```

```
    }
    cout << "Remaining elements:" << endl;
    print_map(m);
    m.clear();                          // clear the mapping
}
```

3.3.2 PGarray

As the name implies this is an array of Playground objects like `PGint`, `PGreal`, etc. An array of Playground objects must be defined first and then a `PGarray` can be set to that array. There are two ways to initialize a `PGarray`: by initialization at declaration time along with storage, or explcitily initialized to an array of Playground objects using `PGinitArray`.

Syntax:

PGarray<Playground type> *arr* (*s, d1, d2, ...*);
Creates a `PGarray` *arr* based on the storage *s* with dimensions *d1, d2, ...*

PGarray<Playground type> *arr*;
Declare the `PGarray` *arr*. This declaration requires that *arr* be initialized using `PGinitArray`.

PGinitArray(*arr, s, d1, d2, ...*);
Initialize the `PGarray` *arr* based on the storage *s* with dimensions *d1, d2, ...*

Example: The following example creates a table of `PGint` objects. Note that `PGarray` elements can be accessed using the storage array or the `PGarray` object.

```
void
int_arr()
{
  int i,j;
  const int nrows=5, ncols=8;
  PGint table_stor[nrows][ncols];                  // declare storage for the array
  PGarray<PGint> table(table_stor,nrows,ncols);    // associate PGarray object with the storage

  for (i=0;i<nrows;i++)
    for (j=0;j<ncols;j++)
      table_stor[i][j] = i * j;

  for (i=0;i<nrows;i++) {
    cout << "\t";
    for (j=0;j<ncols;j++)
      cout << table_stor[i][j] << "\t";
    cout << endl;
  }

  cout << "old: " << table(3,5) << "\t";
  table_stor[3][5] = 267;
  cout << "new: " << table(3,5) << endl;
}
```

Example: The following example shows usage of a `PGarray` in a spreadsheet class.

```
const int NUM_ROWS = 80;
```

```
const int NUM_COLS = 100;

class spreadsheet {
public:
  spreadsheet(int nrows, int ncols);
  PGreal cell_stor[NUM_ROWS][NUM_COLS];
  PGarray<PGreal> cells;
};

spreadsheet::spreadsheet(int nrows, int ncols)
{
  PGinitArray(cells,cell_stor,nrows,ncols);
}

void
test_sheet()
{
  int i,j;
  spreadsheet s(NUM_ROWS,NUM_COLS);

  for (i=0;i<=5;i++)
    for (j=0;j<=5;j++)
      s.cell_stor[j][i] = j * i;

