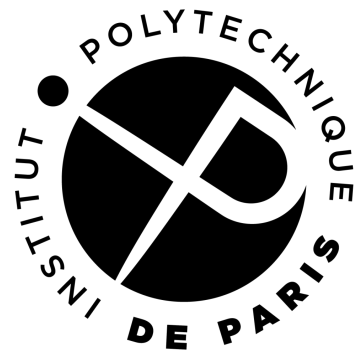


References

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- A reference is an alias to an object
  - Behaves as a pointer that necessarily points to a valid object
  - Declared with `type& var`
  - Assigned at creation, and cannot be change later

The hell of pointers

- Using a pointer is difficult because nothing guarantee that a pointer points to a valid object
  - Can be a null pointer (if you are lucky)
  - Or any random memory location

```
int main(int argc, char* argv[]) {
  struct monster_t* m = (struct monster_t*)0x1000;

  m->print(); // => probably a bug because 0x1000
              //    does not have any reason to
              //    contain a valid monster_t!

  return 0;
}
```

The hell of pointers

- Using a pointer is difficult because nothing guarantees that a pointer points to a valid object
  - Can be a null pointer (if you are lucky)
  - Or any random memory location

```
int main(int argc, char* argv[]) {
  struct monster_t* m; // not initialized!!!

  m->print(); // => probably a bug because m
              //    does not have any reason to
              //    point to a valid monster_t!

  return 0;
}
```

References

- A reference is a pointer that is guaranteed to point to a valid object
  - Cannot be null and can only point to a valid object

- Declared with `type& var`
  - Difference with a pointer: has to be initialized with a valid object + cannot be change after initialization

```
int main(int argc, char* argv[]) {
  int x = 42;
  int& r = x;

  return 0;
}
```

Rewritten by the compiler as

```
int* r = &x
```

References

- A reference is a pointer that is guaranteed to point to a valid object
  - Cannot be null and can only point to a valid object

- Declared with `type& var`
  - Difference with a pointer: has to be initialized with a valid object + cannot be change after initialization

```
int main(int argc, char* argv[]) {
  int x = 42;
  int& r;

  return 0;
}
```

Error, uninitialized reference

References

- Because a reference is necessarily initialized with a valid object, it can only points to a valid object

- Except when we mix pointers and references

```
int main(int argc, char* argv[]) {
  int* x = (int*)0x1000;
  int& r = *x;

  printf("%d\n", r);
  return 0;
}
```

Here the compiler trusts us: the code says that `*x` is a valid object
=> r references an invalid object

References

- Because a reference is necessarily initialized with a valid object, it can only points to a valid object
- Except when we mix pointers and references
- But overall, references avoid many bugs: use them as much as you can!

A reference is assigned once

- A reference is assigned once when initialized and it never changed after

```
int main(int argc, char* argv[]) {
  int x = 42;
  int y = 66;
  int& r = x;
  r = y;
  // => r remains a reference to x
  //    “r = y” stores 66 in x
}
```

A reference is an alias for another object

(=> the compiler does not necessarily allocate memory for the reference, it tries to only use it during compilation)

References and functions

- A function can have a parameter with a reference type
  - In this case, in the caller, we don't explicitly take the address
  - We say that the argument is passed by reference

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
}

int main(int argc, char* argv[]) {
  monster_t m { "Pikachu", 42 };
  f(m);
  return 0;
}
```

