

# Advanced C++



declare pointers:

```
Rectangle *pointerToRect = &aRect;
```

pointerToRect is a *pointer to Rectangle*  
initially points to aRect

dereference pointers:

```
(*pointerToRect).size, pointerToRect->size,
```

```
*pointerToRect = anotherRect
```

all modify pointed-to object (aRect)

&

declare references:

```
Rectangle &refToRect = aRect;  
void print(const Rectangle &theRect);
```

refToRect, theRect are *references to Rectangle*

address-of:

```
pointerToRect = &refToRect;
```

&value is the address of value

## recall: reference v pointer

pointer — explicitly derference, can reassign

reference — “bound” to object on creation, always refers to it

typical implementation of both in asm: pointer

# typed pointers

```
double Z = 26.0;  
int *pointerToInt = &Z;    // ERROR
```

“cannot convert 'double\*' to 'int\*' in initialization”

C++ cares about type (but just addresses in assembly)

# dereference example (1)

```
int n = 26;  
int *somePointer = &n;  
  
cout << somePointer << endl;  
cout << *somePointer << endl;
```

# dereference example (1)

```
int n = 26;  
int *somePointer = &n;
```

```
cout << somePointer << endl;  
cout << *somePointer << endl;
```

---

example output: (address will vary...)

0x7fff35fc3fe4

26

## dereference example (2)

```
int n = 26;  
int *somePointer = &n;  
*somePointer = 45;  
  
cout << somePointer << endl;  
cout << *somePointer << endl;
```



## dereference example (2)

```
int n = 26;  
int *somePointer = &n;  
*somePointer = 45;
```

```
cout << somePointer << endl;  
cout << *somePointer << endl;
```

---

example output: (address will vary...)

```
0x7fff35fc3fe4  
45
```

## dereference example (3)

```
ListNode *ptr1, *ptr2;  
ptr1 = new ListNode;  
ptr2 = new ListNode  
  
bool result1 = (ptr1 == ptr2);  
bool result2 = (*ptr1 == *ptr2);
```

## dereference example (3)

```
ListNode *ptr1, *ptr2;  
ptr1 = new ListNode;  
ptr2 = new ListNode
```

```
bool result1 = (ptr1 == ptr2);  
bool result2 = (*ptr1 == *ptr2);
```

---

result1 definitely false (different addresses)

result2 probably true (depends on `ListNode::operator==`)

# reference example

```
int y = 5;  
int &x = y;  
cout << x << endl;  
cout << &x << endl;  
cout << &y << endl;  
x = 15;  
cout << y << endl;
```

# reference example

```
int y = 5;  
int &x = y;  
cout << x << endl;  
cout << &x << endl;  
cout << &y << endl;  
x = 15;  
cout << y << endl;
```

---

example output (address will vary...)

```
5  
0x7ffeeda220d4  
0x7ffeeda220d4  
15  
can't change address stored in x
```

# pointers to pointers

```
int main() {  
    Animal cow;  
    Animal* cowPtr1 = &cow;  
    Animal** cowPtr2(&cowPtr1);  
    Animal*** cowPtr3 = &cowPtr2;  
    ...  
}
```

---

# pointers to pointers

```
int main() {  
    Animal cow;  
    Animal* cowPtr1 = &cow;  
    Animal** cowPtr2(&cowPtr1);  
    Animal*** cowPtr3 = &cowPtr2;  
    ...  
}
```

---

cow = Animal

cowPtr1 = pointer to Animal

cowPtr2 = pointer to (pointer to Animal)

cowPtr3 = pointer to pointer to (pointer to Animal)

# example memory layout

## memory

address value

...	...
0x10000	0x500
0x10008	0x10000
0x10010	0x10008
0x10018	0x10010
...	...