  cout << "Cell (5,5): " << s.cell_stor[5][5] << endl;
  cout << "Cell (3,4): " << s.cells(3,4) << endl;
}
```

Note that if array dimensions are passed in to the constructor, these values must be of the same magnitude as the dimensions used in the storage declaration in the class definition.

3.3.3 PGgrouping

PGgrouping is a general purpose aggregate data type. Using PGgrouping together with the type PGptr it is possible to create complex, shareable Playground data structures such as trees, lists, and graphs.

A PGgrouping is a homogeneous collection of PGtuple objects and a *bookkeeper* data structure that is used to access the grouping in a specific way. A grouping is instantiated by PGgrouping<B,T>, where B is the type of the bookkeeper and T is the type of the objects in the grouping. Accessing the PGgrouping is accomplished using a PGptr or a bookkeeper object. The bookkeeper must be a Playground type, usually a PGptr or a PGtuple with one or more PGptr fields. The first field of T defines the *element type* of the grouping (i.e., in a linked list of PGint objects, the first field of T is of type PGint.)

The operations beginAccess and endAccess must be used when accessing the grouping. Furthermore, if the grouping is writable by the environment then it is important to call PGshelter before calling beginAccess and PGunshelter when finished accessing the grouping. See Section 4.4.1 for information about PGshelter and PGunshelter.

For example, a doubly linked list can be implemented by creating a PGgrouping of listnode PGtuple objects containing a Playground data type field and PGptr fields to the next and previous listnodes. The bookkeeper for the doubly linked list contains a PGptr to the head and one to the tail, and a PGint containing the number of elements in the list. An illustration of this example is shown in Figure 3.1 and example code is included below.

Syntax:

PGgrouping<B, T> *g*;
PGptr<T> *p*;

Operations: The following operations can be performed on groupings (Let g be a grouping with element type T and bookkeeper type B.):

- `PGptr<T> g.newEntry()`
  Adds a new entry to the grouping and returns a pointer to the entry.
- `PGptr<T> g.add(T& n)`
  Adds the entry n to the grouping. Returns a pointer to n.
- `void g.remove(PGptr<T>& p)`
  Removes the entry pointed to by p if it exists in g.
- `int g.numElements()`
  Returns the number of elements in g.
- `int g.isEmpty()`
  Returns 1 if g is empty.
- `void g.beginAccess(B* &b, PGptr<T> *p1,...)`
  Associates pointers and a bookkeeper object with a grouping. The ellipses indicate that a variable number of pointers (up to 5) can be passed as parameters.
- `void g.endAccess(B* &b, PGptr<T> *p1,...)`
  Disassociates pointers and bookkeeper from the grouping. After this operation is performed the bookkeeper pointer `b` and the `PGptr` objects will point to NULL.

The following operations can be performed on `PGptr` objects (Let `ptr` be a `PGptr` with base type T.):

- `int ptr.isNull()`
  Returns 1 if `ptr` points to NULL.
- `int ptr.clear()`
  Sets `ptr` to NULL. Returns 0 if successful, otherwise -1.

Example: Defining a doubly linked list (see Figure 3.1). Note that in the `insert` and `isMember` functions the bookkeeper is declared as a pointer to the bookkeeper type so that it can be passed into `beginAccess` and `endAccess`.

```
// define the ListNode tuple

PGtuple (ListNode) {
  PGint val;                  // the node's data
  PGptr<ListNode> next;
  PGptr<ListNode> prev;
  PUBLIC_FIELDS (ListNode,
                 field(val);
                 field(next);
                 field(prev));
  void initialize();
};
```

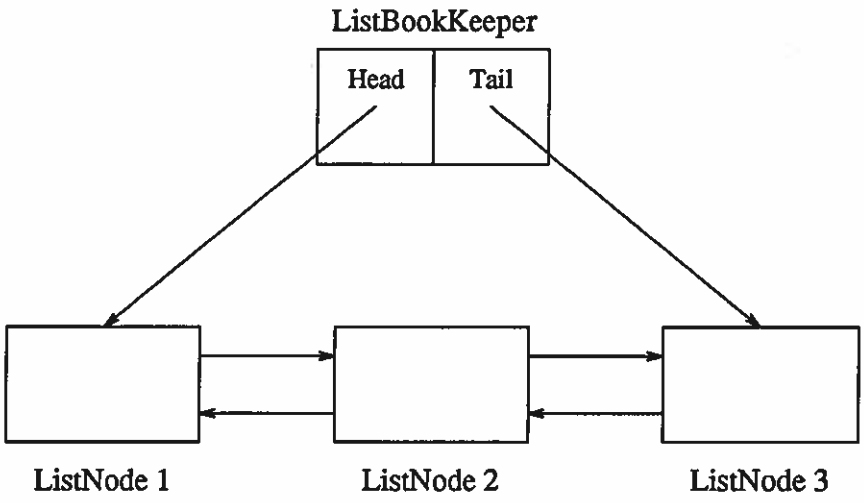

Figure 3.1: A Doubly Linked List Using PGgrouping

```
void
ListNode::initialize()
{
  val = 0;
  next.clear();
  prev.clear();
}

// define the bookkeeper

PGtuple (ListBookKeeper) {
  PGptr<ListNode> head;
  PGptr<ListNode> tail;
  PGint num_elements;         // number of elements in the list
  PUBLIC_FIELDS (ListBookKeeper,
                 field(head);
                 field(tail);
                 field(num_elements));
  void initialize();
};

void
ListBookKeeper::initialize()
{
  head.clear();
  tail.clear();
  num_elements = 0;
}

typedef  PGgrouping<ListBookKeeper, ListNode> List;
typedef  PGptr<ListNode>  ListPtr;
```

```
// Insertion and traversal operations for the doubly linked list.

