Administrivia
Toolchain
In Windows, people usually use an IDE.
In the Linux world, IDEs are available, but command line toolchain use is common.
In all cases the same steps are followed!
The only difference is how much is automated.
And as it turns out, we automate a lot in Linux, too.
But first, let’s talk about how we actually get from code to executable.
C/C++ Source code

Preprocessor

Translation Unit

Compiler

Assembly code

Assembler

Object code

Linker

Executable (or library)
Text editor (nano, vi, emacs)
Compiler
GCC
g++
(preprocessor is included here!)
Preprocessor
#include
(plus include guards)
This prepares the source code file for compilation
The prepared file is called a “translation unit”
The `#includes` let the compiler know what functions and classes exist in files that will be linked later.
This is why include guards are important! C++ only lets you define something once — so there can only be one definition in a translation unit!
Assembler as
Linker

ld
Debugger
gdb
make
PA1.2 stuff
void Entity::dumpObject()
{
    cout << "Entity:" << endl;
    dumpObjectData();
}

void Item::dumpObject()
{
    cout << "Item:" << endl;
    dumpObjectData();
}
void Entity::writeFragment(ostream & output)
{
    output << "<Entity>" << endl;

    writeDataAsFragment(output);

    output << "</Entity>" << endl;
}

void Item::writeFragment(ostream & output)
{
    output << "<Item>" << endl;

    writeDataAsFragment(output);

    output << "</Item>" << endl;
}
void Entity::dumpObjectData()
{
    cout << "    Name: " << getName() << endl
    << "    DisplayChar: " << getDisplayChar() << endl;
}

void Item::dumpObjectData()
{
    Entity::dumpObjectData();

    cout << "    Value: " << getValue() << endl
    << "    Weight: " << getWeight() << endl
    << "    Quantity: " << getQuantity() << endl;
}
void dumpObjects(vector<XMLSerializable*> & vObjects)
{
    for (int i = 0; i < vObjects.size(); i++)
    {
        vObjects[i]->dumpObject();
        cout << endl;
    }
}
for( vector<XMLSerializable>::iterator it = vObjects.begin();
    it != vObjects.end();
    it++ )
{
    (*it)->dumpObject();
    cout << endl;
}
auto keyword
Used to declare a variable. It declares the variable to be the type of the expression assigned to the variable.
Everything is an expression. Expressions have both type and value.
for (vector<XMLSerializable*>::iterator it = vObjects.begin();
    it != vObjects.end();
    it++)
{
    (*it)->dumpObject();
    cout << endl;
}
But this is still clunkier than we’d like
But there’s another C++11 feature — range based for
for( XMLSerializable * pObject : vObjects )
{
    pObject->dumpObject();
    cout << endl;
}

for( auto pObject : vObjects )
{
    pObject->dumpObject();
    cout << endl;
}
void outputXML(vector<XMLSerializable*> & vObjects, 
   ostream & output)
{
   output << "<?xml version="1.0" encoding="utf-8">"
      << endl
      << "<World>"
      << endl;

   for (XMLSerializable * pObject : vObjects)
   {
      pObject->writeFragment(output);
   }

   output << "</World>" << endl;
}