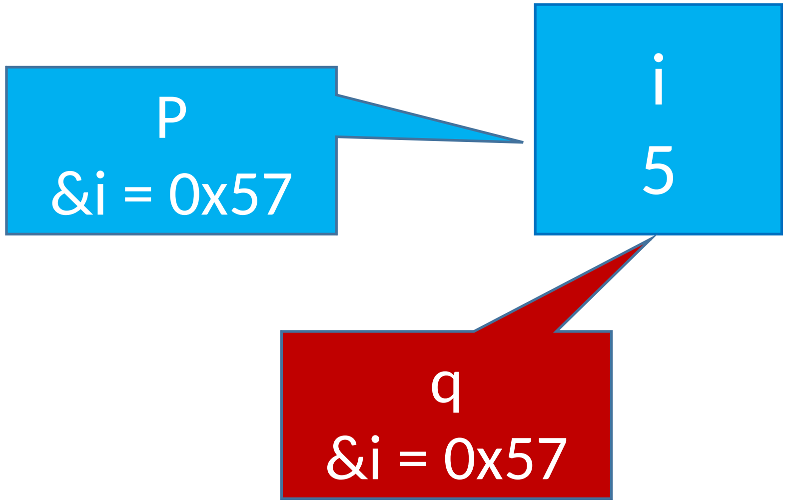
Parameter passing, references

C++ parameter passing made easy

- Everything is passed by value
  - The result of the program is always as if a copy is made of the parameter, and that copy is passed to the routine.
  - The semantics of *references* are less intuitive, but the reference is passed by value, it is just that the interpretation of the reference makes it act differently.
  - This is all you have to remember

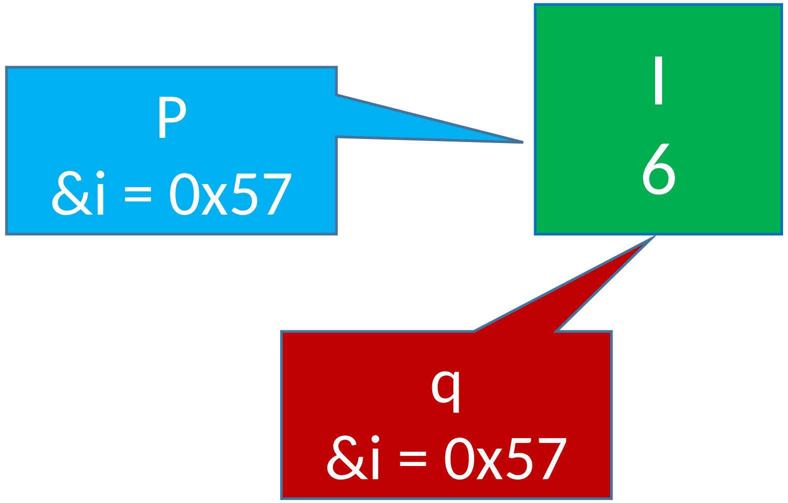
ParamPassPtr

```
class B {
public:
   B(int*);
   virtual ~B();
};
B::B(int* q) {
   *q += 1;
}
B::~B(){ }
int main(int argc, char * argv[]) {
int* p;
int i = 5;
p = &i;
cout << *p << endl;
B* b = new B(p);
cout << *p << endl;
}
```



- When B::B(p) is called, p is passed by value
- *A copy of p is made. The copy is named q*
- B( ) drefs through q, accesses i, and updates it
- But *a copy of p is accessed in B*

```
class B {
public:
   B(int*);
   virtual ~B();
};
B::B(int* q) {
   *q += 1;
}
B::~B(){ }
int main(int argc, char * argv[]) {
int* p;
int i = 5;
p = &i;
cout << *p << endl;
B* b = new B(p);
cout << *p << endl;
}
```



- When B::B(p) is called, p is passed by value
- *A copy of p is made. The copy is named q*
- B( ) drefs through q, accesses i, and updates it
- But *a copy of p is accessed in B*

Primitives are passed by value

```
class B {
public:
   B(int);
   virtual ~B();
};
B::B(int k) {
   k += 1;
}
B::~B(){ }
int main(int argc, char * argv[]) {
   int i = 5;
   cout << i << endl;
   B* b = new B(i);
   cout << i << endl;
}
```

ParamPassPrimitive

i
5

k
5

- When B::B(p) called, i is passed by value
- A copy of i is made, and called k
- B( ) increments the value of k
- i is not changed because a copy of i is accessed by B( )

```
class B {
public:
   B(int);
   virtual ~B();
};
B::B(int k) {
   k += 1;
}
B::~B(){ }
int main(int argc, char * argv[]) {
   int i = 5;
   cout << i << endl;
   B* b = new B(i);
   cout << i << endl;
}
```

i
5

k
6

- When B::B(p) called, i is passed by value
- A copy of i is made, and called k
- B( ) increments the value of k
- i is not changed because a copy of i is accessed by B( )

Objects are passed by value

```cpp
using namespace std;
class B {
public:
   B(int a);
   void print( );
   ~B();
   int age;
};
 B::B(int a) {
   age=a;
}
void B::print( ) {
   std::cout << age << std::endl;
}

B::~B( ) {std::cout << age << " deleted " << std::endl;};
```

```cpp
void foo(B b) {
   b.age = 6;
}
int main(int argc, char * argv[ ])
{
   B b1(5);
   b1.print( );
   foo(b1);
   b1.print( );
   return 0;
}
```

b1
5

b1 copy
(b) 5

- B b1(5) creates an object b1 with age = 5
- b1 is passed by value to foo. A default zero-arg copy
- constructor is invoked, and a copy of b1 is passed to foo.
- foo sets the field of the copy to 6

Objects are passed by value

```
using namespace std;
class B {
public:
   B(int a);
   void print( );
   ~B();
   int age;
};
 B::B(int a) {
   age=a;
}
void B::print( ) {
   std::cout << age << std::endl;
}

B::~B( ) {std::cout << age << " deleted " << std::endl;};
```

```
void foo(B b) {
   b.age = 6;
}
int main(int argc, char * argv[ ])
{
   B b1(5);
   b1.print( );
   foo(b1);
   b1.print( );
   return 0;
}
```

b1
5

b1 copy
(b) 6

- foo sets the field of the copy to 6
- The function foo ends and the copy is destroyed
- In main, b1 is referenced with a value of 5

Objects are passed by value

```
using namespace std;
class B {
public:
   B(int a);
   void print( );
   ~B();
   int age;
};
 B::B(int a) {
   age=a;
}
void B::print( ) {
   std::cout << age << std::endl;
}

B::~B( ) {std::cout << age << " deleted " << std::endl;};
```

```
void foo(B b) {
   b.age = 6;
}
int main(int argc, char * argv[ ])
{
   B b1(5);
   b1.print( );
   foo(b1);
   b1.print( );
   return 0;
}
```

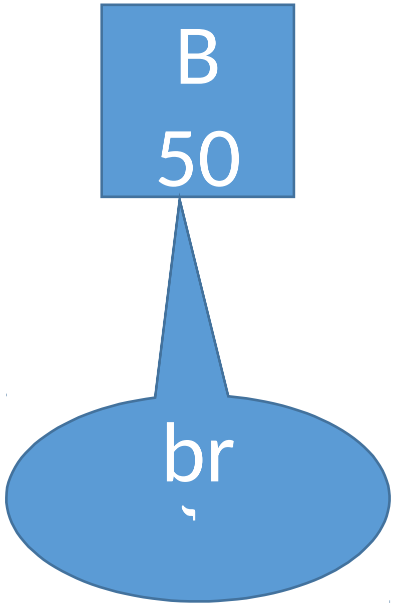
b1
5

```
5
6 deleted
5
5 deleted
```

References!

B b(50);

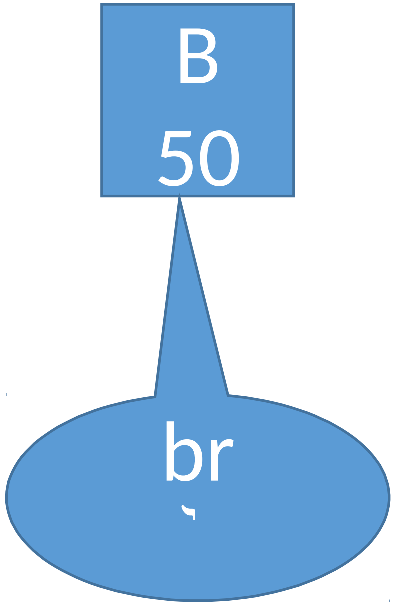
B& br = b



- B b(50) creates an object
- **B& br** creates a reference to a B object
- **B& br = b** makes the reference br refer to the object b
- *Once a reference references an object, it always references that same object. References need to be bound to an object when created.*
- References support polymorphism

B b(50);

B& br = b



Semantically, references are another name for an object -- referring to the reference is the same as referring to the referenced object. *br* is equivalent to *b*

A reference once assigned ***always*** references the same object

A copy of a reference references the same object as the original reference – NOT a copy of the object

References are typically implemented as an object's address, but other implementations are ok as long as the standard is followed.