// Insert an element into the grouping

void
insert(List& list, const PGint& i)
{
  ListPtr p;
  ListBookKeeper* bk;          // define the bookkeeper - note that this must be a pointer

  PGshelter();
  list.beginAccess(bk, &p);    // mandatory before accessing the list
  p = list.newEntry();
  p->val =  i;
  bk->num_elements = bk->num_elements + 1;

  if (bk->head.isNull()) {     // list was empty
    bk->head = p;
    bk->tail = p;
  } else {
    p->prev = bk->tail;
    bk->tail->next = p;
    bk->tail = p;
  }
  list.endAccess(bk, &p);
  PGunshelter();
}

// Traverse the list

int
isMember(List& list, const int& i)
{
  ListBookKeeper* bk;
  int found = 0;
  ListPtr p;

  PGshelter();
  list.beginAccess(bk,&p);
  for(p=bk->head; !found && !p.isNull(); p=p->next)
    if (p->val==i)
      found=1;
  list.endAccess(bk,&p);
  PGunshelter();
  return found;
}

// Try out the list

void
test_list()
{
  List squares;
  int i;
```

```
  for(i=1; i<11; i++)
    insert(squares,i*i);

  cout << "Perfect squares between 1 and 100 (inclusive):" << endl;

  for(i=1; i<=100; i++)
    if (isMember(squares,i))
      cout << "  " << i << endl;
}
```

3.4 Publishing Data and Managing the Presentation

3.4.1 PGpublish

Adding an entry to the presentation is accomplished with **PGpublish**(*var, external name, protection*) where

- *var* is the Playground variable to be published.
- *external name* is a character string which is the name that the variable will have in the presentation. It is not necessary, but is recommended for presentation items to have unique names.
- *protection* is the value of access permissions and is one of the following:
  - PRIVATE : the published variable can be neither read nor written.
  - READ_WORLD : the published variable can only be read.
  - WRITE_WORLD : the published variable can only be written.
  - RW_WORLD : the published variable can be read and written
  - ELT_TO_AGG : the variable is an aggregate type to which element-to-aggregate connections can be made.

Protection and the external name are optional and may be omitted, in which case protection defaults to RW_WORLD and the external name to NULL.

Calling `PGpublish` on a published variable will change the external name or protection. When changing a published variable, omitting the protection will cause it to default to RW_WORLD.

Example: The following demonstrates publishing of a `PGint`.

```
void
publishing()
{
  PGint var;
  PGinitialize("MODULE");              // initialize the module
  PGpublish(var, "VAR");               // publish the variable var under the name
                                       //  "VAR" and default protection of RW_WORLD

  PGpublish(var, "X", READ_WORLD);     // change var's name to "X" and protection to
                                       //  READ_WORLD (now the environment can no longer
                                       //  write to var)
  PGpublish(var, "Y");                 // change var's name to "Y" and change the protection
```