```
… f(...* m)
```

```
m->name
```

```
f(&m)
```

Rewritten by the compiler as

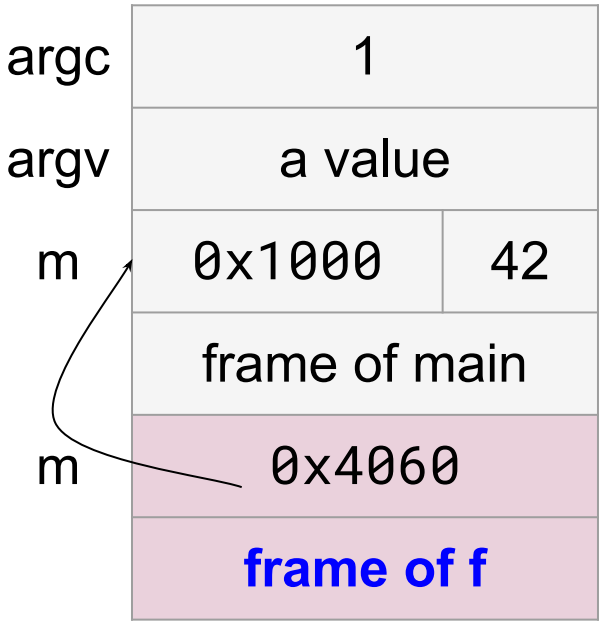
Note: parameter initialized once with the frame is allocated

References and functions

- A function can have a parameter with a reference type
  - In this case, in the caller, we don't explicitly take the address
  - We say that the argument is passed by reference

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
}

int main(int argc, char* argv[]) {
  monster_t m { "Pikachu", 42 };
  f(m);
  return 0;
}
```

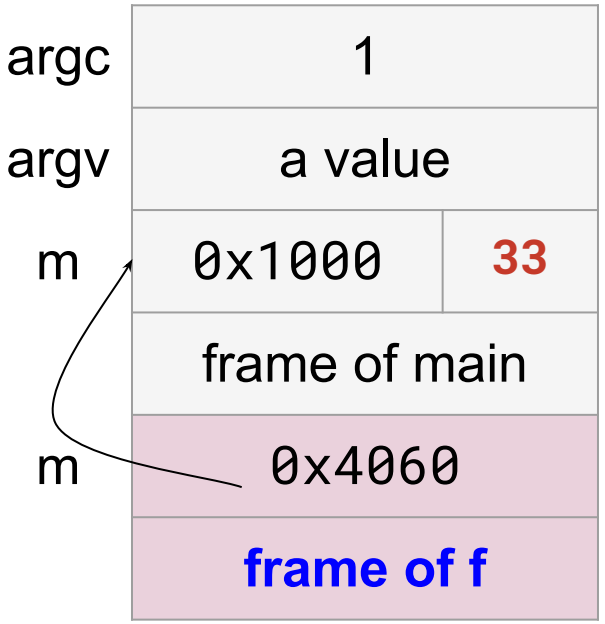


References and functions

- Consequence: the callee modifies the data structure in the caller

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
  m.health = 33;
}

int main(int argc, char* argv[]) {
  monster_t m { "Pikachu", 42 };
  f(m);
  return 0;
}
```

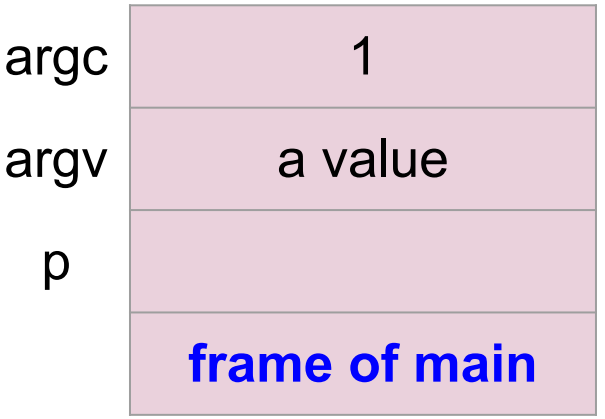


References, pointers and functions

- If p is a pointer to an object allocated with `new`
  - Since *p is a valid object, it can be passed as a parameter

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
  m.health = 33;
}

int main(int argc, char* argv[]) {
  monster_t* p =
   new monster_t { "Pikachu", 42 };
  f(*p);
  return 0;
}
```