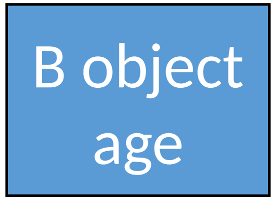
Semantically, reference *br.f* is simply a reference to the field *f*, and actions on *br* are actions on the object itself, and an action on *br.f* is an action on field *f* of *b* itself

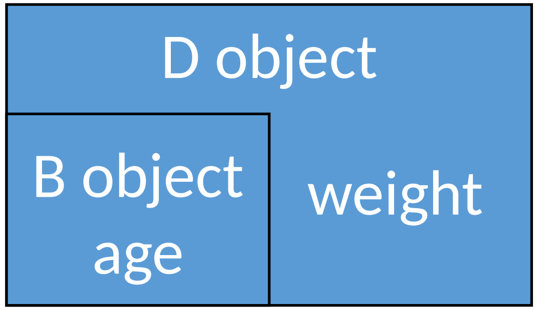
Code in ParamPassRef

```
class B {
public:
  B( );
  B(int a);
  virtual void print( );
  virtual ~B();
  int age;
};

B::B( ) {age=-1;}

B::B(int a) {age=a;}
void B::print( ) {
  cout << "object " << age << endl;
}

class D : public B {
public:
  D(int a, int b);
  virtual void print( );
  virtual ~D();
  int weight;
};
```

```
B::~B( ) {cout << "deleting object " << age << endl; };

D::D(int a, int w) : B(a), weight(w)  { }

void D::print( ) {cout << "object " << age << " " << weight << endl; }

D::~D( ) {cout << "deleting object " << age << " " << weight << endl; }
```

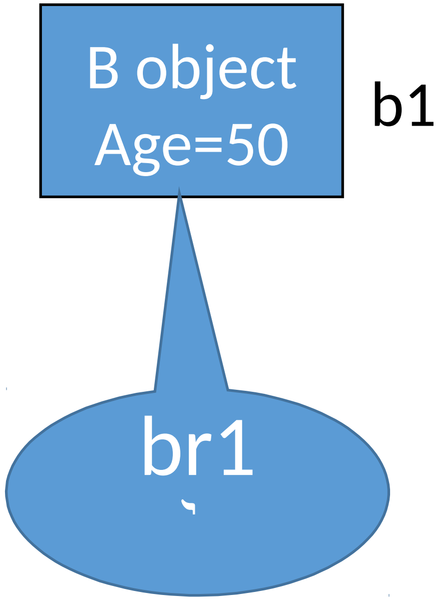
```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
  B& br1 = b1;
  br1.print( );
  br1 = d1;
  br1.print( );
  br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```





```
int main(int argc, char * argv[ ])
{
   B b1(50);
   B b2(150);
   D d1(100, 101);
   D d2(102, 103);
   B& br1 = b1;
   br1.print( );
   br1 = d1;
   br1.print( );
   br1 = b2;
   br1.print( );
   B& br2 = (B&) d2;
   br2.print( );
   br2 = b2;
   br2.print( );
   br2 = d2;
   br2.print( );
   return 0;
}
```

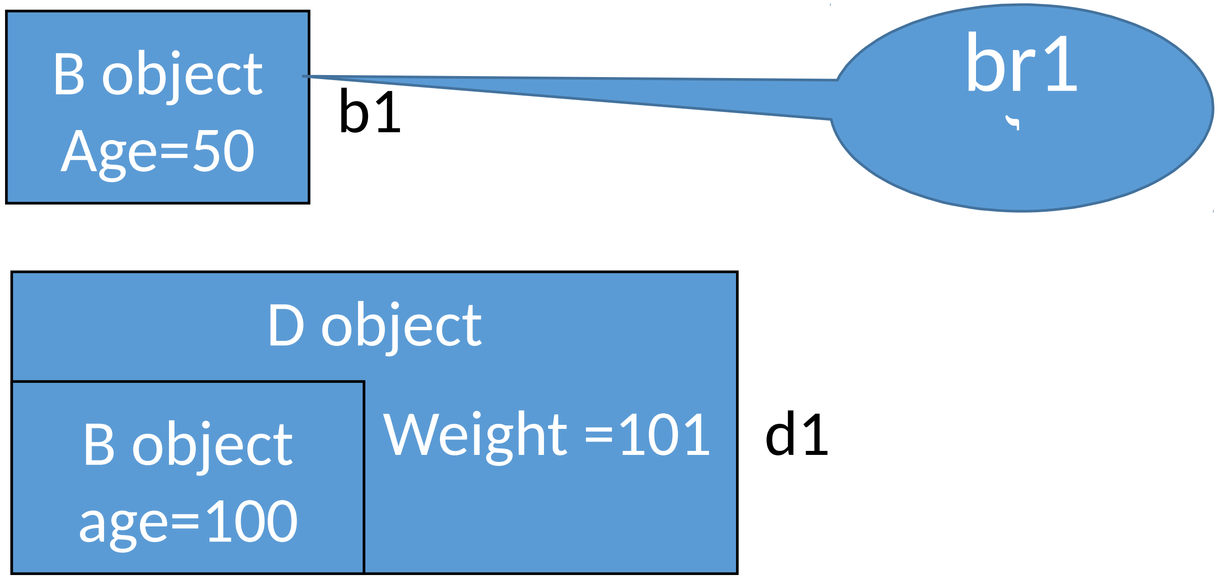
```
int main(int argc, char * argv[ ])
{
   B b1(50);
   B b2(150);
   D d1(100, 101);
   D d2(102, 103);
   B& br1 = b1;
   br1.print( );
   br1 = d1;
   br1.print( );
```




```
void B::print( ) {
  cout << "object " << age << endl;
}
```

object 50

```
int main(int argc, char * argv[ ])
{
   B b1(50);
   B b2(150);
   D d1(100, 101);
   D d2(102, 103);
   B& br1 = b1;
   br1.print( );
   br1 = d1;
   br1.print( );
```



NOT CALLED:

```
void D::print( ) {cout << "object " << age << " " << weight << endl; }
```

CALLED:

```
void B::print( ) {
  cout << "object " << age << endl;
}
```

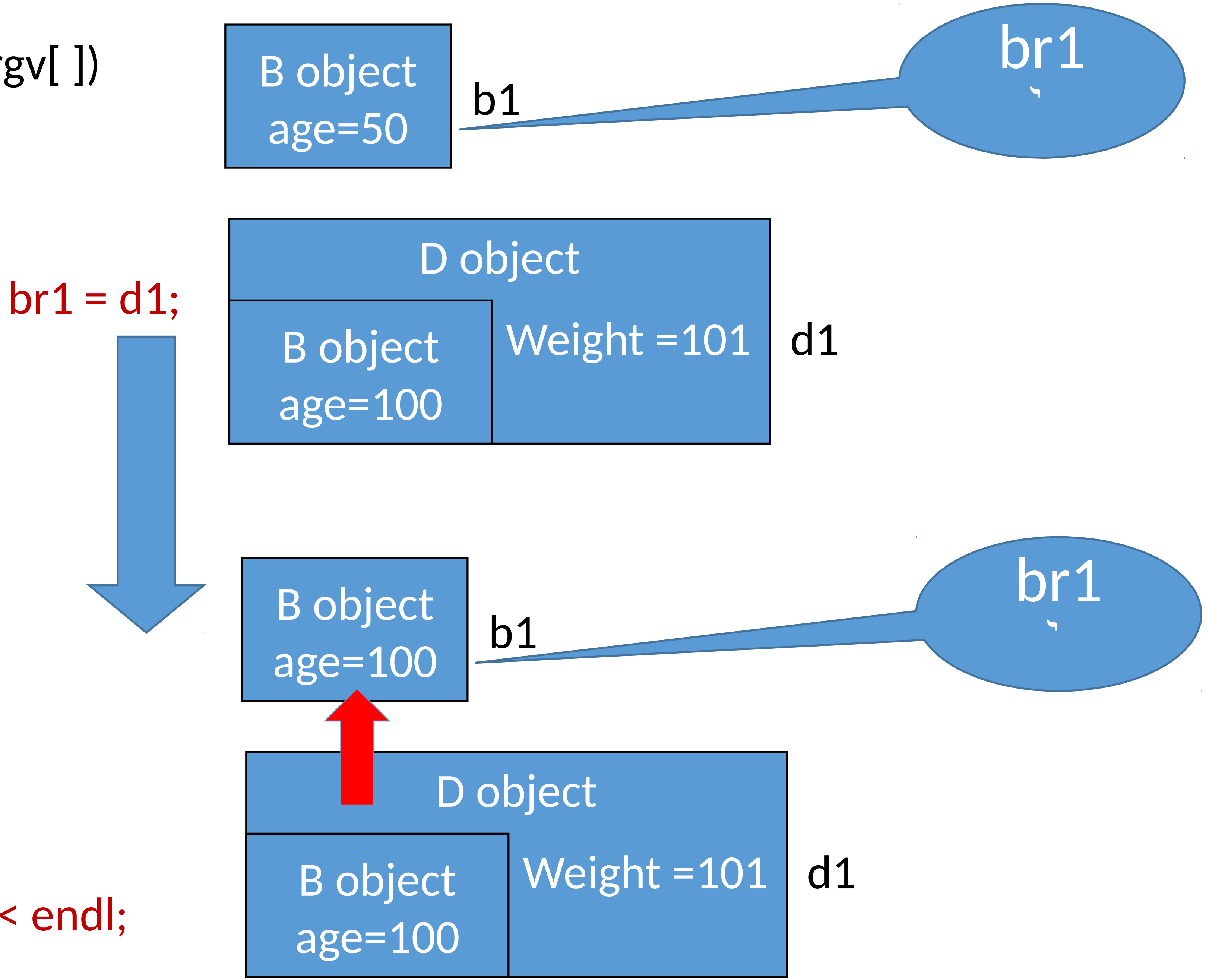
```
object 100
```

Why no polymorphism?

```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
  B& br1 = b1;
  br1.print( );
  br1 = d1;
  br1.print( );
```

CALLED:

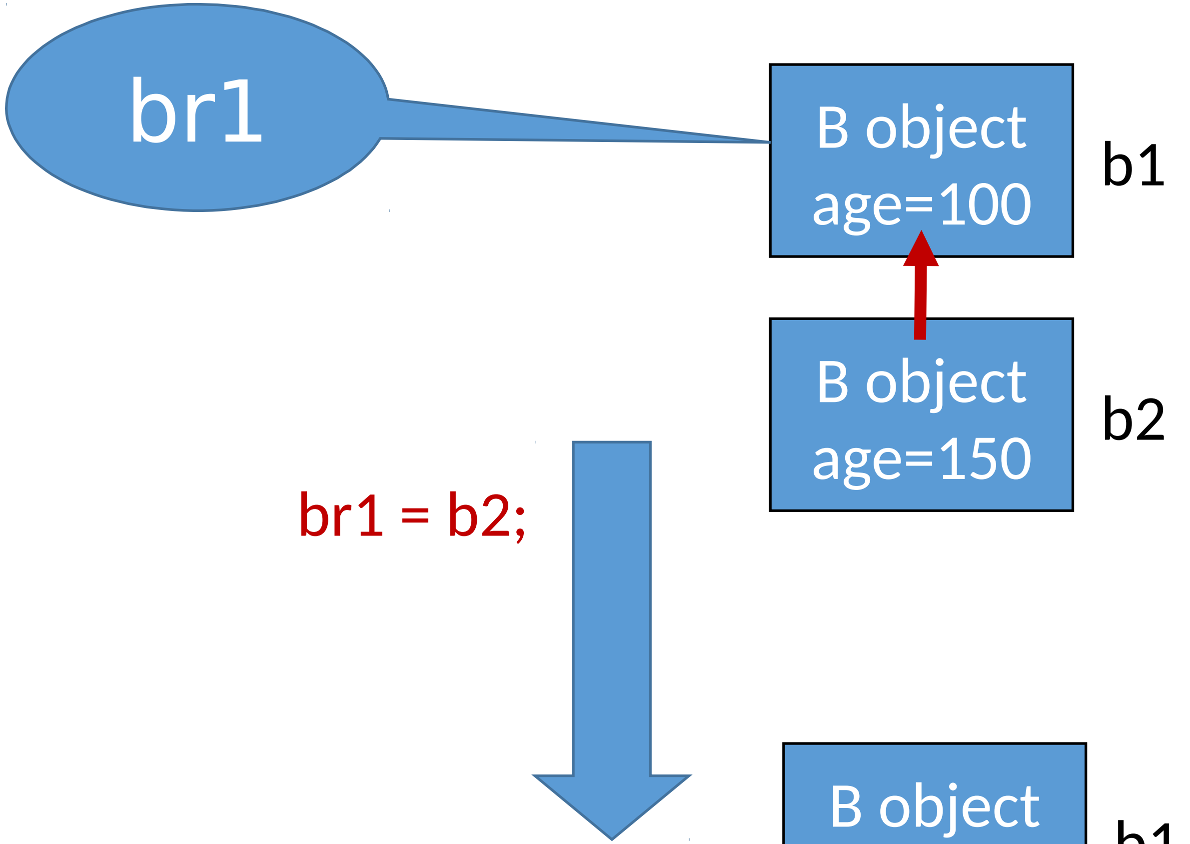
```
void B::print( ) {
  cout << "object " << age << endl;
}
```

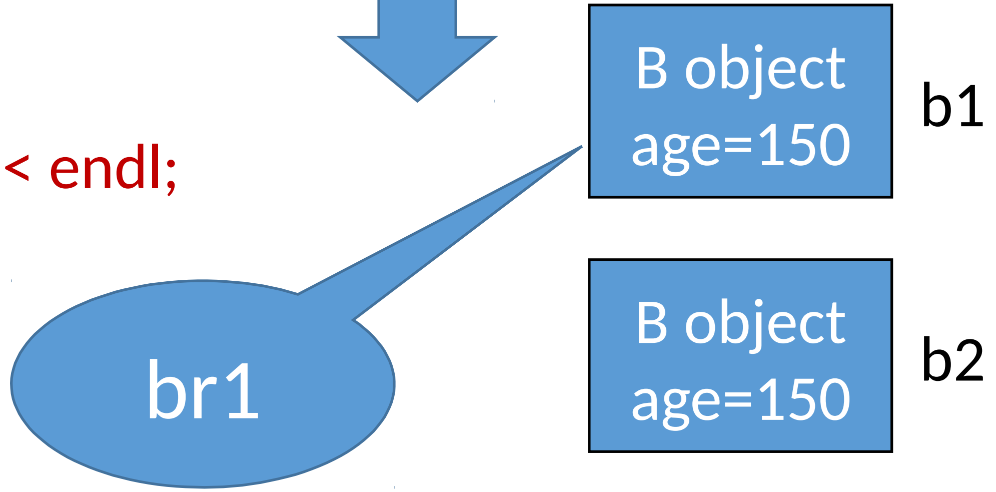


```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
   . . .
  br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```

