Administrivia

Portfolio : Submission

- Blackboard submission is now open
  - Run `` `./prepare_submissions.sh` ``
  - Submit the `.tar.gz` file that it produces
- Submissions are “unlimited”
  - Can upload as many versions as you want
  - The last uploaded version is the one that is assessed
- Sanity check : Monday 11th at 9:00
  - Does it look roughly ok; are you submitting the right file?
  - Will not result in a mark, but might indicate problems
  - *Completely optional*
- Final submission: Friday 15th at 18:00

Portfolio : Assessment

- The portfolio is assessed *functionally*
  - Have you performed/implemented the desired function?
  - There are no marks for style/elegance/performance
  - It is impossible for us to say *how* you did it
- Each exercise is equally weighted
  - That includes “delete this directory” and “dft”
  - Make rational decisions on marks *vs* learning *vs* time
- These exercises are ~~easy~~ straightforward
  - We expect people to get 90-100%
  - We expect people to get ~70% on the mid-term
  - We expect people to get ~65% on the course overall
  - *Which means for the final assessment?*

Mid-term test : coverage + format

- Subject coverage is `[command-line,recursion)`
  - Roughly equal coverage of each week
- Different questions for Thursday and Friday tests
- Format is multiple-choice
  - Team-based-learning questions are a good model
    - Though with less ambiguous answers
- Computing lab time follows after the tests
  - No new lab exercises
  - GTAs/UTAs still available
  - Good chance to catch-up or ask quesitons

Thursday : no new content!

- We are taking a pause and consolidating
  - *What can you do with what you've learnt?*
  - *How does it link to other modules?*
- No new ideas/concepts will be introduced
  - It will not (directly) help you in the mid-term

- ***Instead***: practical applications and demos of code
  - Show how to use this knowledge in practise
  - Demonstrate code through real examples
  - Help explain why we're doing what we're doing
  - Maybe (?) make it clearer how things hang together

Scopes

Scopes

- Scopes manage the naming of things
  - What does "`x`" mean at this point in the source code?
  - In what region of code does "`x`" refer to the same thing?
- *Global* names can only enter scope[1]
  - Functions and types declare *global* names
  - *"From this point onwards, f is a function with this type"*
- *Local* names enter ***and*** leave scope
  - Variables and parameters declare *local* names
  - *"Within this for-loop* `i` *is an integer variable"*
  - *"Within this function* `x` *is an integer parameter"*

[1] – We will not talk here of global variables; you don't need them, and they are a bad habit

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?
  - Curly brackets limit scopes

Strictly-speaking you can use acc in its own initializer, but it is a weird corner-case. Better to think of the variable as not in scope yet.

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?
  - Curly brackets limit scopes

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?
  - Curly brackets limit scopes
  - Scopes can be nested

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?
  - Curly brackets limit scopes
  - Scopes can be nested

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}
```

```
float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}
```

```
float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n)
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
}

float power(float x, int n)
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n); /*
{
    int acc = 1;
    for(int i=1; i<=n; i=i+1){
        acc = acc * i;
    }
    return acc;
} */

float power(float x, int n); /*
{
    float acc = 1;
    for(int i=1; i<=n ; i=i+1){
        acc = acc * x;
    }
    return acc;
}*/

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n);

float power(float x, int n);

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n);
```

```
float power(float x, int n);
```

```
float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

```
int factorial(int n);
```

```
#include <cmath>
```

```
float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + power(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

- `#include` brings things from header file into scope

```
int factorial(int n);
```

```
#include <cmath>
```

```
float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + pow(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types

- How do we know which is which?
- Scopes: which *acc* is my *acc*?

- Applies to functions too

- `#include` brings things from header file into scope

```
int factorial(int n);
```

```
// Contents of /usr/include/c++/8/cmath
// ...
float pow(float base, float exp);
float exp(float arg);
float log(float arg);
float sin(float arg);
float cos(float arg);
// ...

float mexp(float x, int d)
{
    float acc = 0;
    for(int i=0; i<d ; i=i+1){
        acc = acc + pow(x, i) / factorial(i);
    }
    return acc;
}

int main()
{
    float x;
    cin >> x;
    float y = mexp(x);
    cout << y;
}
```

- We have multiple ‘i’s
- …and multiple ‘x’s
- …and multiple ‘acc’s
  - with different types
- How do we know which is which?
- Scopes: which *acc* is my *acc*?
- Applies to functions too
- `#include` brings things from header file into scope

