Typical C++ Environment

Phase 1: Programmer creates program in the editor and stores it on disk

Phase 2: Preprocessor program processes the code – modify code to preprocessor directives

Phase 3: Compiler creates object code and stores it on disk – machine language code

Phase 4: Linker links the object code with the libraries, creates an executable file and stores it on disk. – link the object code with the code from other libraries to produce an executable

Phase 5: Loader puts program in memory. – Place the code in memory, along with additional components from the shared libraries used by the program.

Phase 6: CPU takes each instruction executes it, possibly storing new data values as the program executes. – executes the program one instruction at a time

Compiling and running a program

Preprocessing: the first pass of any C/C++ compilation. It processes include-files, conditional compilation instructions and macros.

Compilation: the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.

Assembly: the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.

Linking: the final stage of compilation. Combines one or more object files or libraries as input to produce a single executable file. Resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses.

Sections in the executable file

. text – This section contains the executable instruction codes and is shared among every process running the same binary. Most effected by optimization.

.bss – Stands for ‘Block Started by Symbol’. It holds un initialized global and static variables. Since the BSS only holds variables that don’t have any values yet, it doesn’t actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS doesn’t take up any actual space in the object file.

.data – Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.

.rdata – Also known as .rodata(read-only data) section. This contains constants and string literals.

.reloc – Stores the information required for relocating the image while loading

Symbol table: Symbol table holds information needed to locate and relocate a program’s symbolic definitions and references.

A reference is an *alias*, it defines an alternative name for an object.

class Book //CLASS EXAMPLE

{

private:

double b\_price;

string b\_title;

string b\_author;

public:

Book( string title );

Book( string author, string title, double price );

void setPrice( double new\_price );

double getPrice(); 🡨 Accessor

void setAuthor( string new\_author );

string getAuthor(); 🡨 Accessor

void setTitle( string new\_title );

string getTitle(); 🡨 Accessor

};

void f(); //OVERLOADING EXAMPLE

void f(int);

void f(int, int);

void f(double, double = 3.14);

f(5.6);  // calls void f(double, double)

You can also declare a function argument to be const, in which case the compiler would ensure that it is not mutated.

static data members have to be initialized outside of class declaration, in class implementation:

class Foo { static int a;  };

int Foo::a = 1;

Sales\_data(const std::string &s, unsigned n, double p)     :  bookNo(s), units\_sold(n), revenue(p\*n)  { }

A constructor initializer list specifies initial values for one or more data members of the object being created.

Double getData() const; The const declaration is saying that this member function does not mutate the Data object.

A default constructor will be created if you don’t make one. A default constructor has no arguments.

class Bar  { private: Foo y;

public: Bar() { y.x = 5; } }; //Bar() is the default constructor

sizeof(data type) – determines the size in bytes of a variable or data type

alignof(data type) – compile time operator that returns the alignment requirement, in bytes, of a variable or data type. Will return at what x-byte boundaries an object can be allocated at.

new takes an argument the type of object and returns a pointer to a dynamically allocated object of that type. Delte deallocates the memory.

pointer = new type; or pointer = new type [number of elements]; each new must match it’s corresponding delete. delete pointer or delete[] pointer.

A shallow copy uses default copy constructors, a deep copy duplicates the object so that the destination gets its own local copy.

An assignment operator copies to existing objects and the copy constructor copies to newly created objects.

A destructor deallocates memory for a class

Copy constructor: MyString(MyString & arg);

Assignment operator: MyString & operator=(const MyString & rhs);

Destructor: ~MyString()

Private members are only accessible withing the class defining them. Protected members are accessible in the class that defines them and in classes that inherit from them.

Using Derived classes:

class Foo

{

  public:

      Foo()

        { cout << "Foo's constructor\n"; }

};

class Bar : public Foo

{

  public:

      Bar()

        { cout << "Bar's constructor\n"; }

};

int main()  {

        Bar x;

}

If a function expects an Employee as a parameter, you can pass a Manager instead. − Similarly, you can assign a pointer to a Manager to a variable whose type is pointer to Employee.

Employee \*empl\_p;

Employee empl(“John Smith”, 25.0);

Manager mgr(“Jeff Gordon”, 1200.0, true);

empl\_p = &mgr;

cout << “Name: ” < empl\_p->getName() << endl;

 cout “Salaried: “ <<getSalaried(); <- error class employee has no member named getSalaried, compiler only knows that empl\_p pointer holds the address of an Employee

By default in C++, the method that runs is determined by the type of expression from which the method is selected. To have the method instead be selected at run time, declare the member function of the base class virtual. Objects with a virtual function definition will then store a pointer to a virtual table that defines the mapping for that method for a given instance. Must be declared virtual in the base class and are customarily declared virtual in the derived classes. Must the same number, order and type of arguments. Must have same return type. Every class that uses virtual functions is given its own virtual table which is a static array set up by the compiler. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most derived function accessible by that class. The compiler also adds a hidden pointer \*\_vptr which is set when a class instance is created so that it points to the virtual table for that class.

Given the classes on the previous slide, consider this code:

int main() {

    D1 obj;

    Base \*ptr = &obj;

    ptr­>function1();

}

Because obj is a D1 object, obj has it’s \*\_\_vptr set to the D1 virtual table. Because ptr is a base pointer, it only points to the Base portion of obj. But \*\_\_vptr is in the Base portion of the class, so ptr has access to this pointer. ptr->\_\_vptr points to the D1 virtual table! Consequently, even though ptr is of type Base, it still has access to D1′s virtual table. So what happens when we try to call ptr->function1()? First, we detect that function1() is a virtual function. Second, ptr->\_\_vptr is used to get to D1′s virtual table. Third, we look up which version of function1() to call in D1′s virtual table. This has been set to D1::function1(). Therefore, ptr->function1() resolves to D1::function1()!