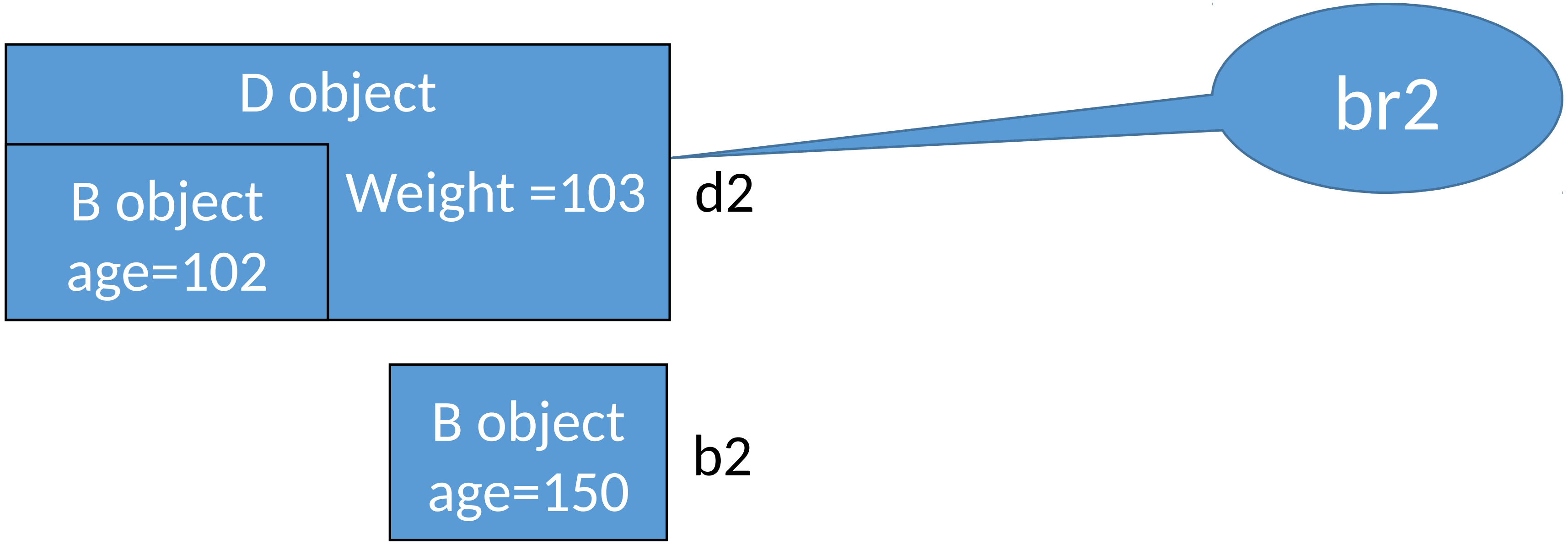
```
void B::print( ) {
  cout << "object " << age << endl;
}
```

```
object 150
```





```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
   . . .
  br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```

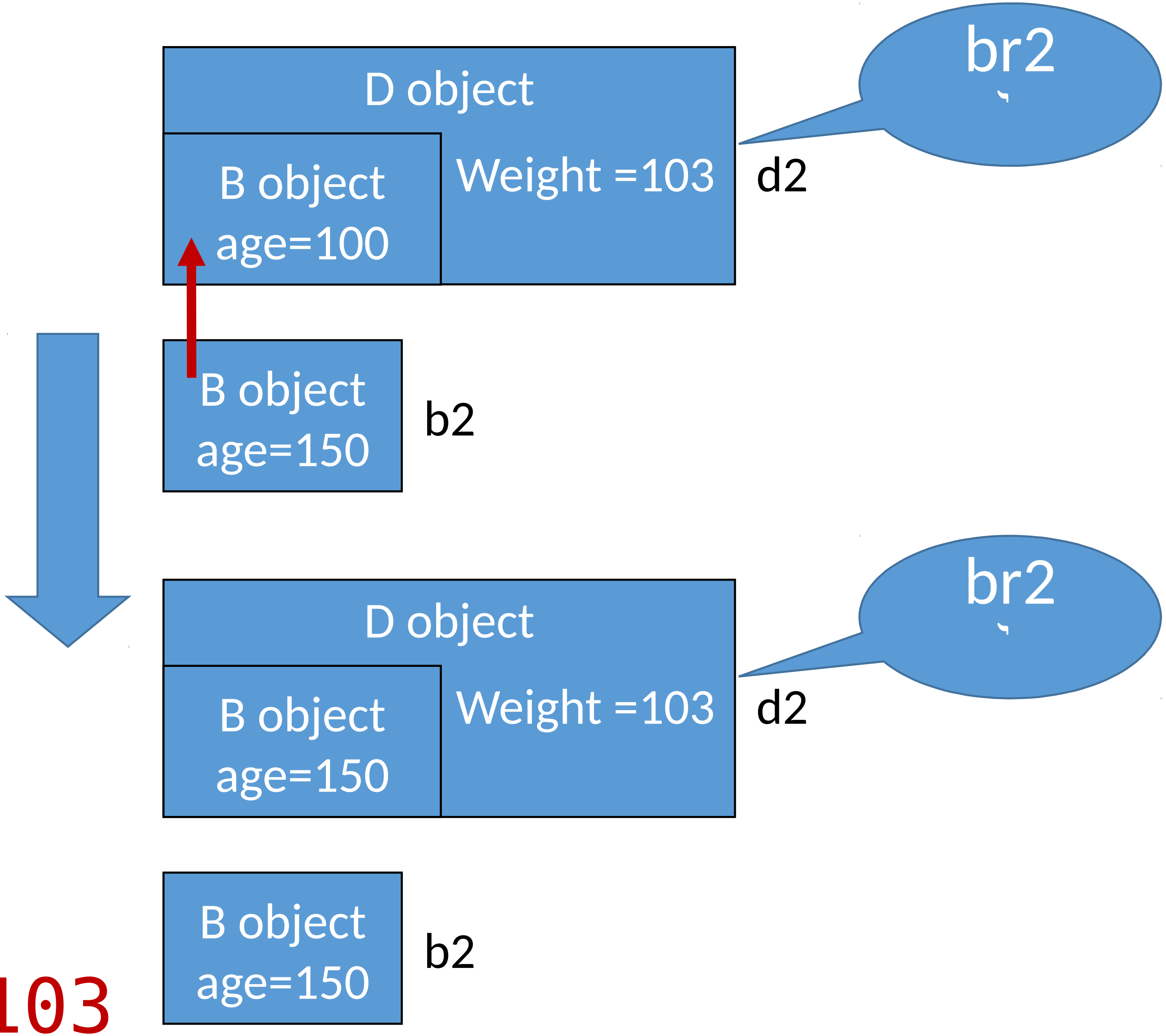



```
object 102 103
object 150 103
```

(note polymorphism)

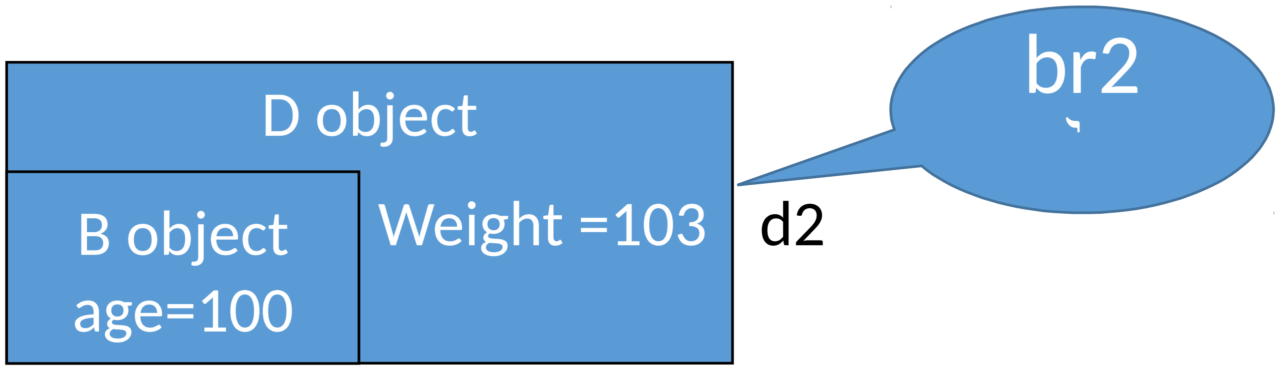
```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
   . . .
  br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```

br2 = d2;