```
                                        //  back to the default RW_WORLD

    PGunpublish(var);                   // remove var from the presentation
    PGterminate();                      // shut down the module
}
```

3.4.2 PGunpublish

Variables can be unpublished using the `PGunpublish` statement: `PGunpublish(`*variable*`)`. This automatically removes them from the presentation and they are no longer visible outside the module.

3.4.3 PGrename

`PGrename` is used to rename a module.

Syntax: `PGrename(`*new name*`)`

Chapter 4

Control Flow

4.1 Basic Concepts

The control portion of a module defines how the module changes state and interacts with its environment. A Playground module's communication with the environment takes place through published variables in its presentation. As discussed in Chapter 1, a module may modify its local state and sometimes the published data may change "miraculously" due to the environment.

Control in a Playground module is divided into two parts – *active control* and *reactive control.* The active control carries out the ongoing computation while the reactive control responds to changes in the state caused by the environment. Applications may be structured with solely active or reactive control but typically have both.

The active control of a Playground application is defined by the `main` function of the module. Reactive control is defined by associating reaction functions with published data items.

4.2 Active Control

Playground modules are autonomous, meaning that they may carry out a computation without a request from the environment. The `main` function defines the active control of the module.

The active control interacts with the environment by reading and writing published variables. Communication occurs implicitly according to the (dynamically changing) configuration of logical connections. The counter program in Chapter 2 is an example of active control.

4.3 Reactive Control

The purpose of reactive control is to allow a module to take a specific course of action whenever a published data item is modified by the environment. This is accomplished by defining a function and associating it with the published data item. The reaction function is called whenever the veneer receives an external update for that published data item. In order for this to occur, the program must give the veneer a chance to run (ways accomplish this are described in Section 4.5.) The receipt of the update and the execution of the associated reaction function is atomic as far as the active control can tell. Therefore, the active control cannot observe the updated value before the reaction function has executed.

A reaction function has one mandatory argument (the object that is externally updated) and one optional argument which is of type `void*`. This allows the programmer to pass arbitrary data to a reaction function. A reaction function is defined as below.

Syntax: `void` *function*`(`*t*`* obj, void* extra object)`
Where *function* is the name of the function, *obj* is the object (of type *t* – this must be a Playground data type) which the function is reacting to, and the optional *extra object* is the arbitrary data being passed into the function.

A reaction function is registered using one of the functions described below. Note the different ways of registering an aggregate type: reacting on the aggregate as a whole, or on the individual elements. In the former case, the entire aggregate is passed into the reaction function, in the latter, only the element that changed is passed into the reaction function. It is possible to register the same reaction function to more then one variable (see example below.)

Syntax:

PGreact(*obj*, *fn*);
Registers the reaction function *fn* to react to changes in *obj*. If *obj* is an aggregate, the entire aggregate is passed to *fn* whenever an element in the aggregate changes.

PGreactPlus(*obj1*, *fn*, *obj2*);
Registers the reaction function *fn* to react to changes in *obj1* and passes *obj2* into the function. If *obj* is an aggregate, the entire aggregate is passed to *fn* whenever an element in the aggregate changes.

PGelementReact(*agg*, *fn*);
Registers the reaction function *fn* to react to changes in any element in the aggregate *agg*. When an element is changed, only that element gets passed into *fn*, not the entire aggregate.

PGelementReactPlus(*agg*, *fn*, *obj*);
Registers the reaction function *fn* to react to changes in any element in the aggregate *agg* and passes *obj* into the function. When an element is changed, only that element gets passed into *fn*, not the entire aggregate.

Example: The example below uses the same reaction function to react to two published temperature variables. The module continually reacts to changes in both temperatures until one or the other has reached its respective "target_temperature" at which point the module shuts down.

```
int TARGET_REACHED = 0;   // set to 1 when one of the two target temperatures has been reached

void update_temp(PGreal& temp, void* obj)
{
  float* target_temp = (float*)obj;     // the extra object must be cast
  if (temp < *target_temp) {
    cout << "Current temp. (" << temp << ") is below target (" << *target_temp << ")." << endl;
  } else if (temp > *target_temp) {
    cout << "Current temp. (" << temp << ") is above target (" << *target_temp << ")." << endl;
  } else {
    cout << "Target temperature (" << *target_temp << ") has been reached." << endl;
    TARGET_REACHED = 1;
  }
```

```
}

void
main()
{
  float target_temp1, target_temp2;
  PGreal temp1, temp2;