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

Active names:

```
int f(int) : function
```

```
int f(int);
```

```
int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Active names:

```
int f(int) : function
int main() : function
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Active names:

```
int f(int) : function
int main() : function
int main   : variable
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
int main   : variable
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
? cout     : ?
? cin      : ?
? to_string: ?
...
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” global

Active names:

```
int f(int) : function
int main() : function
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}
```

```
int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
int x      : parameter
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
int x      : parameter
int main   : variable
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

~~int f(int) : function~~
~~int main() : function~~
int x : parameter
int main : variable
int f : variable

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals

Active names:

```
int f(int) : function
int main() : function
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes : naming at compile-time

- The compiler tracks all names
  - Where is it defined?
  - What type does it have?
- Names can change
  - Depends where you are in code
  - Locals can “shadow” globals
- This code is not very good
  - f is not a good global name
  - Lots of shadowing makes code more difficult to understand

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Scopes vs Lifetimes

- Scope : in what region of code is a name active?
- Lifetime : how long does a value/storage live?

- Scopes and lifetimes are strongly linked
  - Variables live as long as they are in scope
  - But shadowing means live variables may not be in scope
- Lifetimes are more complex with dynamic allocation
  - We'll see this in a little while

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at *run-time*

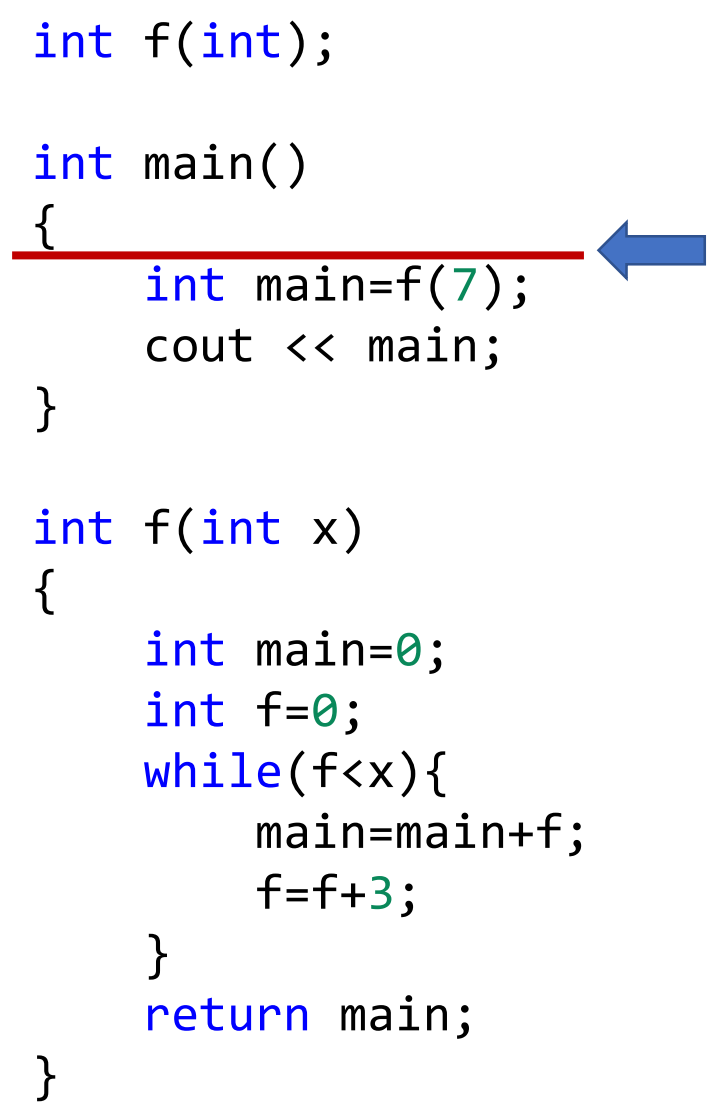
- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

main() :