object 150 103

```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
   . . .
  br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```



Br2 = d2, a no-op. We are assigning d2 to itself

object 150 103

References are useful for passing objects into functions

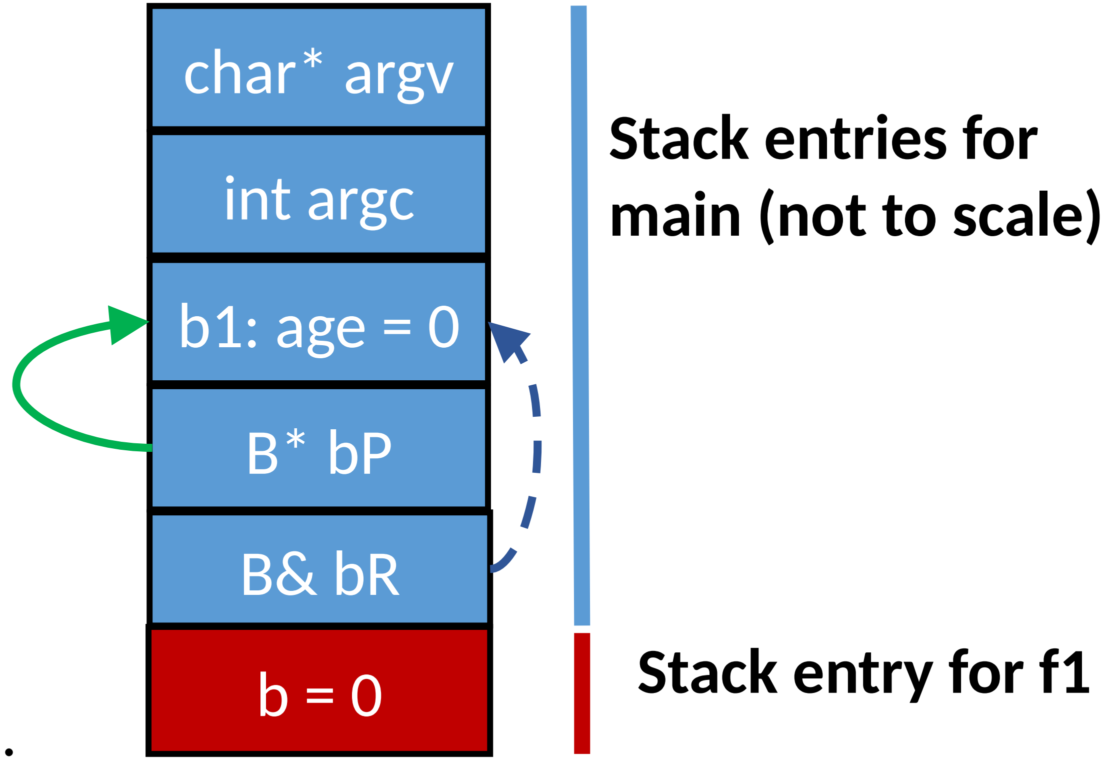
- Objects are passed by value, and therefore a copy of the object is passed to a function
- Passing an object as an argument for a reference parameter causes a copy of the *reference* to be passed.

```
Void do_something(big_object& obj);

big_data d;
do_something(d); // no object copy – a reference to d is
created and passed as the argument
```

- This is more efficient than passing the object (no copy of the object is made)
- The object, through the reference, can be manipulated in the function, not just a copy of the object.

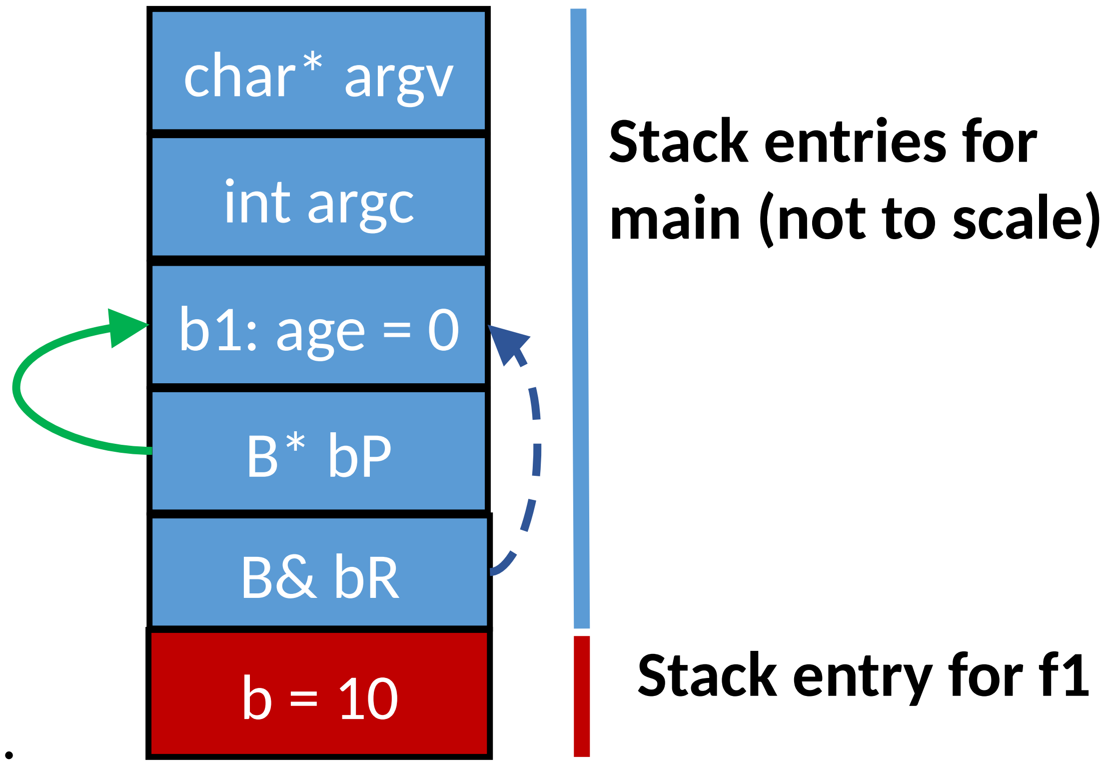
```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```

See ParamPassAll for code. Use main.cpp



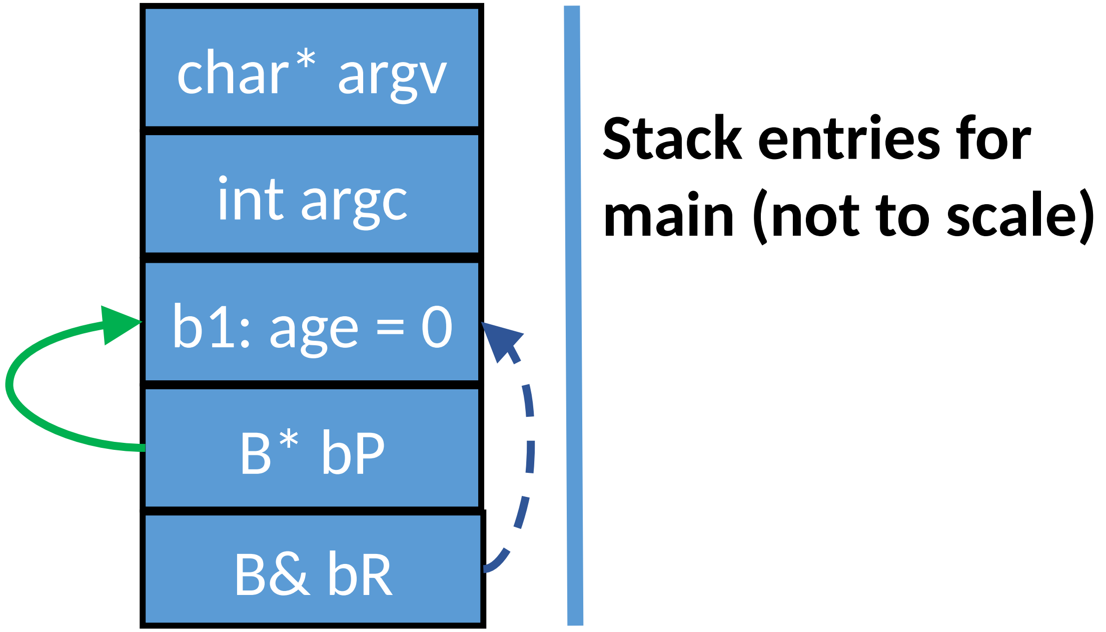
void f1(B b) {b.age = 10;}

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
   B b1;
   B* bP = &b1;
   B& bR = (B&) b1;
   f1(b1);
   std::cout << "after f1 call " << std::endl;
   b1.print(b1, bP, bR);
   fr(bR);
   std::cout << "after fr call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   fp(bP);
   std::cout << "after fp call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   return 0;
}
```




