C++ Types

C++ has a larger set of types than Java, including: primitive types, constants, arrays, enumerations, structures, unions, pointers, and classes. You can also define new type names using the *typedef* facility.

C++ classes will be discussed in a separate set of notes. The other types are discussed below.

Primitive (built-in) types

The primitive C++ types are essentially the same as the primitive Java types: *int*, *char*, *bool*, *float*, and *double* (note that C++ uses *bool*, not *boolean*). Some of these types can also be qualified as *short*, *long*, *signed*, or *unsigned*, but we won't go into that here.

Unfortunately, (to be consistent with C) C++ permits integers to be used as booleans (with zero meaning false and non-zero meaning true). This can lead to hard-to-find errors in your code like:

```
if (x = 0) ...
```

The expression "x = 0" sets variable x to zero and evaluates to false, so this code will compile without error. One trick you can use to avoid this is to write all comparisons between a constant and a variable with the constant first; e.g.:

```
if (0 == x) ...
```

Since an attempt to assign to a constant causes a syntax error, code like:

```
if (0 = x) ...
```

will not compile.

Constants

You can declare a variable of any of the primitive types to be a constant, by using the keyword **const**. For example:

```
const int MAXSIZE = 100;
const double PI = 3.14159;
const char GEE = 'g';
```

Constants must be initialized as part of their declarations, and their values cannot be changed.

Enumerations

New types with a fixed (usually small) set of possible values can be defined using an **enum** declaration, which has the following form:

```
enum type-name { list-of-values };
```

For example:

```
enum Color { red, blue, yellow };
```

This defined a new type called **Color**. A variable of type Color can have one of 3 values: red, blue, or yellow. For example, here's how to declare and assign to a variable of type Color:

```
Color col;
col = blue;
```

Structures

A C++ structure is:

- a way to group logically related values together
- very similar to a C++ class

Here's an example declaration:

```
struct Student {
    int id;
    bool isGrad;
};                    // note that decl must end with a ;
```

The structure name (Student) defines a new type, so you can declare variables of that type, for example:

```
Student s1, s2;
```

To access the individual fields of a structure, use the dot operator:

```
s1.id = 123;
s2.id = s1.id + 1;
```

It is possible to have (arbitrarily) **nested** structures. For example:

```
struct Address {
    string city;
    int zip;
};

struct Student {
    int id;
    bool isGrad;
    Address addr;
};
```

Access nested structure fields using more dots:

```
Student s;
s.addr.zip = 53706;
```

If you get confused, think about the type of each sub-expression:

- The type of s is Student, so it makes sense to be able to follow s with ".addr" (since addr is a field of struct Student).
- The type of s.addr is Address, so it makes sense to be able to follow the expression "s.addr" with ".zip" (since zip is a field of struct Address).
- The type of s.addr.zip is int, so it makes sense to be able to assign 123 to s.addr.zip.

Unions

A union declaration is similar to a struct declaration, but the fields of a union all share the same space; so really at any one time, you can think of a union as having just one "active" field. You should use a union when you want one variable to be able to have values of different types. For example, assume that only undergrads care about their GPA, and only grads can be RAs. In that case, we might use the following declarations:

```
union Info {
    double GPA;
    bool   isRA;
};

struct Student {
    int id;
    bool isGrad;
    Info inf;
};
```

Of course, we could just add *two* new fields to the Student structure (both a GPA field and an isRA field), but that would be a bit wasteful of space, since only one of those fields would be valid for any one student. Using a union also makes it more clear that only one of those two fields is meaningful for each student.

It is important to realize that it is up to you as the programmer to keep track of which field of a union is currently valid. For example, if you write the following bad code:

```
Info inf;

inf.GPA = 3.7;
if (inf.isRA) ...
```

you will get neither a compile-time nor a run-time error. However, this code makes no sense, and there is no way to be sure what will happen -- the value of inf.isRA depends on how 3.7 is represented and how a bool is represented.