ADVANCED PROGRAMMING IN C++

Inheritance

Patrick Bader · SS 2023

Inheritance

```
class Parent {
public:
  void foo();
protected:
  int i;
private:
  int x;
};

class Child : public Parent {
public:
  void foo();
  void bar();
private:
  int y;
};
```

```
void Parent::foo() {
  x = 10;
  i = 20;
}
void Child::foo() {
  i = 15;
  y = 25;
}
void Child::bar() {
  foo();
  i = 7;
  y = 5;
}
```

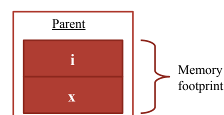
Inheritance - Assignment

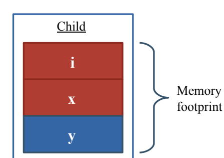
```
class Parent {
public:
  void foo();
protected:
  int i;
private:
  int x;
};

class Child : public Parent {
public:
  void foo();
  void bar();
private:
  int y;
};
```

```
int main() {
  Parent p;
  p.foo() // Parent::foo()

  Child c;
  c.foo(); // Child::foo()
  c.bar(); // Child::bar()

  p = c;   // Assign c to p
  p.foo(); // Parent::foo()

  return 0;
}
```

Inheritance - Assignment

```
class Parent {
public:
  void foo();
protected:
  int i;
private:
  int x;
};

class Child : public Parent {
public:
  void foo();
  void bar();
private:
  int y;
};
```

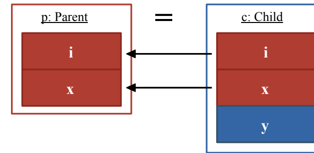




Inheritance - Assignment

```
int main() {
  Parent p;
  p.foo() // Parent::foo()

  Child c;
  c.foo(); // Child::foo()
  c.bar(); // Child::bar()

  p = c;   // Assign c to p
  p.foo(); // Parent::foo()

  return 0;
}
```

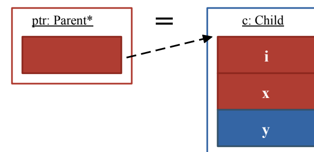
Assignment **only copies `Parent` part** as there is no space available for `Child` members in `p`.



Inheritance - Pointer Assignment

```
int main() {
  Parent *ptr;

  Child c;

  // Assign address of c to ptr
  ptr = &c;

  ptr->foo(); // still Parent::foo()



  return 0;
}
```

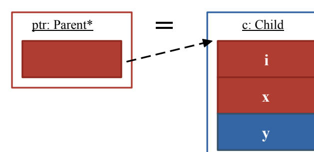
Assignment **copies address of `Child` object** to `ptr`.



Inheritance - Pointer Assignment

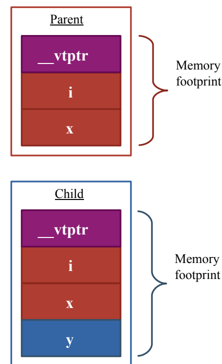
```
int main() {
  Parent *ptr;

  Child c;

  // Assign address of c to ptr
  ptr = &c;

  ptr->foo(); // still Parent::foo()

  // Downcast to Child pointer
  static_cast<Child *>(ptr)->foo();

  return 0;
}
```

Assignment **copies address of `Child` object** to `ptr`.



Inheritance - Dynamic Dispatch

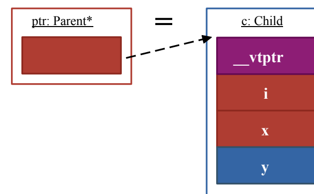
```
class Parent {
public:
  virtual void foo();
protected:
  int i;
private:
  int x;
};

class Child : public Parent {
public:
  void foo() override;
  void bar();
private:
  int y;
};
```