```
void f1(B b) {b.age = 10;}
```

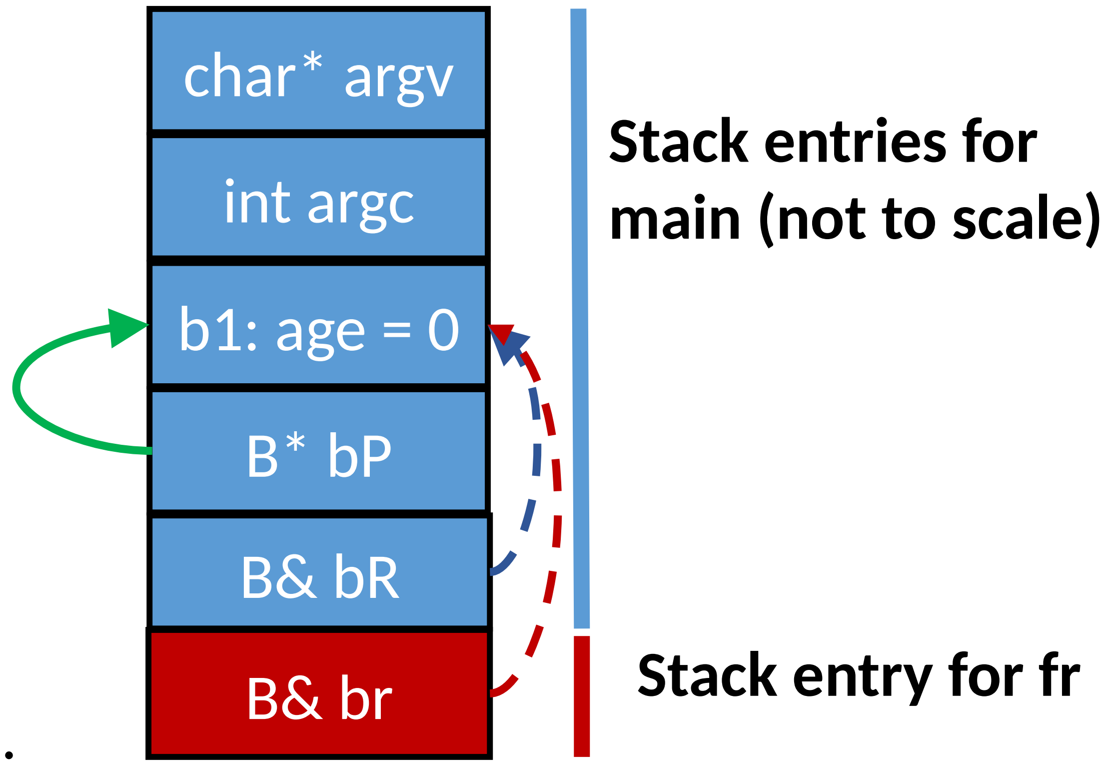
```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```




```
void f1(B b) {b.age = 10;}
```

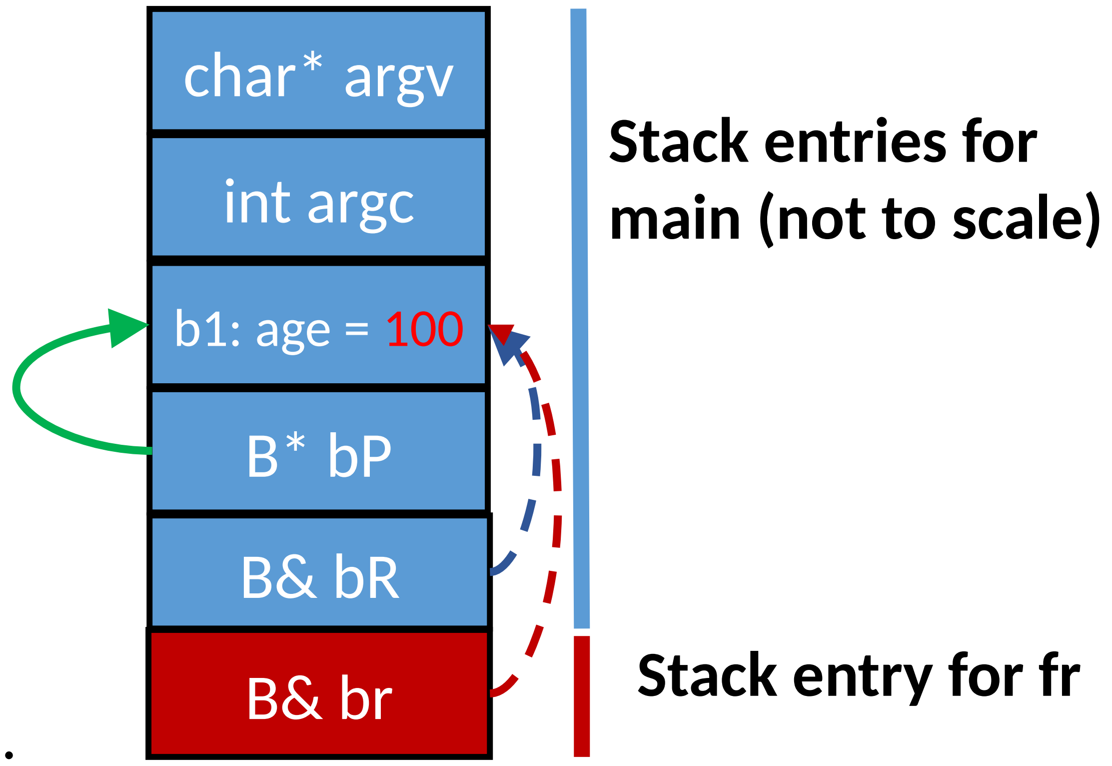
```
after f1 call
b.age = 0, bp.age = 0, br.age = 0
```