COW

cowPtr1

cowPtr2

cowPtr3



## ref to pointer v pointer to pointer

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
void insert(TreeNode** n, int value) {  
    if (*n == NULL)  
        *n = new TreeNode(value);  
    else if (value < n->value)  
        insert(&(n->left), value);  
    else if (value > n->value)  
        insert(&(n->right), value);  
}
```

## ref to pointer v pointer to pointer

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
void insert(TreeNode** n, int value) {  
    if (*n == NULL)  
        *n = new TreeNode(value);  
    else if (value < n->value)  
        insert(&(n->left), value);  
    else if (value > n->value)  
        insert(&(n->right), value);  
}
```

## ref to pointer v pointer to pointer

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
void insert(TreeNode** n, int value) {  
    if (*n == NULL)  
        *n = new TreeNode(value);  
    else if (value < n->value)  
        insert(&(n->left), value);  
    else if (value > n->value)  
        insert(&(n->right), value);  
}
```

## by ref versus by value

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
TreeNode *insert(TreeNode* n, int value) {  
    if (n == NULL)  
        return new TreeNode(value);  
    else if (value < n->value) {  
        n->left = insert(n->left, value);  
        return n;  
    } else if (value > n->value) {  
        n->right = insert(n->right, value);  
        return n;  
    }  
}
```

# by ref versus by value

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
TreeNode *insert(TreeNode* n, int value) {  
    if (n == NULL)  
        return new TreeNode(value);  
    else if (value < n->value) {  
        n->left = insert(n->left, value);  
        return n;  
    } else if (value > n->value) {  
        n->right = insert(n->right, value);  
        return n;  
    }  
}
```

# by ref versus by value

```
void insert(TreeNode*& n, int value) {  
    if (n == NULL)  
        n = new TreeNode(value);  
    else if (value < n->value)  
        insert(n->left, value);  
    else if (value > n->value)  
        insert(n->right, value);  
}
```

---

```
TreeNode *insert(TreeNode* n, int value) {  
    if (n == NULL)  
        return new TreeNode(value);  
    else if (value < n->value) {  
        n->left = insert(n->left, value);  
        return n;  
    } else if (value > n->value) {  
        n->right = insert(n->right, value);  
        return n;  
    }  
}
```

# several memory allocation problems

BROKEN:

```
void someFunc(int *somePointer) {  
    int someVal(3);  
    somePointer = &someVal;  
}  
  
int main() {  
    int *firstPointer;  
    someFunc(firstPointer);  
    cout << *firstPointer << endl;  
    return 0;  
}
```

---

# several memory allocation problems

BROKEN:

```
void someFunc(int *somePointer) {  
    int someVal(3);  
    somePointer = &someVal;  
}
```

```
int main() {  
    int *firstPointer;  
    someFunc(firstPointer);  
    cout << *firstPointer << endl;  
    return 0;  
}
```

---

pointer to deallocated memory — need new



# several memory allocation problems

BROKEN:

```
void someFunc(int *somePointer) {  
    int someVal(3);  
    somePointer = &someVal;  
}  
  
int main() {  
    int *firstPointer;  
    someFunc(firstPointer);  
    cout << *firstPointer << endl;  
    return 0;  
}
```

---

pointer to deallocated memory — need new

pass by value, not by reference

# several memory allocation problems (fixed?)

```
void someFunc(int *&somePointer) {  
    somePointer = new int(3);  
}
```

```
int main() {  
    int *firstPointer;  
    someFunc(firstPointer);  
    cout << *firstPointer << endl;  
    return 0;  
}
```

# several memory allocation problems

BROKEN:

```
void someFunc() {  
    double *aliasPointer;  
    aliasPointer = new double(6.27);  
    cout << *aliasPointer << endl;  
}
```

---

# several memory allocation problems

BROKEN:

```
void someFunc() {  
    double *aliasPointer;  
    aliasPointer = new double(6.27);  
    cout << *aliasPointer << endl;  
}
```

---

memory leak — never deleted

# several memory allocation problems

BROKEN:

```
void someFunc() {  
    double duration = 3.14;  
    {  
        double * somePtr;  
        {  
            somePtr = &duration;  
        }  
    }  
    cout << *somePtr << endl;  
    return 0;  
}
```

---

# several memory allocation problems

BROKEN:

