Today's lecture

- C++ roots: C, Simula, Smalltalk
- C++ language features
  - Is a better version of C
  - Object oriented (OO) features
  - Efficient, useful, but complex

"...we do use C++ regularly and find it very useful but certainly not perfect. Every full moon, however, we sacrifice a virgin disk to the language gods in hopes that the True Object Oriented Language will someday manifest on earth, or at least on all major platforms."

-- Rick Pember, LLNL

Better version of C

- Function types are more explicit; better type checking
- Compiler flags errors
- No more implicit declarations. In C, a function that is not declared is implicitly assumed to be a “int fun()”
- You have to explicitly provide the argument list
- In C, “int fun()” declaration does not preclude the program from passing arguments to fun
- Declare variable names at any point in the program
- Better print statements (type safe)
- Can interface with C code easily:
  
  ```c
  extern "C" int strcmp(char *, char *);
  ```

Example

```c
#include <iostream.h>
extern int foo(int x); // not "extern int foo();"
int bar(int y)
{
int z = foo(y);
cin >> z;
for (int j=1; j<10; j++)
z += j;
int k = z + 10;
return k;
}
```

Question: what are the implications of explicit types and rigorous type-checking?

A Fortran Anecdote

- Urban folklore - a spacecraft was supposed to have been lost by the following bug (but didn't actually happen).

```
10  Continue

Do 10 I = 1, 10
```

```c
instead of:

DO 10 I = 1, 10

10 Continue
```

- Compiler interpreted the statement as an assignment to variable “DO10”

Object Oriented Language

- Objects are active entities:
  - contain data fields
  - contain methods to operate on data

- Terminology:
  - Classes: blueprint for objects (extension of structs and types in C)
  - A particular object is an instance of a class
  - Member functions are the methods that operate on objects
  - Constructors and destructors are special methods used for initializing and removing objects
  - Methods and data can be declared to be public or private
Example: Stack class

class Stack {
  public:
    void Push(int value); // everything is public!
    int top;              // which one is it?
    int stack[10];
  }

  void Stack::Push(int value) {
    if (top < 10)                        // top is visible
      stack[top++] = value;
  }

Creating Objects

Stack::Stack()              // no return type; not even "void"
{                             Stack::Stack()
  top = 0;                    size = 10; top = 0;
}                             stack = new int[size];

void foo() {
  Stack s1; // object is allocated on the stack
  // constructor automatically called
  s1.Push(10); // invoke Push on s1
  Stack *sptr; // no space allocated yet
  sptr = new Stack; // space allocated on the heap
  sptr->Push(20);
}

Customizable Stack

class Stack {
  public:
    Stack();                   Stack::Stack()
    Stack(int sz);            size = 10; top = 0;
    void Push(int value);      stack = new int[size];
    private:
      int top;
      int size;
      int *stack;
  }

  Stack::Stack(int sz) {
    size = sz; top = 0;
    stack = new int[size];
  }

Overloading and Deletion

void foo() {
  Stack s1; // gets you stack of size 10
  Stack s2(42); // gets you stack of size 42
  Stack *s3 = new Stack(50);
  delete s3;
}

  Stack::~Stack() {
    delete [] stack; // delete an array of integers
  }
}

Object Oriented Features

- Discussion:
  - How do we implement the features described so far?
  - What are the compile-time/run-time implications?

Inheritance

- Two purposes:
  - define a standard interface for many classes (shared behavior)
  - reuse code across classes

- C++ allows multiple inheritance
  - almost all of the standard C++ libraries use inheritance
  - very hairy to understand code with multiple inheritance
  - tough to understand code even with single inheritance; never sure
    where the code is coming from

- Underscores the importance of the design aspect in developing a program
Example of Shared Behavior

```cpp
class Stack {
public:
    Stack() { } // inline method, easier to write
    virtual ~Stack() { }
    virtual void Push(int value) = 0;
};
```

- A derived class inherits from a base class or superclass
- Virtual methods can be defined or redefined by the derived class
- Cannot instantiate a Stack object because it has a pure virtual method
- Destructors for base classes have to be virtual