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



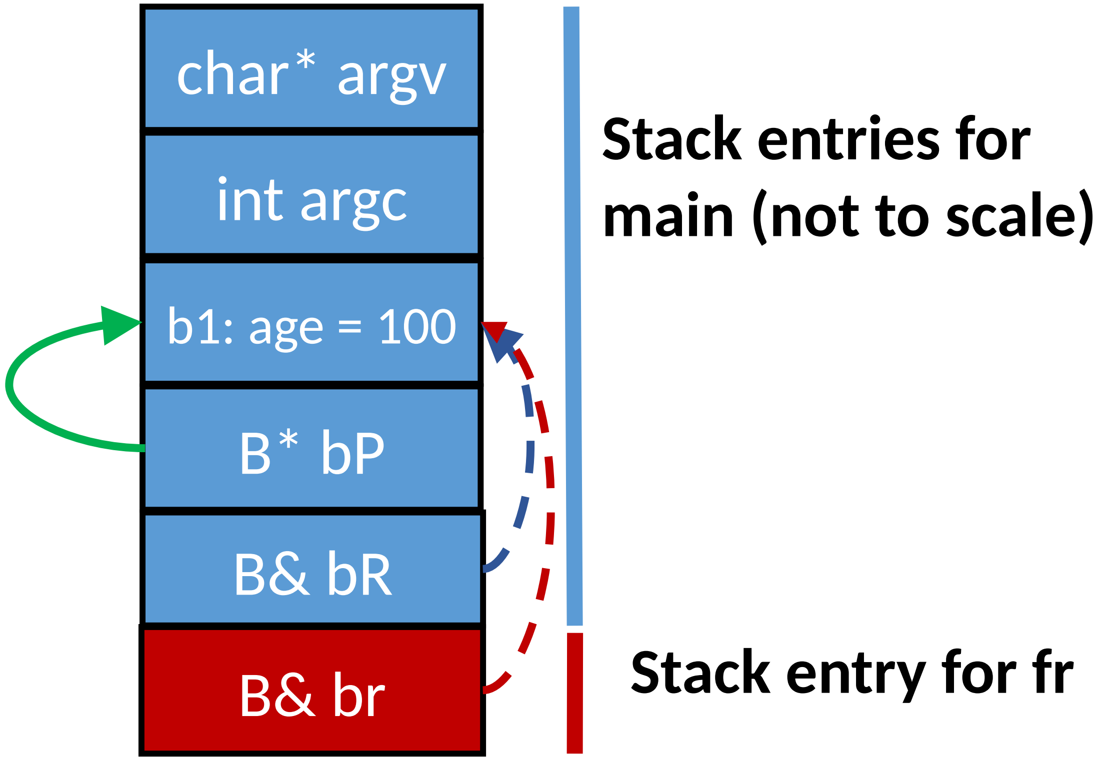

```
void fr(B& br) {br.age = 100;}
```

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



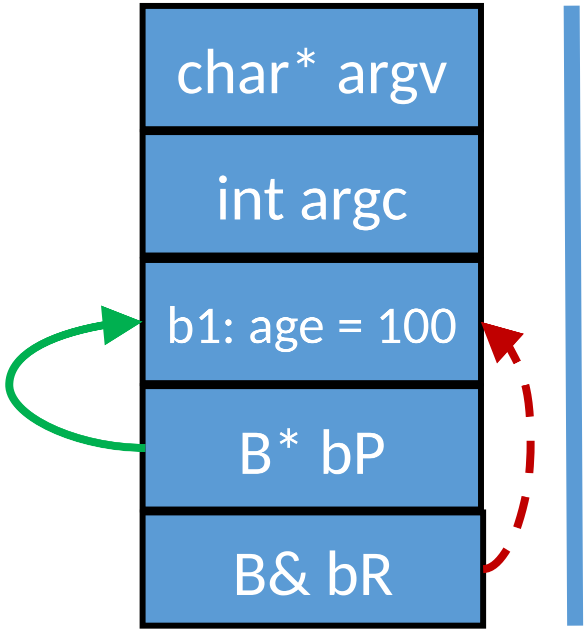

```
void fr(B& br) {br.age = 100;}
```

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
   B b1;
   B* bP = &b1;
   B& bR = (B&) b1;
   f1(b1);
   std::cout << "after f1 call " << std::endl;
   b1.print(b1, bP, bR);
   fr(bR);
   std::cout << "after fr call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   fp(bP);
   std::cout << "after fp call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   return 0;
}
```



A reference is another name for an object, say b1. A copy of the reference is another name for exactly the same object b1. So br, a copy of bR, is simply another name for b1. Put differently, it is the same as b1.

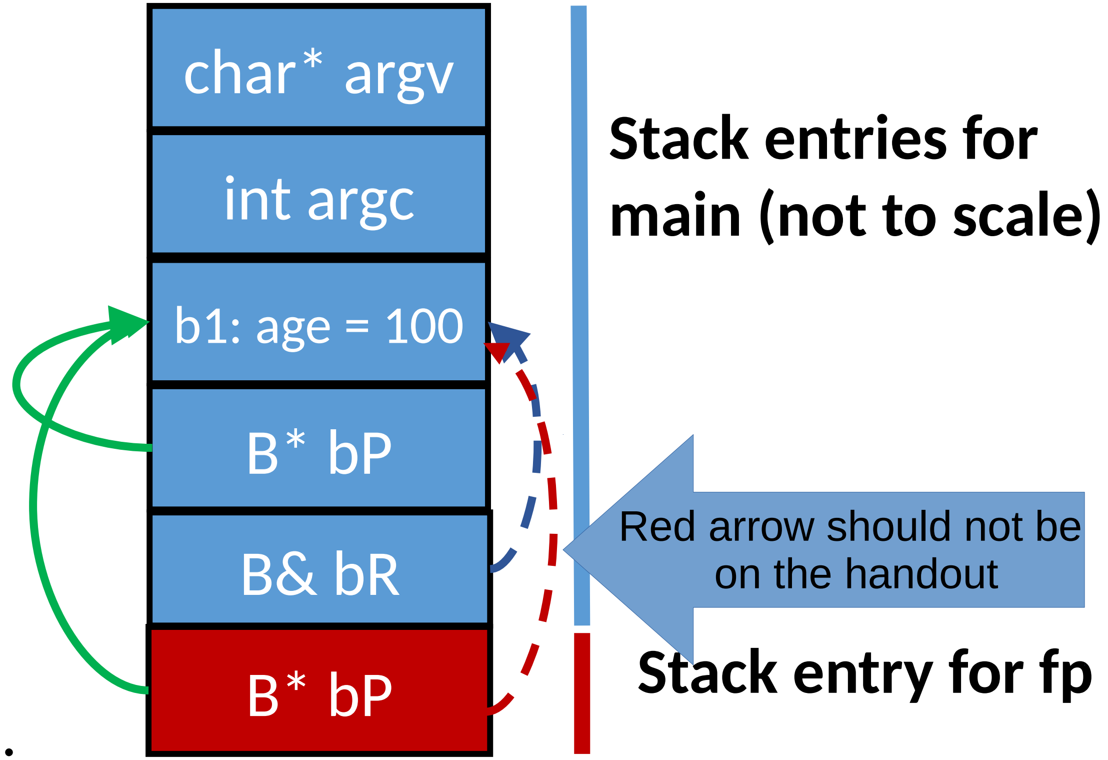
```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
   B b1;
   B* bP = &b1;
   B& bR = (B&) b1;
   f1(b1);
   std::cout << "after f1 call " << std::endl;
   b1.print(b1, bP, bR);
   fr(bR);
   std::cout << "after fr call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   fp(bP);
   std::cout << "after fp call " << std::endl << std::endl;
   b1.print(b1, bP, bR);
   return 0;
}
```



Stack entries for main (not to scale)

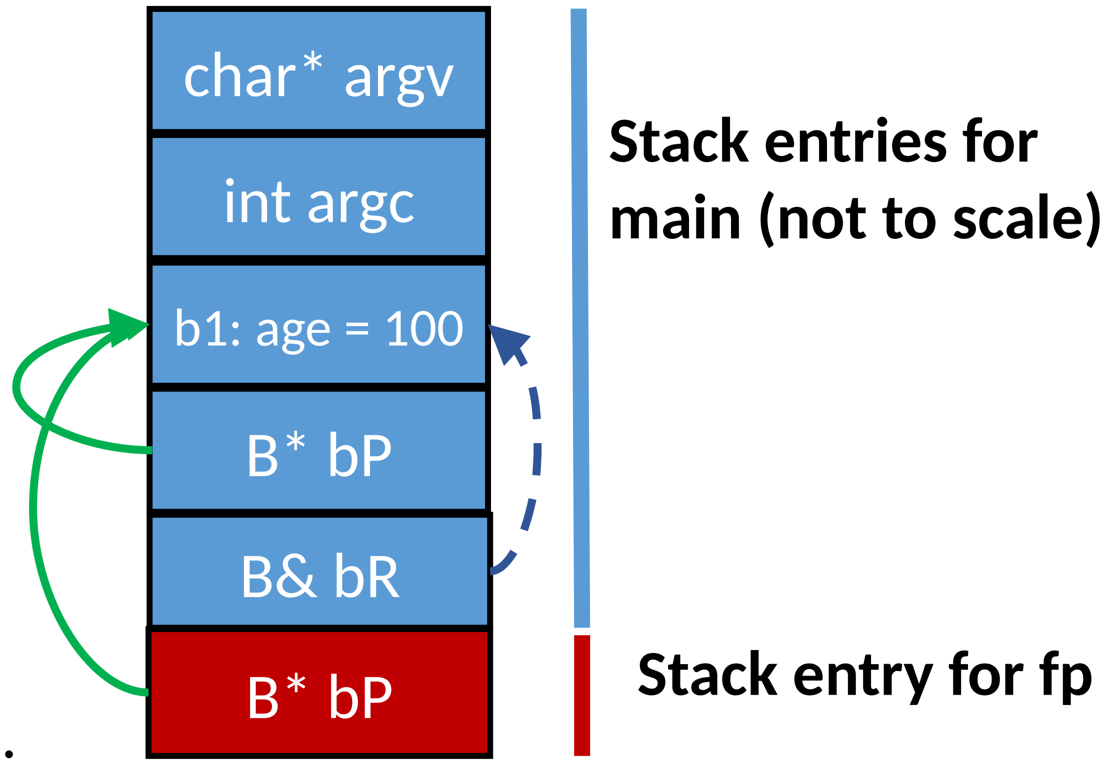
```
after fr call