```
void someFunc() {  
    double duration = 3.14;  
    {  
        double * somePtr;  
        {  
            somePtr = &duration;  
        }  
    }  
    cout << *somePtr << endl;  
    return 0;  
}
```

---

syntax error: somePtr no longer exists

# several memory allocation problems

BROKEN:

```
int main() {  
    int * anotherPtr;  
    {  
        int someVal(8);  
        cout << *anotherPtr << endl;  
        anotherPtr = &someVal;  
    }  
  
    return 0;  
}
```

---

# several memory allocation problems

BROKEN:

```
int main() {  
    int * anotherPtr;  
    {  
        int someVal(8);  
        cout << *anotherPtr << endl;  
        anotherPtr = &someVal;  
    }  
  
    return 0;  
}
```

---

undefined behavior: accessing uninitialized pointer



# several memory allocation problems

BROKEN:

```
void someFunc(int *somePointer) {  
    int someVal(12);  
    {  
        int anotherVal(16);  
        somePointer = &anotherVal;  
    }  
}  
  
int main() {  
    int * yetAnotherPtr;  
    someFunc(yetAnotherPtr);  
    cout << *yetAnotherPtr << endl;  
    return 0;  
}
```

## a correct example

```
int main() {  
    float * somePtr;  
    somePtr = new float(3.14);  
    cout << *somePtr << endl;  
    delete somePtr;  
    return 0;  
}
```

## a correct example

```
void someFunc() {  
    int *aliasPtr;  
    aliasPtr = new int(25);  
    cout << *aliasPtr << endl;  
}
```

```
int main() {  
    int * somePtr;  
    somePtr = new int(3);  
    someFunc();  
    cout << *somePtr << endl;  
    return 0;  
}
```

## a correct example

```
void someFunc() {  
    int *aliasPtr;  
    aliasPtr = new int(25);  
    cout << *aliasPtr << endl;  
}
```

```
int main() {  
    int * somePtr;  
    somePtr = new int(3);  
    someFunc();  
    cout << *somePtr << endl;  
    return 0;  
}
```

memory leaks

# C++ inheritance example (1)

```
class Name {  
public:  
    Name();  
    ~Name();  
    void setName(const string &name);  
    void print() {  
        cout << myName << endl;  
    }  
private:  
    string myName;  
};
```

## C++ inheritance example (2)

```
class Contact : public Name {  
public:  
    Contact() {  
        myAddress = ""  
    }  
  
    ~Contact() {  
    }  
  
    void setAddress(const string &address) {  
        myAddress = address;  
    }  
  
    void print() {  
        Name::print();  
        cout << myAddress << endl;  
    }  
private:  
    string myAddress;  
}
```

## C++ inheritance example (2)

```
class Contact : public Name {  
public:  
    Contact() {  
        myAddress = ""  
    }  
  
    ~Contact() {  
    }  
  
    void setAddress(const string& address) {  
        myAddress = address;  
    }  
  
    void print() {  
        Name::print();  
        cout << myAddress << endl;  
    }  
private:  
    string myAddress;  
}
```

```
public class Contact  
    extends Name {  
    ...  
    void print() {  
        super.print();  
    }  
    ...  
}
```

## C++ inheritance example (2)

```
class Contact : public Name {  
public:  
    Contact() {  
        myAddress = ""  
    }  
  
    ~Contact() {  
    }  
  
    void setAddress(const string& address) {  
        myAddress = address;  
    }  
  
    void print() {  
        Name::print();  
        cout << myAddress << endl;  
    }  
private:  
    string myAddress;  
}
```

```
public class Contact  
    extends Name {  
    ...  
    void print() {  
        super.print();  
    }  
    ...  
}
```



## contact usage (1)

```
int main(void) {  
    Contact c;  
    c.SetName("John_Doe");  
    c.SetAddress("009_Olsson_Hall");  
    c.print();  
}
```

## contact usage (2)