```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

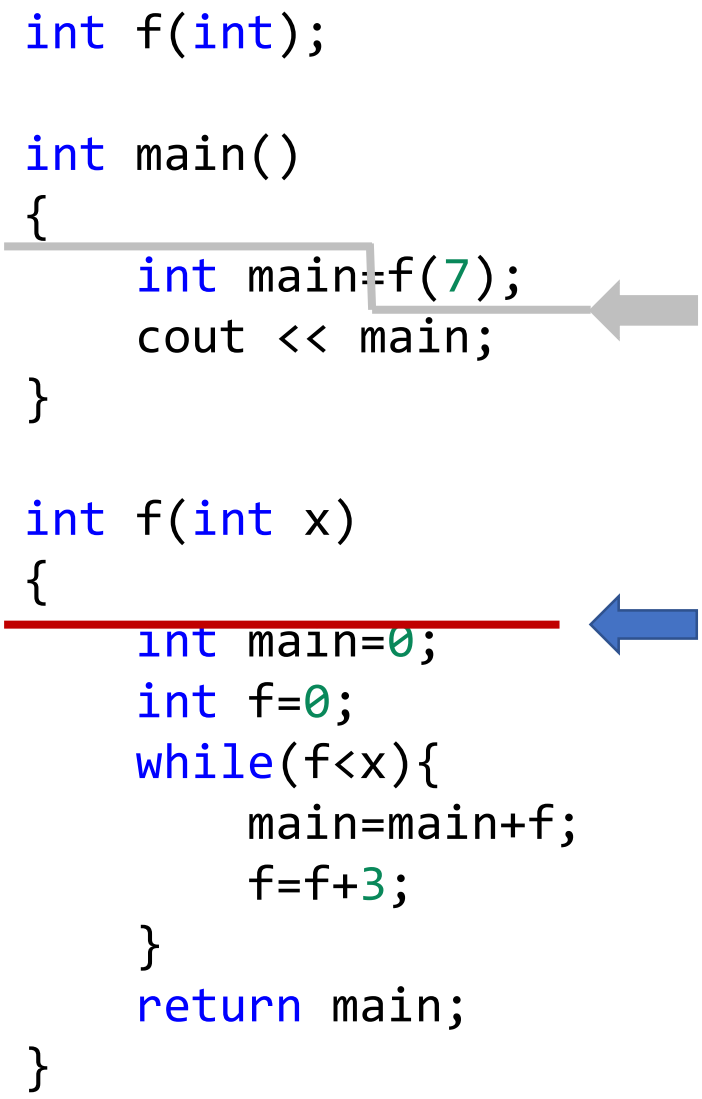
`main() : main = ?`

```
int f(int);

int main()
{
     int main=f(7);
     cout << main;
}

int f(int x)
{
     int main=0;
     int f=0;
     while(f<x){
          main=main+f;
          f=f+3;
     }
     return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   :
```