  PGinitialize("TEMP_CONTROL");          // initialize the module

  // publish the two temperature variables
  PGpublish(temp1,"temp1",WRITE_WORLD);
  PGpublish(temp2,"temp2",WRITE_WORLD);

  // set the target temperature for the temperature variables
  target_temp1 = 537.6;
  target_temp2 = 647.8;

  // register the reaction function with both variables
  PGreactPlus(temp1,update_temp,&target_temp1);
  PGreactPlus(temp2,update_temp,&target_temp2);

  PGreactUntilTrue(TARGET_REACHED);      // continue reacting to changes
        //  until one of the two targets
                                         //  has been reached
  cout << "Target temperature has been reached. Closing down..." << endl;
  PGterminate();                         // shut down the module
}
```

4.4 Preventing Presentation Updates

At times it may be desirable to prevent published variables from being updated. This can be accomplished by using the functions `PGshelter` and `PGbeginAtomicStep`. `PGshelter` prevents updating of incoming values and `PGbeginAtomicStep` prevents updating of both incoming and outgoing values. Both functions must be matched with the corresponding `PGunshelter` or `PGendAtomicStep` in order to resume updating.

4.4.1 PGshelter

The function `PGshelter` shields the entire presentation from the environment. Incoming values are not seen but updates go out. When `PGunshelter` is called the queued updates are allowed to come in.

Syntax:

PGshelter();
PGunshelter();

4.4.2 PGbeginAtomicStep

Atomic operations can be achieved using the statements `PGbeginAtomicStep` and `PGendAtomicStep`.

The series of statements intended to be executed atomically is bracketed between a `PGbeginAtomicStep` and a `PGendAtomicStep` statement.

At the beginning of the atomic step the programmer may specify a list of Playground variables to be accessed inside the atomic step. During an atomic step, no updates to or from the environment will occur. At the end of the atomic step, 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.

Syntax:

PGbeginAtomicStep(*list*);
Where *list* is a NULL terminated array of pointers to Playground variables, or
PGbeginAtomicStep(*n,ptr 1, ...ptr n*);
Where *ptr 1, ...ptr n* refers to *n* pointers to Playground variables.
PGendAtomicStep();

Note: Currently, the object list to `PGbeginAtomicStep` is not required. However, for compatibility with future implementations of Playground, every published object that could possibly be used or modified inside an atomic step should be listed in the parameter list to `PGbeginAtomicStep`. Also, atomic steps should not be nested and should not call other functions that may modify arbitrary published variables.

Example: The following procedure `swap` exchanges the two published `PGreal` objects `a` and `b` every 5 seconds, until both variables equal -1.

```
void
swap()
{
  PGreal a = 0.0;
  PGreal b = 0.0;
  float temp;

  PGinitialize("SWAP");
  PGpublish(a,"a");
  PGpublish(b,"b");

  while(!((a == -1) && (b == -1))) {
    PGbeginAtomicStep(2,&a,&b);
    temp = a;
    a = b;
    b = temp;
    PGendAtomicStep();
    PGsleep(5);
  }