b.age = 100, bp.age = 100, br.age = 100
```

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



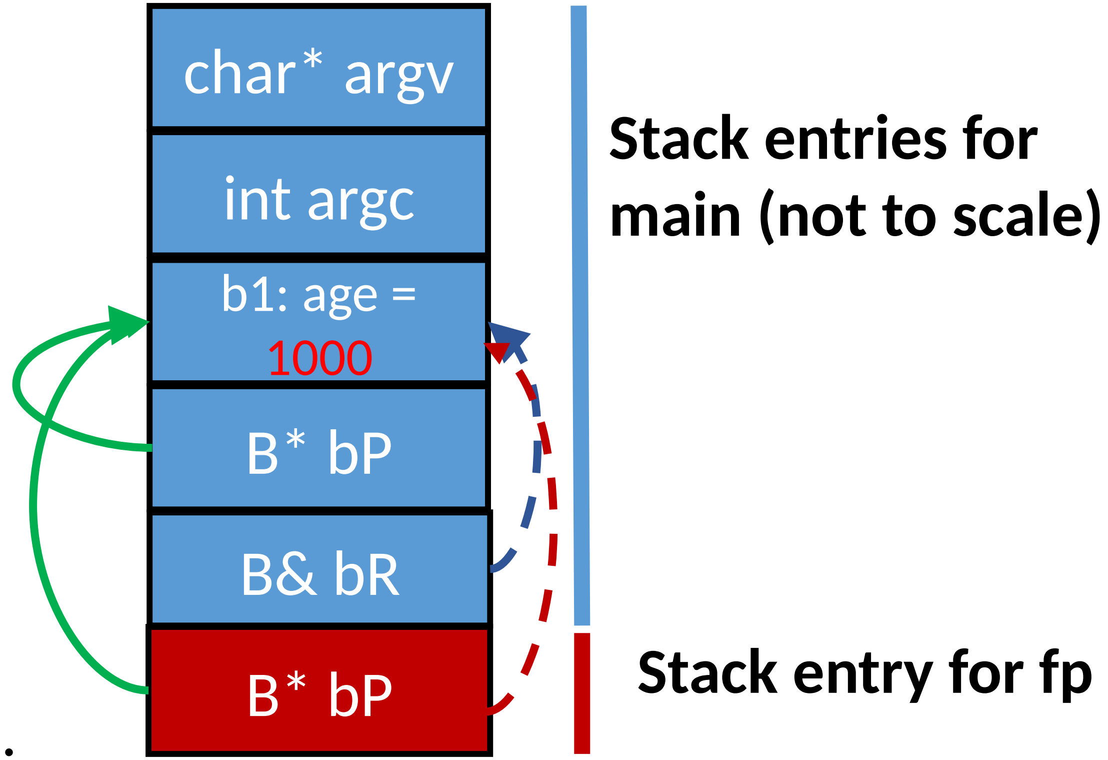
void fp(B* bp) {bp->age = 1000;}

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



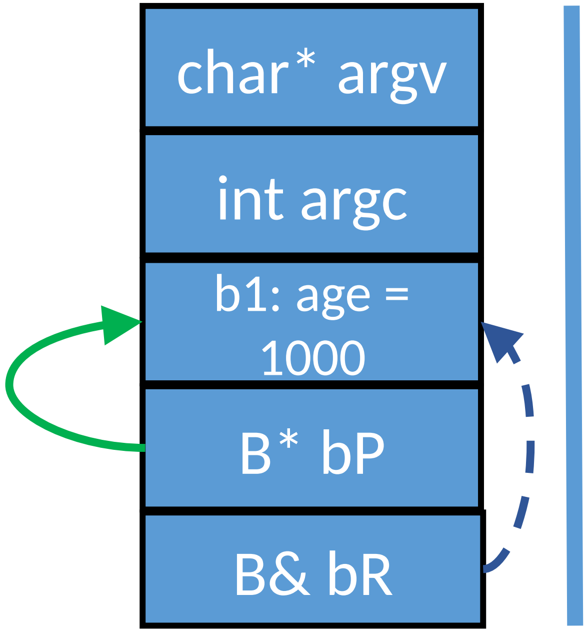

```
void fp(B* bp) {bp->age = 1000;}
```

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



void fp(B* bp) {bp->age = 1000;}

```
void f1(B b) {b.age = 10;}
void fp(B* bp) {bp->age = 1000;}
void fr(B& br) {br.age = 100;}
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  f1(b1);
  std::cout << "after f1 call " << std::endl;
  b1.print(b1, bP, bR);
  fr(bR);
  std::cout << "after fr call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  fp(bP);
  std::cout << "after fp call " << std::endl << std::endl;
  b1.print(b1, bP, bR);
  return 0;
}
```



Stack entries for main (not to scale)

```
after fp call

b.age = 1000, bp.age = 1000, br.age = 1000
```

So why not just use pointers?

See ParamPassAll for code.
Use main2.cpp

```
void f1(B b) {b.age = 10;}
void fr(B& br) {br.age = 100;}

int main(int argc, char * argv[ ]) {
    B b1; // what happens with B b1( );?
    B* bP = &b1;
    B& bR = (B&) b1;
    fr(b1);
    f1(bR);
    fr(bP);
    return 0;
}
```

- An object argument is silently converted to work with a reference parameter (pass a temporary reference)
- A reference parameter is silently converted to work with an object parameter (pass a copy of the object referenced object)
- A pointer is not converted to either – fr(bP) gives an error

```
g++ main2.cpp B.cpp
main2.cpp: In function ‘int main(int, char**)’:
main2.cpp:14:9: error: invalid initialization of reference of type ‘B&’ from expression of type ‘B*’
   fr(bP);
       ^
main2.cpp:4:6: error: in passing argument 1 of ‘void fr(B&)’
 void fr(B& br) {br.age = 100;}
      ^
```

But there is a problem

- When passing objects, we knew it would be passed by value, and not changed.
- If we pass an object as an argument to a reference parameter it might be changed – we have to look at the function code to figure this out
  - This breaks encapsulation: The promise is that the interface will not change, not the code in the function
- *const* parameters prevent this

```
void do_something(const big_data& data);
. . .
big_data d;
do_something(d); // no object copies at all! data aliases d within the function
```

And a solution . . .

void do_something(**const** big_data& data);

. . .

big_data d;

do_something(d); // no object copies at all! *data* aliases *d* within the function

- With const parameters, the compiler guarantees that:
  - The function (do_something) will not assign into **data**.
  - Cannot assign address of **data** into a pointer
  - **data** will not be passed into any non-const function parameter
- const is part of the function prototype, and part of the specification promise.

```
void fr(const B& br) {
  B* b = &br;
  br.age = 100;
}
```

```
int main(int argc, char * argv[ ]) {
  B b1;
  B* bP = &b1;
  B& bR = (B&) b1;
  fr(bR);
  return 0;
}
```

See code in Const

```
g++ main.cpp B.cpp
main.cpp: In function 'void fr(const B&)':
main.cpp:5:12: error: invalid conversion from 'const B*' to 'B*' [-fpermissive]
    B* b = &br;
           ^
main.cpp:6:11: error: assignment of member 'B::age' in read-only object
    br.age = 100;
          ^
```

From Bjourne Stroustrup's "Design and Evolution of C++"

- Why must references always refer to the same object? It is not possible to change what a reference refers to after initialization. That is, once a C++ reference is initialized it cannot be made to refer to a different object later; it cannot be re-bound. I had in the past been bitten by Algol68 references where r1=r2 can either assign through r1 to the object referred to or assign a new reference value to r1 (re-binding r1) depending on the type of r2. I wanted to avoid such problems in C++

Let’s go back to the program at the start of the discussion on references

- This code is found in ParamPassRef

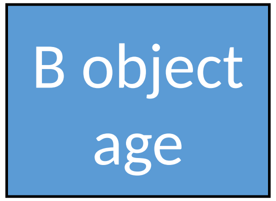
```
class B {
public:
  B( );
  B(int a);
  virtual void print( );
  virtual ~B();
  int age;
};

B::B( ) {age=-1;}

B::B(int a) {age=a;}
void B::print( ) {
  cout << "object " << age << endl;
}

class D : public B {
public:
  D(int a, int b);
  virtual void print( );
  virtual ~D();
  int weight;
};
```

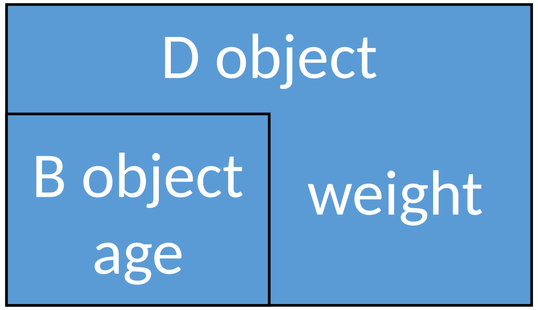
```
B::~B( ) {cout << "deleting object " << age << endl; };
```

Code in ParamPassRef

```
D::D(int a, int w) : B(a), weight(w)  { }

void D::print( ) {cout << "object " << age << " " << weight << endl; }

D::~D( ) {cout << "deleting object " << age << " " << weight << endl; }
```

```
int main(int argc, char * argv[ ])
{
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
  B& br1 = b1;
  br1.print( );
  br1 = d1;
  br1.print( );
br1 = b2;
  br1.print( );
  B& br2 = (B&) d2;
  br2.print( );
  br2 = b2;
  br2.print( );
  br2 = d2;
  br2.print( );
  return 0;
}
```





The destructor calls in this example

The full output from running this program is shown below. Red items have already been discussed.

object 50
object 100
object 150
object 102 103
object 150 103
object 150 103
deleting object 150 103
deleting object 150
deleting object 100 101
deleting object 100
deleting object 150
deleting object 150

- Destructors called as object leave the stack, i.e., the destructor on the last object on the stack is called first
- Destructors always call the base class destructor automatically -- you don't have to do this.

```
int main(int argc, char * argv[ ]) {
  B b1(50);
  B b2(150);
  D d1(100, 101);
  D d2(102, 103);
  . . .
  B& br1 = b1;
  . . .
  B& br2 = (B&) d2;
```

deleting object 150 103 *D part of d2 object*
deleting object 150 B part of d2 object
deleting object 100 101 *D part of d1 object*
deleting object 100 *B part of d1 object*
deleting object 150 *B part of b2 object*
deleting object 150 *B part of b1 object*