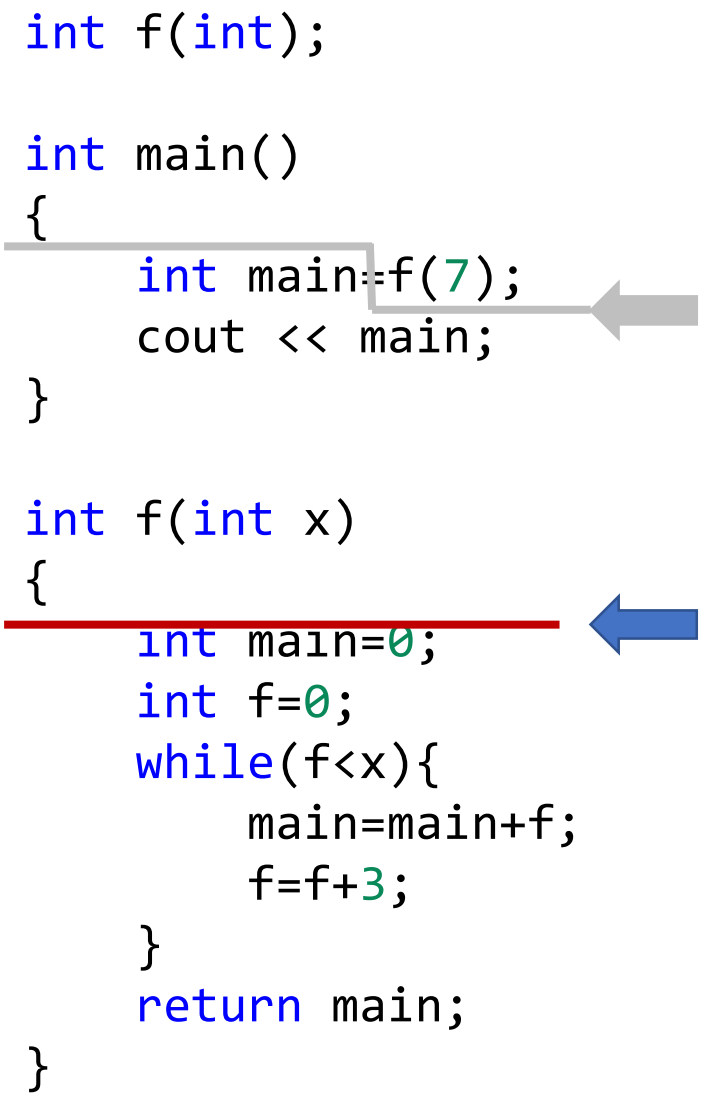


```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

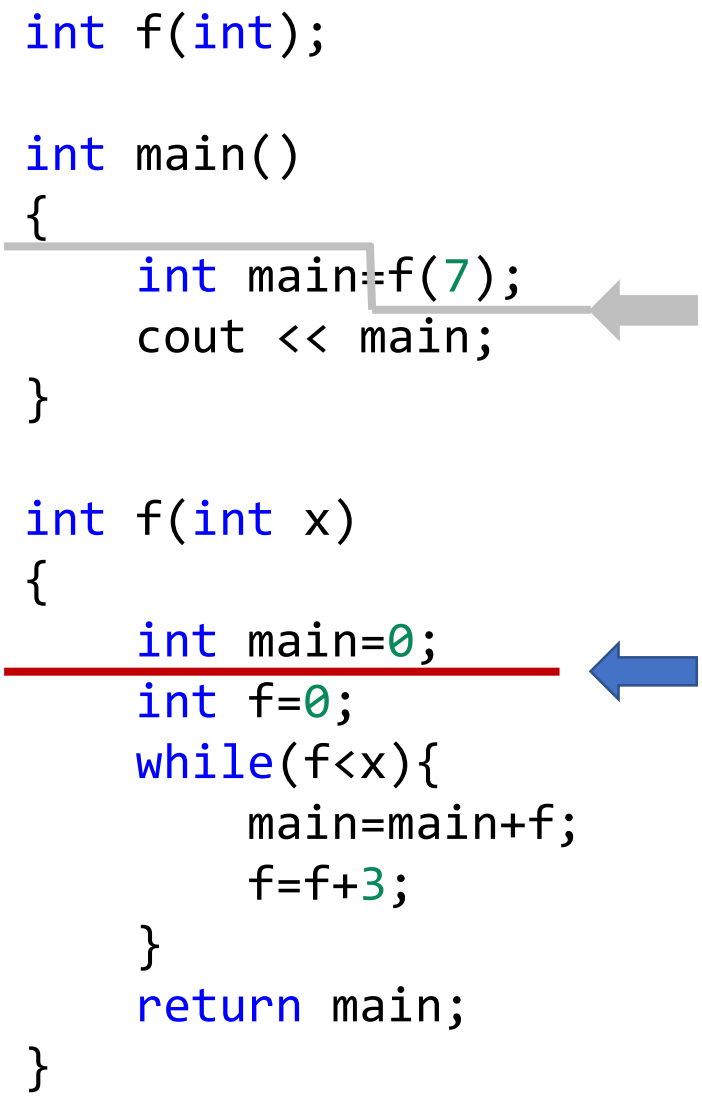
```
main() : main = ?
f(7)   : x = 7
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
```

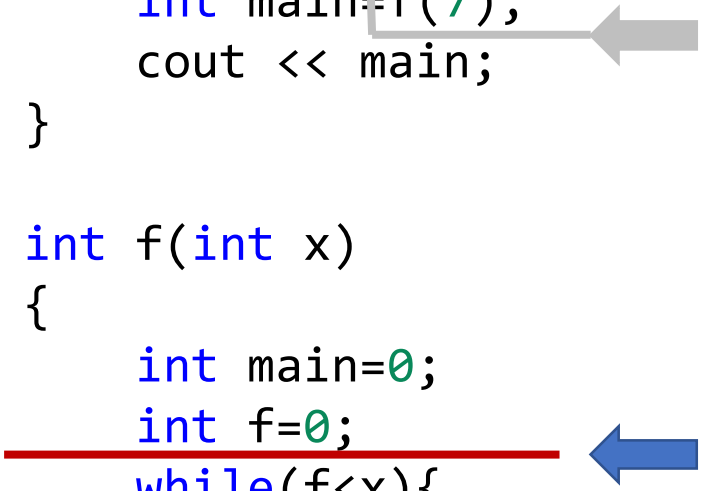


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 0
```