```
int main(void) {  
    Contact c;  
    Name &r = c;  
    r.SetName("John_Doe");  
    // or:  
    Name *p = &c;  
    p->SetName("John_Doe");  
    c.SetAddress("009_Olsson_Hall");  
    c.print();  
}
```

# memory layout

address value

...	...	
0x10000	"John Doe"	c.myName
0x10020	"009 Olsson Hall"	c.myAddress
0x10040	0x10000	r (ref. to c as Name)
...	...	

} Contact c

# memory layout

address value

...	...	
0x10000	"John Doe"	c.myName
0x10020	"009 Olsson Hall"	c.myAddress
0x10040	0x10000	r (ref. to c as Name)
...	...	

} as Name  
Contact c

# inheritance in C++

Contact is child of parent Name

has member variables, functions of parent

...with same layout in memory

- parent's methods work without changing assembly
- can get reference/pointer to Name from Contact

add new member functions/variables

# inheritance and constructors, etc.

```
class Parent {  
public:  
    Parent() { cout << "Parent()\n"; }  
    ~Parent() { cout << "~Parent()\n"; }  
};  
class Child : public Parent {  
public:  
    Child() { cout << "Child()\n"; }  
    ~Child() { cout << "~Child()\n"; }  
};  
int main() {  
    Child var;  
    cout << "in_main()\n";  
}
```

---

Parent()  
Child()  
in main()  
~Child()  
~Parent()

# construction/destruction order

parent part constructed first

then child

child part destroyed first

then parent

# arguments to parent constructors?

```
class Parent {  
public:  
    Parent(int x) { cout << "Parent(" << x << ")\n"; }  
    ~Parent() { cout << "~Parent()\n"; }  
};  
class Child : public Parent {  
public:  
    Child(int x) : Parent(x + 1) { cout << "Child(" << x << ")\n"; }  
    ~Child() { cout << "~Child()\n"; }  
};  
int main() {  
    Child var(100);  
    cout << "in_main()\n";  
}
```

---

Parent(101)  
Child(100)  
in main()  
~Child()  
~Parent()



# multiple inheritance

```
class Sphere : public Shape, public Comparable,  
               public Serializable {  
    // ...  
};
```

# multiple inheritance

```
class Sphere : public Shape, public Comparable,  
               public Serializable {  
    // ...  
};
```

*sort of* like this Java code:

```
public class Sphere extends Shape  
    implements Comparable, Serializable {  
    // ..  
}
```

but — Comparable, Serializable might have their own member variables and implemented methods

# C++ defaults to static dispatch (1)

```
class Parent {  
public:  
    void print() { cout << "Parent::print()\n"; }  
};  
class Child : public Parent {  
public:  
    void print() { cout << "Child::print()\n"; }  
};  
Parent* getParent() { return new Child; }  
int main() {  
    Parent *p = getParent();  
    p->print();  
    delete p;  
}
```

output:

Parent::print()

# C++ defaults to static dispatch (1)

```
class Parent {  
public:  
    void print() { cout << "Parent::print()\n"; }  
};  
class Child : public Parent {  
public:  
    void print() { cout << "Child::print()\n"; }  
};  
Parent* getParent() { return new Child; }  
int main() {  
    Parent *p = getParent();  
    p->print();  
    delete p;  
}
```

output:

Parent::print()

# static versus dynamic dispatch

static dispatch — call method based on compile-time type

dynamic dispatch — call method based on run-time type

## C++ defaults to static dispatch (2)

```
class Parent {  
public:  
    Parent() {cout << "Parent()\n"; }  
    ~Parent() { cout << "~Parent()\n"; }  
};  
class Child : public Parent {  
public:  
    Child() { cout << "Child()\n"; }  
    ~Child() { cout << "~Child()\n"; }  
};  
Parent* getParent() { return new Child; }  
int main() {  
    Parent *p = getParent();  
    delete p;  
}
```

output (*probably*):  
Parent()  
Child()  
~Parent()

## C++ defaults to static dispatch (2)