Derived classes

```cpp
class ArrayStack : public Stack {
public:
    ArrayStack(int sz) {   Stack();  size = sz;  stack = new int[sz];  }
    void Push(int value)  {
        if (NumPushed() < size) {
            stack[NumPushed()] = value;
            Stack::Push();
        }
    }
};
class ListStack : public Stack {
public:
    ListStack(int sz)      {  list = new IntList;  }
    void Push(int value)   {  list->Prepend(value);  }
};
```

Using base types

- Can refer to instances of derived classes using the base type
- The right method gets invoked

```cpp
Stack *s1 = new ArrayStack(10);
Stack *s2 = new ListStack;
s1->Push(10);
s2->Push(12);
delete s1;
delete s2;
```

Sharing Code through Inheritance

```cpp
class Stack {
private:
    int numPushed;
public:
    virtual ~Stack() { numPushed = 0; }
    virtual void Push(int value) { numPushed++; }
    int NumPushed() {   return numPushed;  }
protected:
    Stack()   { numPushed = 0;  }
};
ArrayStack::ArrayStack(int sz) {   Stack();  size = sz;  stack = new int[sz];  }
ArrayStack::Push(int value)  {
    if (NumPushed() < size) {
        stack[NumPushed()] = value;
    }
    Stack::Push();
}
```

C++ Inheritance Summary

- Inheritance used for sharing behavior and sharing code
- Derived classes inherit all of the data and methods from the base class
- Generalizes to a hierarchy of classes
- How do we implement an operation such as:
  ```cpp
  StackPtr->dataVal = 4
  ```
  or
  ```cpp
  StackPtr->method();
  ```
  - Method invocation depends on whether method is declared virtual or not
  - If not virtual, hardcode function address at the call site
  - If virtual, index through a table of function pointers
- Question: how do we implement multiple inheritance?

Announcements

- No class on Monday
- Suggested work:
  - Fiddle around with threads in Nachos
- Suggested background reading:
  - "Emperor's Old Clothes" – Tony Hoare
OO Discussion
- Protected allows access only to members of derived classes
- All of the base class’s code exists, just need to invoke it using the right syntax
- Division of labor between the different classes in constructing and deleting the objects
- Easy to write code that doesn’t make much sense!
- Runtime system and compilers need to be smart to implement all of these language features

Templates
- Parameterize a class definition over many types
- DoubleStack is not going to be much different from intStacks
- Compiler generates as many versions of the classes as required

```
template <class T> class Stack {
public:
    Stack(int sz);
    void Push(T value);
private:
    int size;
    int top;
    T *stack;
};
```

Template methods and usage
```
template <class T>
Stack<T> Stack(int sz) {
    size = sz;
    top = 0;
    stack = new T[size];
}
template <class T>
void Stack<T>::Push(T value) {
    if (top < size)
        stack[top++] = value;
}
```

Other Features
- Operator overloading
  - Common example is to perform input/output
  - “Complex operator *(Complex c1, Complex c2)"
  - Precedence rules are the same
- Function overloading
  - Different methods with different number/type of arguments
  - Method to be invoked is determined based on number of arguments used
  - Easy to make mistakes
- Exceptions
  - Methods can return errors
  - Allow exceptions to be handled by calling methods

Programming Guidelines
- Name declarations:
  - Pick some style and stick to it
  - For e.g., classes have names that begin with capital letters (Stack)
- Minimize the use of global variables
- Minimize the use of global functions
- Develop the interface first
  - Implementation should fall out of the interface definition
  - Develop and test each module separately
- Try to stay away from the complex features of the language

Java vs. C++
- Java is a better language
  - Smaller
  - Real type safety
  - Garbage collection
  - Single inheritance
  - Differentiates between using inheritance for sharing as opposed to using inheritance for interface definition
  - Lots of runtime checks to help development
  - Concrete specification of how things execute
- Performance is a worry
  - Debugging and development tools trying to catch up with implementation and use