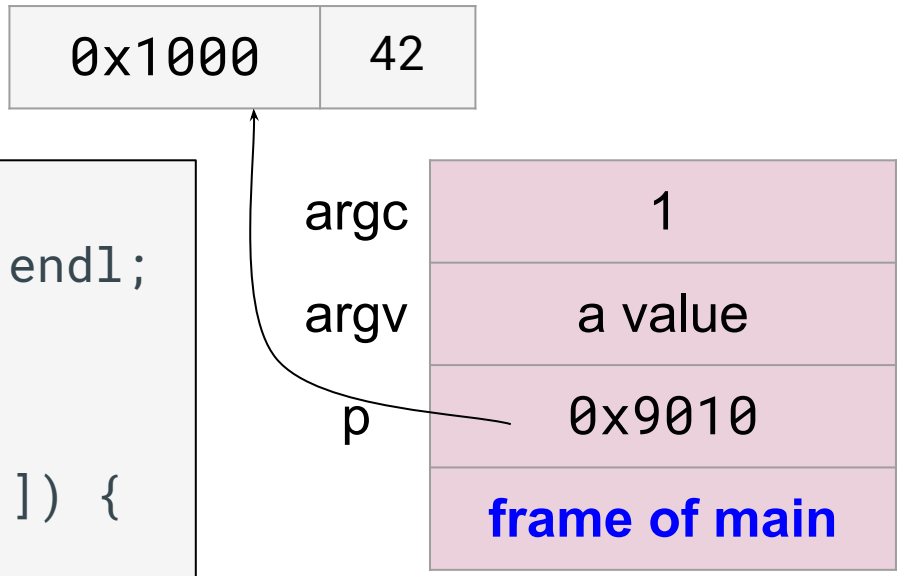


References, pointers and functions

- If p is a pointer to an object allocated with `new`
  - Since *p is a valid object, it can be passed as a parameter

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
  m.health = 33;
}

int main(int argc, char* argv[]) {
  monster_t* p =
   new monster_t { "Pikachu", 42 };
  f(*p);
  return 0;
}
```

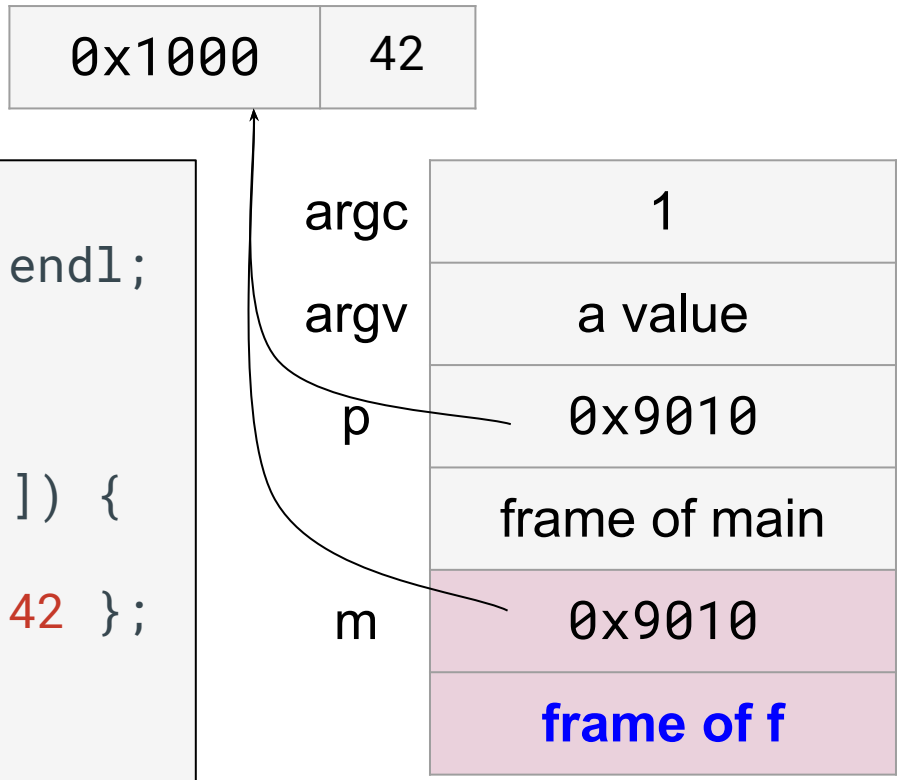


References, pointers and functions

- If p is a pointer to an object allocated with `new`
  - Since *p is a valid object, it can be passed as a parameter

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
  m.health = 33;
}

int main(int argc, char* argv[]) {
  monster_t* p =
   new monster_t { "Pikachu", 42 };
  f(*p);
  return 0;
}
```