  PGterminate;
}
```

4.5 Sharing Control with the Veneer

The veneer checks for updates to the presentation whenever a Playground variable is read or written (unless the read or write action occurs after a `PGbeginAtomicStep` or the read occurs after a `PGshelter`.)

In addition to this, it is possible to give the veneer a chance to run using any of the following:

- **PGupdatePresentation()** explicitly asks the veneer to process any pending updates.
- **PGsleep** (`int` *seconds*, `int` *nanoseconds*) causes the module to sleep for at least the specified time and calls the veneer to update the presentation before waking up the module.
- **PGreactUntilTrue**(`int` *termination variable*, `unsigned int` *active time*, `unsigned int` *sleep time*) gives control to the veneer and continues waiting and acting on updates to the presentation until *termination variable* has a non-zero value. The termination variable must be set in a reaction function or else the program will never terminate. `PGreactUntilTrue` is particularly useful in modules with solely reactive control. The parameter *active time* (in microseconds) is the approximate time spent looping waiting for updates to arrive before going to sleep (defaults to 5000 microseconds) and *sleep time* (in microseconds) is the minimum time for which the veneer sleeps before checking for updates (defaults to 100 microseconds.) `PGreactUntilTrue` checks to see if any updates have arrived in the last *active time* interval. If an update did arrive, the function goes back to checking for updates, otherwise it goes to sleep for the time specified in *sleep time.* It repeats the cycle until *termination variable* is true.

Chapter 5

Configuring Applications

5.1 Basic Concepts

For the environment to be able to change the value of a presentation data item of a module, the environment itself needs to be defined. The environment, in fact, is a collection of Playground modules communicating among themselves. Relationships need to be established among the data items in the presentations of the modules. These relationships can be established by creating *connections* among the presentation variables. Together, these relationships define a *configuration.*

This chapter deals with connections, how they are made, and what types of connections are possible. The current implementation assumes that the hardware and underlying network are failure-free. Fault-tolerance is left for future versions of the Playground system.

5.2 Using the connection manager

All connections between modules in the Playground system are managed by a special Playground module called the connection manager. The connection manager enforces type compatibility across connections and guards against access protection violations by establishing only authorized connections. [1]

The connection manager has a front-end that is a graphical user interface allowing users to establish connections between presentation items of the Playground modules. The connections can be drawn using a mouse. Chapter 2 explains how to bring up the connection manager and the front end.

5.3 Unidirectional connections

A unidirectional connection defines a flow of data from a producer to a consumer. A unidirectional connection is represented in the connection manager front end by an arrow from the producer presentation variable to the consumer presentation variable. The connection is made by drawing a line using the left mouse button from the source variable to the destination variable. The direction of communication is indicated by the arrow.

The semantics are first-in-first-out, meaning that updates to the consumer's presentation will occur in the same order as the updates made by the producer.

[1] The connection manager is itself a Playground module. It interacts with protocol processes of other Playground modules through element-to-aggregate connections that are automatically set up by those protocol processes.

Figure 5.1: Unidirectional communication

Figure 5.1 shows a unidirectional connection between two Playground modules M1 and M2. If module M1 produces two values of x then it is guaranteed that M2 will see these values in exactly the order that they were generated.

5.4 Bidirectional connections

