Class 5: C++ part III, and using APIs

Pointers

Pointers are just values that indicate memory addresses and the type of data that the programmer thinks is there.

Declaration syntax: *type* `*` *pointer_variable*;

Dereference operator (get the value at this address): `*` *pointer_variable*;

Reference operator (get the address of a variable as a pointer): `&` *variable*;

"Syntactic sugar" for pointers to structs

Suppose `bptr` is a pointer to a structure:

```
struct MyBalls_s * bptr;
```

Then I could access the fields in the structure like this:

```
(*bptr).x = (*bptr).x + (*bptr).u * dt;
```

The `->` operator is "syntactic sugar" to make this prettier:

```
bptr->x = bptr->x + bptr->u * dt;
```

Formally: *pointer* `->` *fieldname* is equivalent to `(*` *pointer* `).` *fieldname*

Pointers to functions

If `f` is a function, then `&f` is a pointer to its address. Example:

```
#include <iostream>

double square(double x)
{  return x*x; }

double cube(double x)
{  return x*square(x); }

int main() {
  using namespace std;
  double (*fptr)(double);      // fptr declared as type "double (*)(double)"
```

```
  double t=5.0;

  fptr= &square;  // note no ()
  cout << "square of " << t << " is " << (*fptr)(t) << "\n";
  fptr= &cube;
  cout << "cube of " << t << " is " << (*fptr)(t) << "\n";
}
```

Object oriented programming in C

- You can do object oriented programming using `structs` in C.
- This is what we did in the "bounce2" exercise last class.
- You can even put pointers to functions in the data fields to make your objects more abstract.
- X windows and Microsoft Windows and many other applications and libraries are written this way.
- That's pretty much all there is to object oriented programming.
- C++ adds some nice "syntactic sugar".

Equivalents in C++: "syntactic sugar" for "member functions"

C++	C

`struct MyStruct_s {`
`  int i;`
`  void set_i(int ii);`
`};`
 	
`void test() {`
`  struct MyStruct_s s;`
`  s.set_i(42);`
`}`
	`struct MyStruct_s {`
	`  int i;`
	`};`
	`void MyStruct_set_i(int ii);`
	
	`void test() {`
	`  struct MyStruct_s s;`
	`  MyStruct_set_i(&s, 42);`
	`}`

Equivalents in C++: "syntactic sugar" for "virtual functions"

C++	C

```
struct MyStruct_s {
  int i;
  virtual void set_i(int ii);
};

void test() {
  struct MyStruct_s s;
  s.set_i(42);
}
```

```
struct MyStruct_s {
  int i;
  void (*set_i)(int ii);
};

void test() {
  struct MyStruct_s s;
  (*(s.set_i))(42);
}
```

The above example is just to give you the idea, and is slightly incomplete because I omitted the initialization of the function pointer, and because I don't use a "virtual function table" like most C++ compilers.

C++ `class` type

This is just like a `struct` with the addition of `public`, `protected` and `private` flags for the members:

```
class MyClass {
  private:
    int i;
  public:
    virtual void set_i(int ii);
};

void test() {
  MyClass c;
  c.set_i(42);  // OK
  c.i= 45;      // compile-time error
}
```

Constructors and Destructors

A constructor is a function that automatically gets run each time a variable of a particular class type is made. It has the same name as the class, and doesn't have a declared type, not even void.

A destructor is a function that automatically runs each time a variable of class type is deleted, e.g., when the block of code in which it was made ends.

```
class MyClass {
  public:
    MyClass();      // constructor
    MyClass(int i); // constructor with 1 argument
    ~MyClass();     // destructor
}
```

Dynamic memory allocation in C++

Sometimes you don't know in advance how many objects will need to make. (E.g., if the number of balls can be specified by the user at run time.) A vector is one way of handling this. Another way is the `new` operator:

```
MyClass *cptr1;
MyClass *cptr2;
cptr1= new MyClass;     // runs default constructor
cptr2= new MyClass(42); // runs constructor with 1 int argument
```

This allocates memory for the new object and runs the constructor.
You have to explicitly free the memory when you're done with it:

```
delete cptr1;
```

Other important C++ features

Significant:

Compiler features: templates, exceptions

Standard library features: file I/O, string stream I/O

Widely appreciated:

Compiler: function overloading, operator overloading, namespaces

Break

What is an API

API: Application Programming Interface

- This basically just means the functions "you" can call in an external library.
- People used to talk loosely of "the user" calling these functions, meaning "the user of the library".
- It's considered more correct to use the word "user" to mean the person sitting in front of the computer interacting with some application program.
- It's really some part of a program calling these functions, not "you" or "the user". Hence the term API.

Linking

- The compiler can automatically link to the standard libraries for you.
- If you want additional libraries, you have to tell the compiler.
  - For gcc, use option `-L` *(directory)* to tell it where to find the library files, and `-l` *(name)* to tell it which library files.
  - For VC++, set `LIB` environment variable and/or give the names of the libraries on the command line.

Linking to the ROOT package libraries

ROOT is a package of tools useful for analyzing and plotting data.

- You can also use its graphics for other things.

Compilation recipes:

- VC:

```
C:\ROOT\BIN\thisroot.bat
set INCLUDE=%INCLUDE%;%ROOTSYS%\include
cl /EHsc (filename.cc) C:\ROOT\LIB\lib*.lib
```

- GCC (linux):

```
source /path-to-root/bin/thisroot.sh
g++ `root-config --cflags --libs` (filename.cc) -o exename
```

Example: 2-d bouncing ball using ROOT graphics

The ROOT API is too complicated to summarize in class, although the documentation is not bad.

There is a link to an example application on the course web page.

References

[ROOT] http://root.cern.ch/