```
class Parent {
public:
    Parent() {cout << "Parent()\n"; }
    ~Parent() { cout << "~Parent()\n"; }
};
class Child : public Parent {
public:
    Child() { cout << "Child()\n"; }
    ~Child() { cout << "~Child()\n"; }
};
Parent* getParent() { return new Child; }
int main() {
    Parent *p = getParent();
    delete p;
}
```

output (*probably*):  
Parent()  
Child()  
~Parent()

# virtual: ask for dynamic dispatch

`virtual` keyword — ask for dynamic dispatch

not default — because slower:

- static dispatch: just a function call

- dynamic dispatch: lookup correct function first!



# virtual methods (1)

```
class Parent {  
public:  
    virtual void print() { cout << "Parent::print()\n"; }  
};  
class Child : public Parent {  
public:  
    void print() { cout << "Child::print()\n"; }  
};  
Parent* getParent() { return new Child; }  
int main() {  
    Parent *p = getParent();  
    p->print();  
    delete p;  
}
```

output:

Child::print()

# virtual methods (1)

```
class Parent {  
public:  
    Parent() {cout << "Parent()\n"; }  
    virtual ~Parent() { cout << "~Parent()\n"; }  
};  
class Child : public Parent {  
public:  
    Child() { cout << "Child()\n"; }  
    ~Child() { cout << "~Child()\n"; }  
};  
Parent* getParent() { return new Child; }  
int main() {  
    Parent *p = getParent();  
    delete p;  
}
```

output:

```
Parent()  
Child()  
~Child()  
~Parent()
```

# virtual destructors

**required** if you call `delete` on a base-class pointer  
(but it's actually an instance of the subclass)

(even if destructor doesn't do anything)

compiler **might use destructor to know how much memory to free**

C++ standard quote: "If the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined."

# a dynamic call

```
class Parent {
public:
    virtual void foo() { ... }
    ...
};
class Child : public Parent {
    virtual void foo() { ... }
    ...
};
Parent *get(); // return Parent or Child
int main() {
    Parent *p = get();
    p->foo();
}
```

What does assembly for main look like?

Could call Parent::foo or Child::foo

# dynamic call: assembly

```
// Parent *p (RAX) = get();  
call get  
mov rcx, [rax + 0] // rcx ← "VTable" address  
mov rdi, rax // rdi (this arg) ← p  
call [rcx + 0] // call what rcx points to
```

Parent object

vtable pointer
Parent variable 1
Parent variable 2
...

Parent vtable

Parent::foo addr.
...
Parent::~~Parent addr.

Child object

vtable pointer
Parent variable 1
Parent variable 2
...
Child variable 1
...

Child vtable

Child::foo addr.
...
Child::~~Child addr.

# dynamic call: assembly

```
// Parent *p (RAX) = get();  
call get  
mov rcx, [rax + 0] // rcx ← "VTable" address  
mov rdi, rax // rdi (this arg) ← p  
call [rcx + 0] // call what rcx points to
```

Parent object

vtable pointer
Parent variable 1
Parent variable 2
...

Parent vtable

Parent::foo addr.
...
Parent::~~Parent addr.

Child object

vtable pointer
Parent variable 1
Parent variable 2
...
Child variable 1
...

Child vtable

Child::foo addr.
...
Child::~~Child addr.

# dynamic call: assembly

```
// Parent *p (RAX) = get();  
call get  
mov rcx, [rax + 0] // rcx ← "VTable" address  
mov rdi, rax // rdi (this arg) ← p  
call [rcx + 0] // call what rcx points to
```

Parent object

vtable pointer
Parent variable 1
Parent variable 2
...

Parent vtable

Parent::foo addr.
...
Parent::~~Parent addr.

Child object

vtable pointer
Parent variable 1
Parent variable 2
...
Child variable 1
...

Child vtable

Child::foo addr.
...
Child::~~Child addr.

# vtables during construction

vtable set by constructor call

constructor call order:

- parent vtable set first
- then Parent() constructor run
- overwritten with child vtable
- then Child() constructor run
- ...

rule: never call method before it's type's constructor



# pure virtual member functions

