Virtual functions are the only thing that requires any trickiness. They are implemented by creating a dispatch table ("vtable") for the class and putting a pointer to that table in the data of the object. A derived class has a different table. In that table, functions defined in the parent come first, though some of the pointers point to overridable functions. You could put the whole vtable in the object itself. That would save a little time, but potentially waste a lot of space.

The C++ philosophy is to avoid run-time overhead wherever possible. (Sort of the legacy from C.) That's why non-virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The vtable for each class has virtual functions. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

The compiler needs the default code, the compiler generates a static, empty code pointer. It dereferences this pointer to find the specific forwarding routine that accepts the 'this' parameter, adds back in the correction that the regular calling sequence just offset out of the first word of the object and subtracts it from the offset back from the object support to the object view. (Likewise for the class itself.)

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

Each interface begins with a vtable:

```
   virtual char *t( ...)
vtable
```

The virtual functions require no extra space at run-time; the compiler can pass to other methods. The C++ philosophy is to avoid run-time support. Of course you only have a vtable...

The vtable technique is to put a pointer to the type info at the beginning of the object in which that pointer appears:

```
for (i = 0; i < sizeof(vtable); i++)
    *(char*)p += sizeof(void*);
```

The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```

Note that if you can query the type of an object, then you need to be passed a pointer to the vtable. That's why you can't do a dynamic_cast in C++. The vtable includes the (non-virtual) methods of widget and named_widget, as well as all existing uses of that interface.

```
   virtual char *t( ...)
vtable
   virtual double s( ...)
```