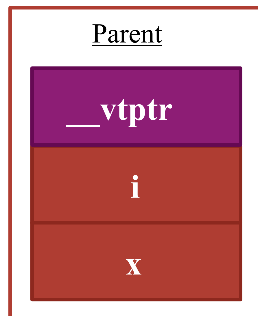
Inheritance - Dynamic Dispatch

```
int main() {
  Parent *ptr;

  Child c;

  // Assign address of c to ptr
  ptr = &c;

  ptr->foo(); // Child::foo()



  return 0;
}
```

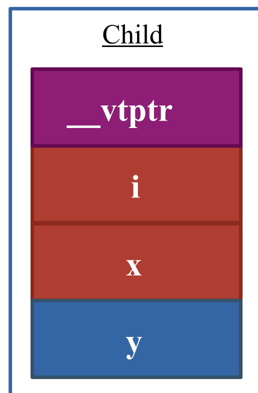
Assignment **copies address of `Child` object** to `ptr`.



Inheritance - Dynamic Dispatch

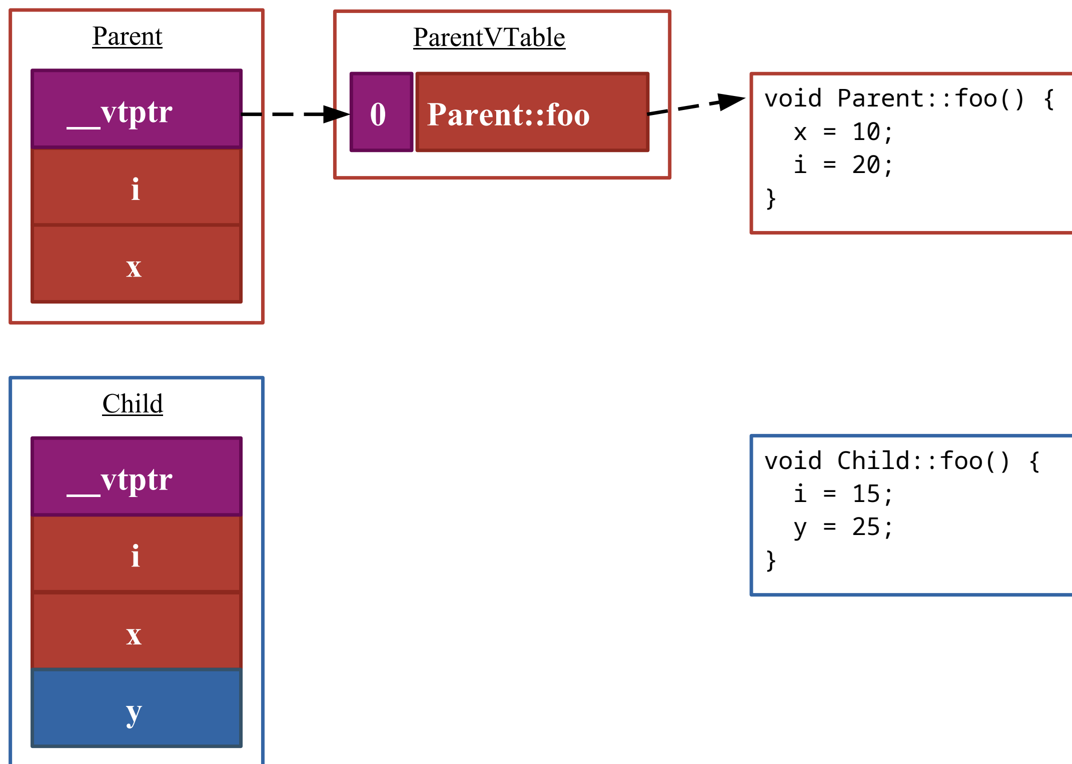



```
void Parent::foo() {
  x = 10;
  i = 20;
}
```