```
class Shape {  
public:  
    virtual void draw() = 0;  
}
```

= 0 — no implementation!

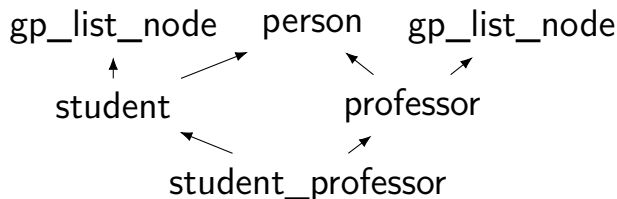
“pure virtual function/method”

**must be overridden** to create object  
otherwise, “abstract class”

≈ abstract in Java

only abstract methods ≈ Java interface

# diamonds or duplicates



replicated parents (`gp_list_node`)

one copy each time inherited

separate lists of students, professors

shared parents (`person`)

one copy of attributes (name?) for person

# C++ default: replicated inheritance

```
class Parent { public:  
    int value;  
};  
class A : public Parent {};  
class B : public Parent {};  
class C : public A, public B {};  
  
int main() {  
    C c;  
    A& as_a = c; B& as_b = c;  
    as_a.value = 1; as_b.value = 2;  
    cout << as_a.value << "_" << as_b.value << endl;  
}
```

---

output: 1 2 (two copies of value)

# virtual inheritance: one copy

```
class Parent { public:  
    int value;  
};  
class A : public virtual Parent {};  
class B : public virtual Parent {};  
class C : public A, public B {};  
  
int main() {  
    C c;  
    A& as_a = c; B& as_b = c;  
    as_a.value = 1; as_b.value = 2;  
    cout << as_a.value << "_" << as_b.value << endl;  
}
```

---

output: 2 2 (as\_a.value same as as\_b.value)

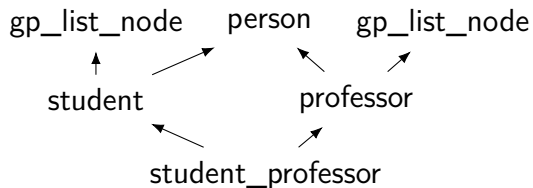
# virtual inheritance: one copy

```
class Parent { public:  
    int value;  
};  
class A : public virtual Parent {};  
class B : public virtual Parent {};  
class C : public A, public B {};  
  
int main() {  
    C c;  
    A& as_a = c; B& as_b = c;  
    as_a.value = 1; as_b.value = 2;  
    cout << as_a.value << "_" << as_b.value << endl;  
}
```

---

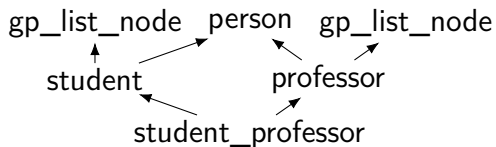
output: 2 2 (as\_a.value same as as\_b.value)

# declaring a mix



```
class student: public virtual person, public gp_list_node {...};  
class professor: public virtual person, public gp_list_node {...};  
class student_professor: public professor, public student {...};
```

# duplicate layout



```
gp_list_node &getStudentList(student_professor &p) {
    return (gp_list_node &) (student &) p;
}
gp_list_node &getProfessorList(student_professor &p) {
    return (gp_list_node &) (professor &) p;
}
```

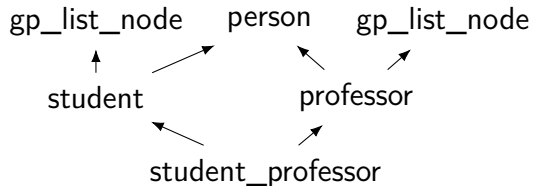
---

example assembly:

```
getStudentList:
    lea rax, [rdi + 8]
    ret
```

```
getProfessorList:
    lea rax, [rdi + 64]
    ret
```

# diamond layout



```
(person&) studentProf ==  
    (person &) (student &) studentProf ==  
        (person &) (professor &) studentProf
```

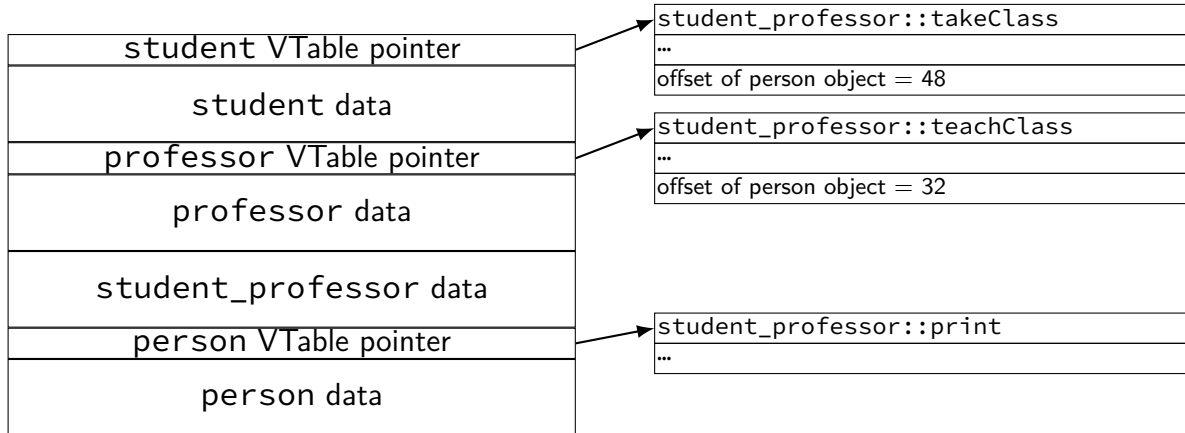
---

casts need more indirection to implement

example: vtable lookup of offset to 'person' fields  
different 'offsets' of object for professor versus  
student\_professor

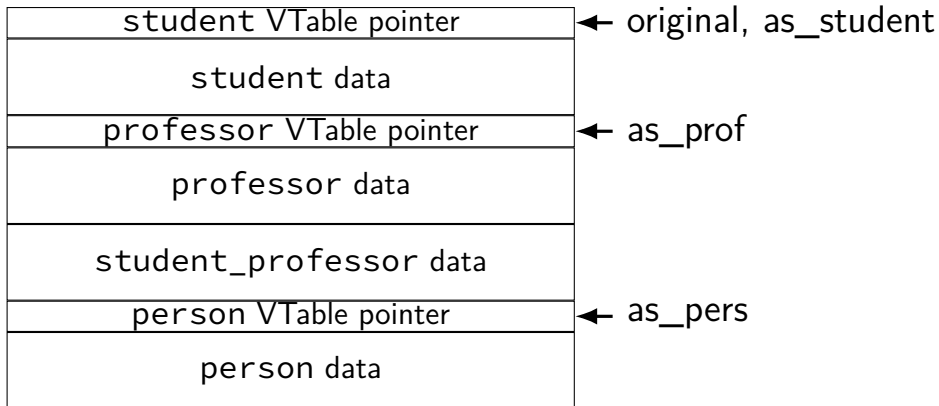


# a possible layout



## a possible layout

```
student_professor *original  
    = new student_professor;  
student *as_student = original;  
professor *as_prof = original;  
person *as_pers = original;
```



## co-variant arrays

```
String[] a = new String[1];  
Object[] aAsObjects = a;  
b[0] = new Integer(1);
```

compiles, but `ArrayStoreException`

not really an array of objects

## non-co-variant containers

```
class Parent {};  
class Child : public Parent {};  
...  
vector<Child*> v;  
  
vector<Parent*> &vAsParent = v;  // DOES NOT COMPILE
```

# C++11 and beyond

C++ standard versions:

1997(C++03): `-std=c++98`

2003 (C++03): `-std=c++03`

August 2011 (C++11): `-std=c++11`

August 2014 (C++14): `-std=c++14`

March 2017 (C++17): `-std=c++17`

??? 2020 (C++20)

# notable C++11 features

move constructors and r-value references

option for moving value from  $x$  to  $y$  without copying

initialization with braces: `Class name{arg1, arg2, ...}`

foreach loops: `for (int &x: some_vector) {...}`

type inference: `auto it = some_vector.iterator();`

return type at end: `auto foo(int x, int y) -> int;`

`nullptr`

(more) smart pointers

# move motivation