In some cases, two modules may need to exchange information of the same type, as in shared data or collaborative work applications. Instead of having each module publish two variables and make unidirectional connections between the two modules in opposite directions, it is possible to create a single bidirectional connection between the two modules. A bidirectional connection is represented in the connection manger front end by arrows on both ends of the link. The connection is made by drawing a line from one presentation variable to the other using the middle mouse button.

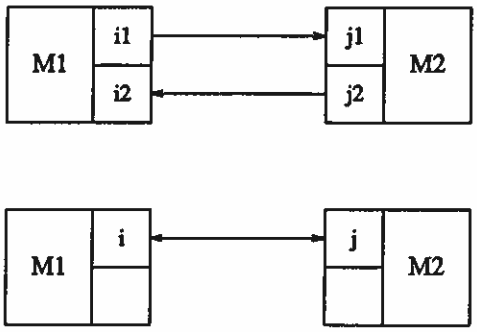

Figure 5.2: Bidirectional Communication

This also allows the modules to act on a particular variable instead of keeping track of two related variables where one is just needed for communication. Bidirectional connections mean that updates to variables at either end result in updates to the variable at the other end of the connection. Currently the only communication semantics implemented are asynchronous. The connections are FIFO queues in each direction but currently nothing prevents them overwriting each other. Currently it is possible for values to "cross" each other in transit.

Example Figure 5.2 shows bidirectional connections between two Playground modules M1 and M2. Module M1 may write a value of 1 in presentation item "i" at the same time that M2 writes the value 5 in "j". These values are sent out over the connection and may cross each other. Thus, each module overwrites the other.

5.5 Fan-in and fan-out

Arbitrary fan-in and fan-out are permitted meaning that multiple simple connections can converge into or diverge from a single presentation item. High fan-out would be useful where a single producer like a video source is sending video data to multiple viewing modules.

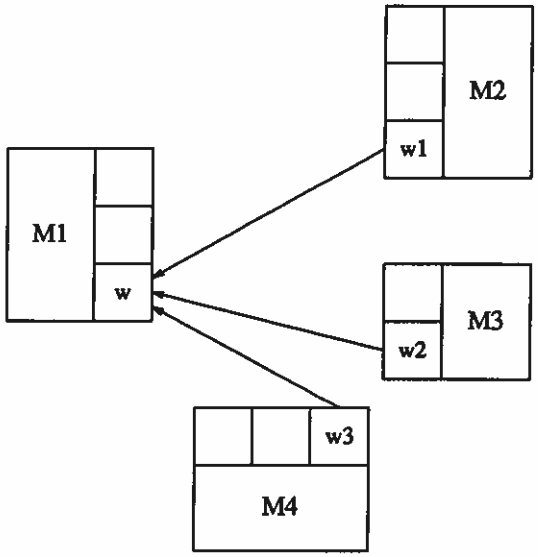

Figure 5.3: Fan-in

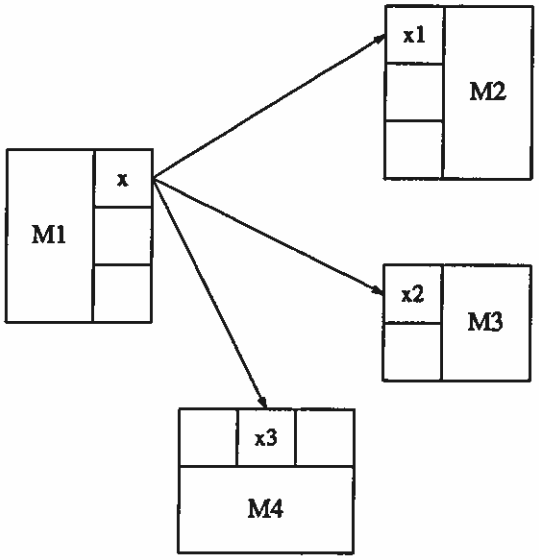

Figure 5.4: Fan-out

There is no particular ordering among multiple connections. In the case of fan-in two updates from

different sources may arrive "simultaneously" but may be seen by the receiving module in any arbitrary order. Similarly in the case of fan-out the update may be sent along the various connections in any order.

5.6 Deleting connections

A connection can be deleted at any time to stop communication. We define data to be in transit if a presentation item has been updated but the value has not been received yet at the other end of the connection. Data in transit may not be received if the connection is broken. Connections can be deleted in the connection manager graphical user interface by positioning the cursor on the connection and clicking the third mouse button.

5.7 Element-to-aggregate connections

For an overview of element-to-aggregate connections refer to Section 1.2.3.2.

An element-to-aggregate connection between an integer variable of a module M1 and an aggregate of integers in another module is drawn just as any other ordinary connection would be drawn. The programmer can publish an aggregate and make it available for element-to-aggregate connections as follows.

`PGpublish(PGobj& object,char* name,int protection,f_new,f_del)`

The protection field must be of the form `READ_WORLD|ELT_TO_AGG` or `RW_WORLD|ELT_TO_AGG` etc. The argument `f_new` is a pointer to a function that will be called automatically when a new element-to-aggregate connection is formed with the aggregate. When called, the function pointed to by `f_new` returns a pointer to a `PGobj` that is then published by the veneer as the distinguished element of the new element-to-aggregate connection. The programmer has the option of either creating a new element in `f_new` or simply returning an existing element. The last parameter `f_del` is a pointer to a function that will be called when a connection is to be deleted. It takes as a parameter the pointer to the element being deleted. The function `f_del` is optional. If it is not defined then no action is taken when the connection is broken.

Example: This example demonstrates the use of element-to-aggregate connections using a `PGmapping`. The module accepts an arbitrary number of numeric-type connections and computes their sum. The summation value is printed by the module and published in the variable `SUM`.