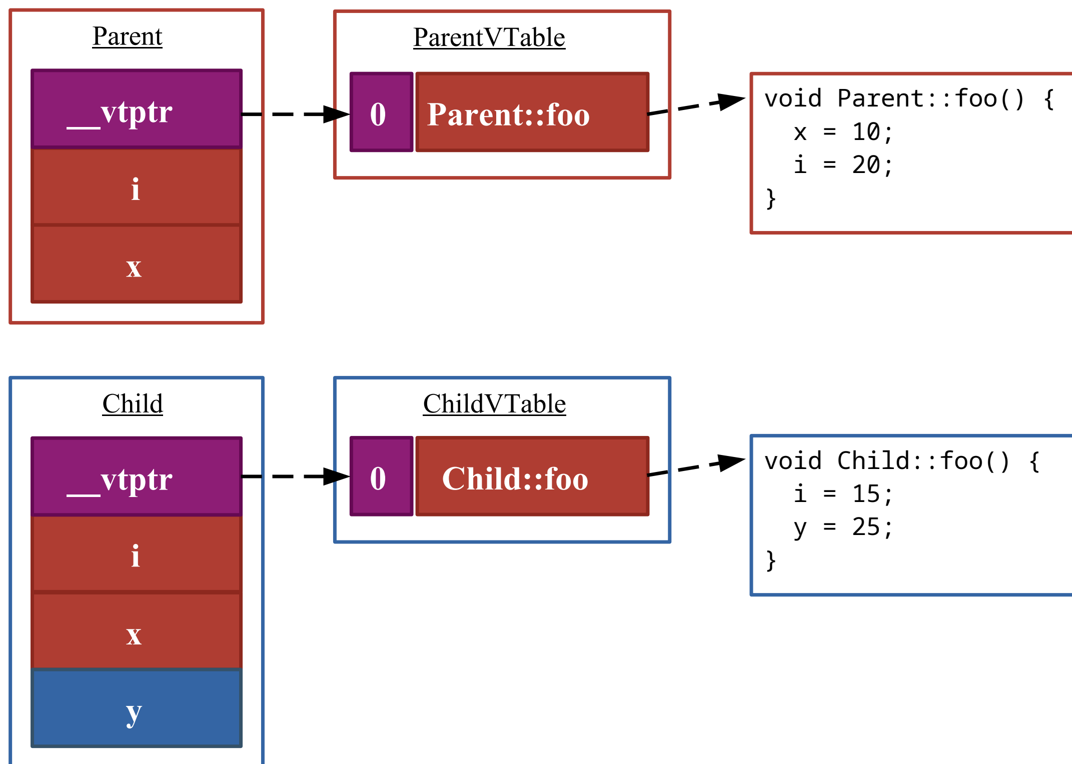



```
void Child::foo() {
  i = 15;
  y = 25;
}
```