```
vector<string> v;  
...  
v.push_back(getBigString());
```

---

C++03: this **makes a copy** of what `getBigString()` returns!  
(`push_back` calls the copy constructor)

# using move constructors

```
string one = "some_contents";  
string two = std::move(one);
```

---

two contains "some contents"

one's contents **unspecified**



# move constructors

```
class DynamicArray {  
public:  
    ...  
    DynamicArray(DynamicArray &&moveFrom) {  
        pointer = moveFrom.pointer;  
        size = moveFrom.size;  
        moveFrom.pointer = NULL;  
        moveFrom.size = 0;  
    }  
private:  
    int *pointer;  
    int size;  
};
```

## using move assignment

```
string one = "some_contents";  
string two = "other_contents";  
two = std::move(one);
```

---

two contains "some contents"

one's contents **unspecified**

# move assignment operators

```
class DynamicArray {
public:
    ...
    DynamicArray &operator=(DynamicArray &&moveFrom) {
        if (pointer != NULL)
            delete[] pointer;
        pointer = moveFrom.pointer;
        size = moveFrom.size;
        moveFrom.pointer = NULL;
        moveFrom.size = 0;
        return *this;
    }
private:
    int *pointer;
    int size;
};
```

# brace-based initialization

can now use {} to initialize objects:

```
// SomeClass()  
SomeClass foo;
```

```
SomeClass foo{};
```

```
// SomeClass(const SomeClass &)  
SomeClass bar(foo);
```

```
SomeClass bar{foo};
```

```
// SomeClass(int, int, int, int)  
SomeClass quux(1, 2, 3, 4);
```

```
SomeClass quux{1, 2, 3, 4};
```

```
// vector<int>(initializer_list<int>)  
/* not supported in C++03 */
```

```
vector<int> v{0, 1, 2, 3, 4, 5};
```

# range-based for loops

```
int array[1000];  
....  
for (int &x : array) {  
    ...  
}
```

---

```
vector<int> v;  
...  
for (int &x : v) {  
    ...  
}
```

# auto

```
vector<int> v;  
auto it = v.begin();  
// instead of:  
vector<int>::iterator it = v.begin();
```

# trailing return types

```
auto foo(int x, int y) -> int { ... }  
// instead of:  
int foo(int x, int y) { ... }
```

# nullptr

nullptr is substitute for 0/NULL

typechecks better

```
int x = nullptr; — ERROR
```

```
int x = NULL; — sets x to 0
```



# unique\_ptr

instead of:

```
class Foo {  
    ...  
    ~Foo() { delete bar; }  
    void set() {  
        if (bar) delete bar;  
        bar = new Bar(...);  
    }  
    Bar *bar;  
};
```

---

```
class Foo {  
    void set() {  
        bar.reset(new Bar(...));  
    }  
    unique_ptr<Bar> bar;  
};
```

# unique\_ptr implementation

```
template <class T> class unique_ptr {  
    ...  
    T& operator->() { return *value; }  
    T& operator*() { return *value; }  
    void reset(T* new_value) {  
        if (value) delete value;  
        value = new_value;  
    }  
    ~unique_ptr() {  
        if (value) delete value;  
    }  
private:  
    T *value;  
}
```

# the smart pointers

`unique_ptr` — “owns” the object

delete object when pointer goes away/changes

`shared_ptr` — keeps a reference count

delete object when last `shared_ptr` goes away

`weak_ptr` — works with `shared_ptr`, but doesn't modify reference count

handles circular references

# miscellaneous C++11/14/17

lambda expressions (closures)

compile-time arithmetic (constexpr)

function attributes

e.g. `[[override]]`: 'give me an error if this isn't overriding something'

compile-time assertions

using C++11, etc.

`clang -std=c++11 (or c++14 ...)`