```
PGmapping<PGint,PGreal> operands;     // declare the mapping
PGreal SUM = 0.0;                     // declare and initialize the sum value

int ID = 0;        // the ID is used as the domain of the mapping.
                   //  In this example, the domain is not relevant.

void
compute()
{
  float s=0;       // temporary sum holder
  PGmappingIterator<PGint,PGreal> oper_iter(operands); // set up an iterator
  PGmappingItem<PGint,PGreal>* item;
  for (item=oper_iter.first(); item; item=oper_iter.next()) {
    // while there is still an item in the mapping, add its
    //  value to the temporary sum holder
    s = s + item->range;
  }
```

```
  SUM = s;        // assign the new sum value to the published SUM variable
  cout << "SUM (" << operands.numElements() << " operands): "
    << s << endl;
}

PGobj*
new_operand(PGcoll* agg)
{
  ID++;
  PGreal r = 0.0;
  operands.add(ID,r);      // add an element (with range = 0 and
                           //  domain = ID (the domain is not
                           //  significant in this mapping)) to
                           //  the operands mapping
  cout << "Operand added. There are now "
    << operands.numElements() << " operands." << endl;
  PGreal& added_operand = operands[ID];     // create a reference
                                            //  to the new element
  return (PGobj*)&added_operand;            // return this reference
}

void
del_operand(PGcoll* agg, PGobj* obj)
{
  PGmapping<PGint,PGreal>* ops = (PGmapping<PGint,PGreal>*) agg;
  PGreal* r = (PGreal*) obj;        // create a pointer to the object
                                    //  that is being removed from
                                    //  the aggregate
  operands.removeRangeByAddress(r); // remove the object from
                                    //  the operands mapping
  cout << "Operand removed. There are now "
    << operands.numElements() << " operands." << endl;
  compute();                        // compute the new sum
}

void
ReactToChange(PGmapping<PGint, PGreal> &m)
{
  compute();     // whenever an operand value changes, compute
                 //  the new sum
}

void
main()
{
  int done = 0;
  PGinitialize("SUM");

  // set up the element to aggregate connection and register
  //  the new and delete functions
  PGpublish(operands,"ops",WRITE_WORLD | ELEMENT_TO_AGG,
    new_operand, del_operand);
  PGpublish(SUM,"sum",READ_WORLD);       // publish the sum variable
  PGreact(operands,ReactToChange);       // register the reaction function
```

```
    PGreactUntilTrue(done);                    // continue reacting until user terminates module
    PGterminate();
}
```

If an aggregate is willing to accept element-to-aggregate connections, then it cannot have an incoming simple connection, or else the correspondence between connections and distinguished elements would be lost. Reaction functions involving element-to-aggregate connections are discussed in Section 4.3.

5.8 Message ordering

The order in which updates to a presentation item arrive at a module may be significant depending on the module. Updates are sent to connected items as they occur. There is no ordering between updates from different modules arriving at one module.

Bibliography

[1] Jerome R. Cox, Jr., Mike Gaddis, and Jonathan S. Turner. Project Zeus: Design of a broadband network and its application on a university campus. *IEEE Network*, pages 20–30, March 1993.

[2] Kenneth J. Goldman. Welcome to The Programmers' Playground! World wide web URL http://www.cs.wustl.edu/cs/playground.

[3] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ramachandran Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering.* to appear.

[4] T. Paul McCartney and Kenneth J. Goldman. EUPHORIA reference manual. Technical Report WUCS–95–19, Washington University in St. Louis, July 1995.