Inheritance - Dynamic Dispatch

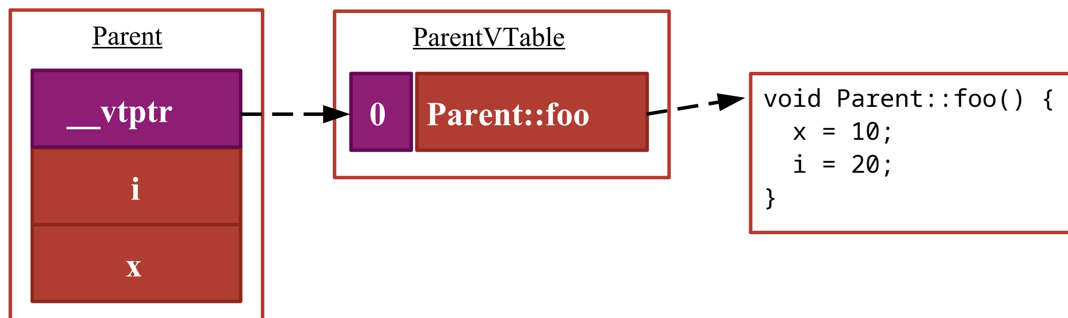


Inheritance - Dynamic Dispatch



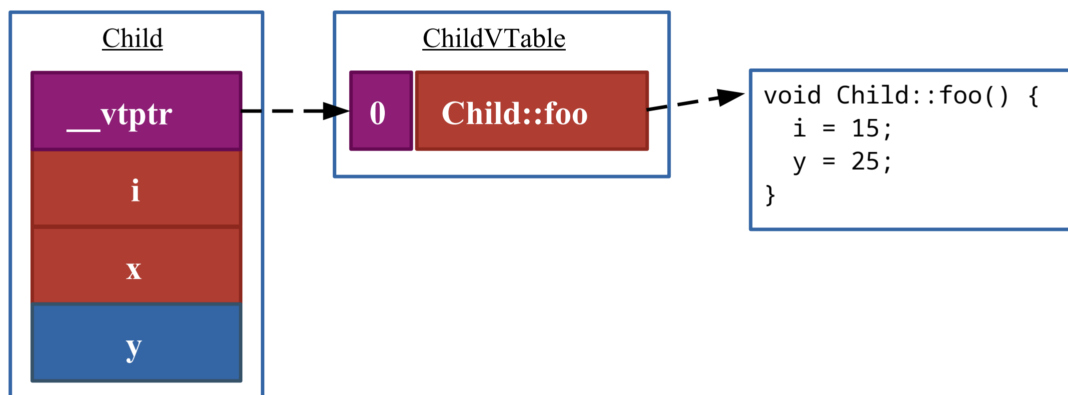
Inheritance - Dynamic Dispatch

```
int main() {
  Parent *ptr;
  Parent p;
  Child c;

  // Assign address of p to ptr
  ptr = &p;

  ptr->foo(); // == ptr->(*__vtptr)[0]()
  // ...
  return 0;
}
```



Inheritance - Dynamic Dispatch

```
int main() {
  Parent *ptr;
  Parent p;
  Child c;

  // Assign address of c to ptr
  ptr = &c;

  ptr->foo(); // == ptr->(*__vtptr)[0]()
  // ...
  return 0;
}
```



Inheritance - Dispatch Types

Function Type	Variable Type		
	Direct `Type var;`	Reference `Type& var;`	Pointer `Type* var;`
non-virtual	static	static	static
virtual	static	dynamic	dynamic
override	static	dynamic	dynamic
final	static	static	static

Inheritance - Summary

Calls of non-virtual functions are faster, however only virtual functions allow polymorphism

Polymorphism is only possible through pointer or reference variables

Destructors of base classes should be declared virtual so derived constructor is called when object is deleted via base pointer

Inheritance can be `public`: **is-a** relationship, `protected`: **??** relationship, and `private`: **has-a** relationship

Inheritance - Summary

Methods that need to be overridden in concrete subclasses are called **pure virtual** as indicated by "`= 0;`" at the end of the method declaration

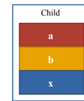
Classes with **at least one pure virtual** function cannot be instantiated and are called **abstract classes**

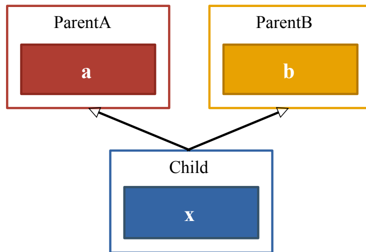
Destructors can be pure virtual

There are **no interfaces** in C++

Interfaces can be **simulated** with abstract classes that contain **only** (pure) **abstract** methods and no data members

Multiple Inheritance





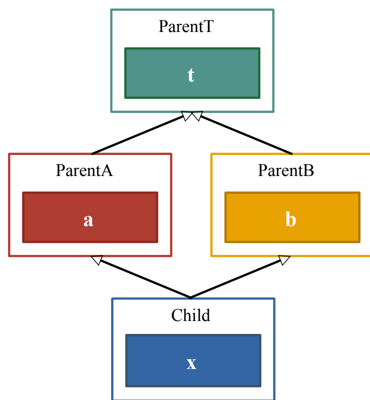

```
class Child
  : public ParentA, public ParentB
{
  // ...
};
```