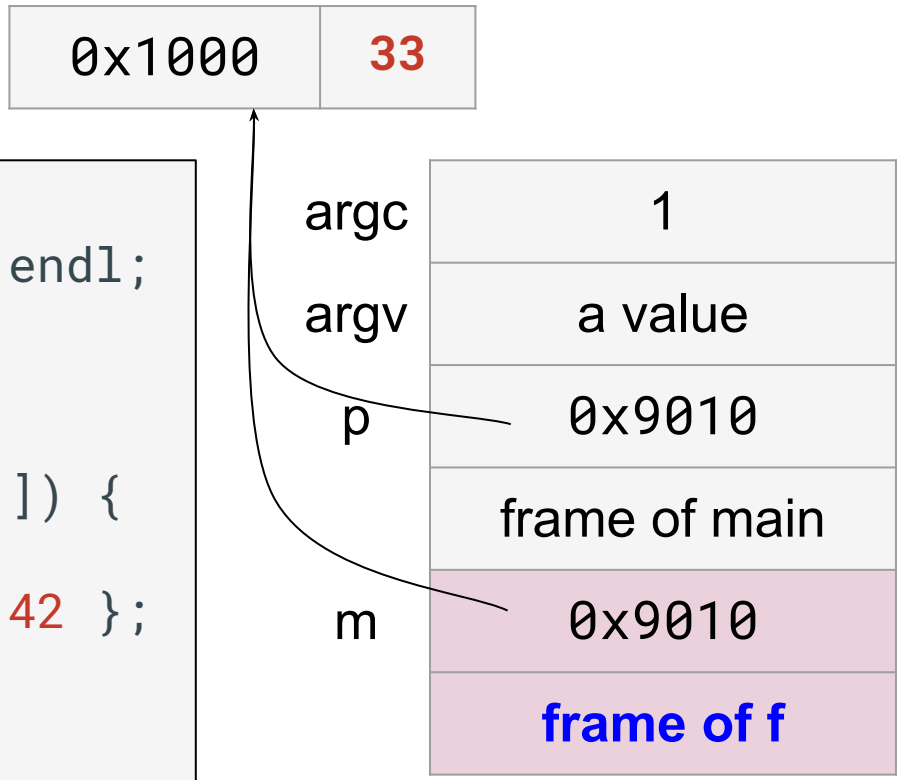


References, pointers and functions

- If p is a pointer to an object allocated with `new`
  - Since *p is a valid object, it can be passed as a parameter

```
void f(struct monster_t& m) {
  std::cout << m.name << std::endl;
  m.health = 33;
}

int main(int argc, char* argv[]) {
  monster_t* p =
   new monster_t { "Pikachu", 42 };
  f(*p);
  return 0;
}
```



Arrays and references

- You cannot create an array of references
  - The compiler cannot easily check that the elements point to valid objects

- But you can use a reference to an array
  - An array declaration already declares a reference

```
void f(int (&tab)[3]) {
}

int main(int argc, char* argv[]) {
  int x[] = { 1, 2, 3 };
  f(x);
  return 0;
}
```

`tab` is guaranteed to reference a valid array of 3 elements

`x` is a reference to an array (guaranteed to reference a valid object)

Fields and references

- The field of a class can be a reference
  - Initialized in the constructor, never null

```
struct holder_t {
  int& val;

  holder_t(int& val) : val { val } { }
};

int main(int argc, char* argv[]) {
  int x = 42;
  holder_t h { x };
  h.val = 666;
  std::cout << x << std::endl; // 666
  return 0;
}
```

Fields and references

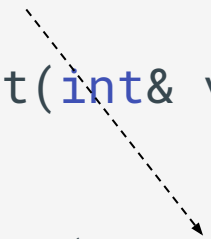
- But using a reference field can be dangerous

```
struct holder_t {
  int& val;

  holder_t(int& val) : val { val } { }
};

holder_t* f(int x) {
  return new holder_t { x };
}

int main(int argc, char* argv[]) {
  holder_t* q = f(33);
  // bug: q->val references
  //      an invalid memory location
  return 0;
}
```



`q->val` references unallocated memory inside the frame of `f`
=> `q->val` has a random value

Bad design because the bug is hidden to the user of `holder_t` in f

Key concepts

- A reference is an alias to an object
  - Behaves as a pointer that necessarily points to a valid object
  - Declared with `type& var`
  - Assigned at creation, and cannot be change later