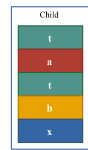
Multiple Inheritance

```
class ParentA {
public:
  int a;

  ParentA(int arg)
    : a(arg * 2)
  {
  }
};

class ParentB {
public:
  int b;

  ParentB(int arg)
    : b(arg * 3)
  {
  }
};
```

```
class Child
  : public ParentA, public ParentB
{
public:
  Child()
    : ParentA(4), ParentB(5), x(11)
  {
  }
  void foo()
  {
    ParentA::a += 5;
  }

private:
  int x;
};
```

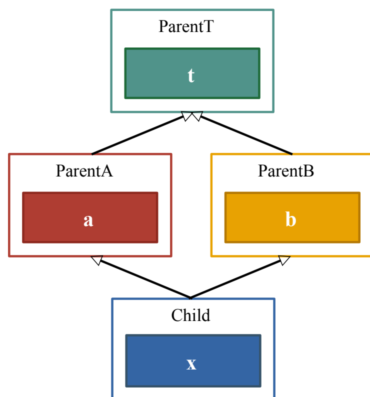
Multiple Inheritance - Dreaded/Deadly Diamond

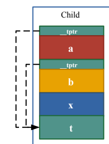





```cpp
class ParentA : public ParentT {};
class ParentB : public ParentT {};

class Child
  : public ParentA, public ParentB
{
  // ...
};
```

Multiple Inheritance - Virtual Inheritance






```cpp
class ParentA : public virtual ParentT {};
class ParentB : public virtual ParentT {};

class Child
  : public ParentA, public ParentB
{
  // ...
};
```

Type Casts

various different casts for different use cases

use casts rarely and only if necessary

some casts may break type safety!

Type Casts - `dynamic_cast`

safe: checks if dynamic types (object types) are compatible

slow: checks at run-time

```
Derived derived;
Base* pBase = &derived;

Derived* pResult = dynamic_cast<Derived*>(pBase);

Base base;
pBase = &base;

pResult = dynamic_cast<Derived*>(pBase); // pResult == nullptr

Base& rResult = dynamic_cast<Derived&>(*pBase); // throws std::bad_cast
```

Type Casts - `static_cast`

rather safe: checks if static types (variable types) are compatible

fast: checks at compile-time

```
Derived derived;
Base* pBase = &derived;

Derived* pResult = static_cast<Derived*>(pBase);

Base base;
pBase = &base;

pResult = static_cast<Derived*>(pBase); // undefined behavior

Base& rResult = static_cast<Derived&>(*pBase); // undefined behavior
```

Type Casts - `const_cast`

manipulates constness of an object

unsafe: **this is very dangerous** as non-const access leads to undefined behavior

primarily used when interfacing with old C libraries

fast: checks at compile-time

```
const char* text = "hello world!";
char* mText = const_cast<char*>(text);

mText[2] = 'f'; // undefined behavior
```

Type Casts - `reinterpret_cast`

very unsafe: hardly does any type checking, most uses lead to undefined behavior

use this only if you really know what you are doing! (you will **not** know from attending this lecture!)

fast: checks at compile-time

```
Base base;
char* ptr = reinterpret_cast<char*>(&base);

ptr[2] = 'f'; // probably undefined behavior
```

Type Casts - C-style cast: `(Typename)variable`

allows pretty much any cast

very unsafe: **this is extremely dangerous** as all other cases above and combinations thereof may happen

never use this cast in C++ code

cannot be searched for with full-text search

```
const char* text = "hello world!";
char* mText = (char*)(text);

mText[2] = 'f'; // probably undefined behavior
```

Automatic Type Deduction

Compiler can often deduce variable type for you:

```
auto a = 5; // a --> int

auto b = a * 5 / 2.0; // b --> double

auto& c = a; // c --> int&

const auto d = 6; // d --> const int

const auto &e = d; // e --> const int&

class Widget {} widget;

auto &wRef = widget; // wRef --> Widget&
```

Saves you from typing long type names

Makes refactoring code easier

This is **not** dynamic typing! The type is fixed at compile time and won't ever change.