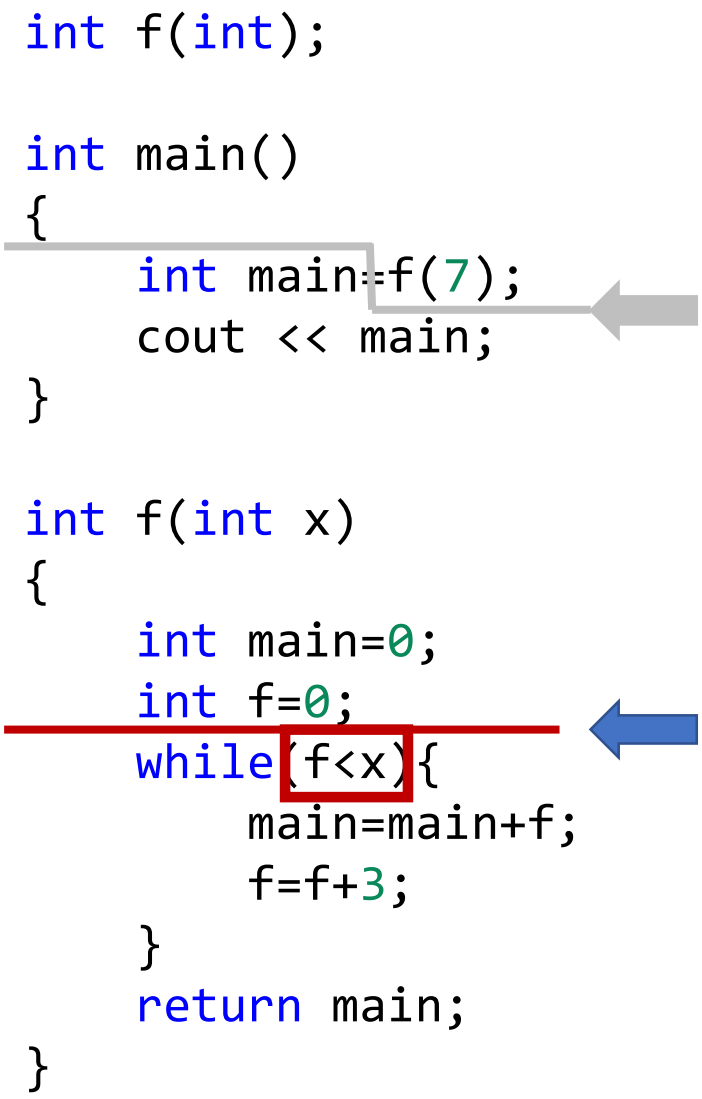
```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 0
```

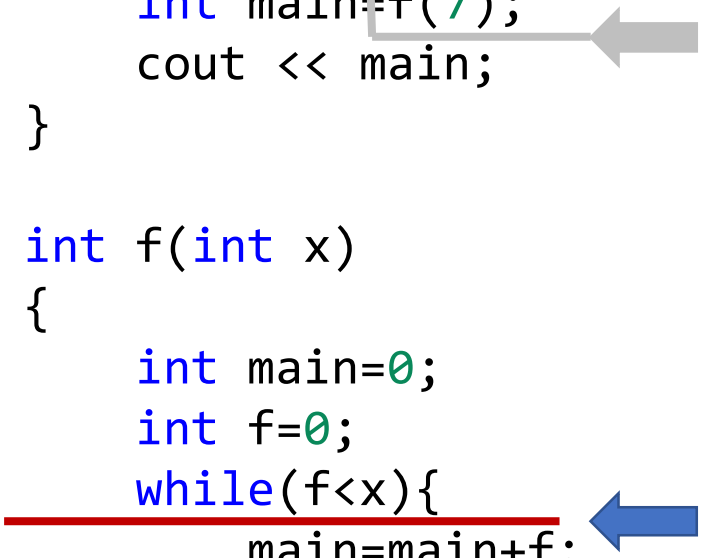


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 0
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

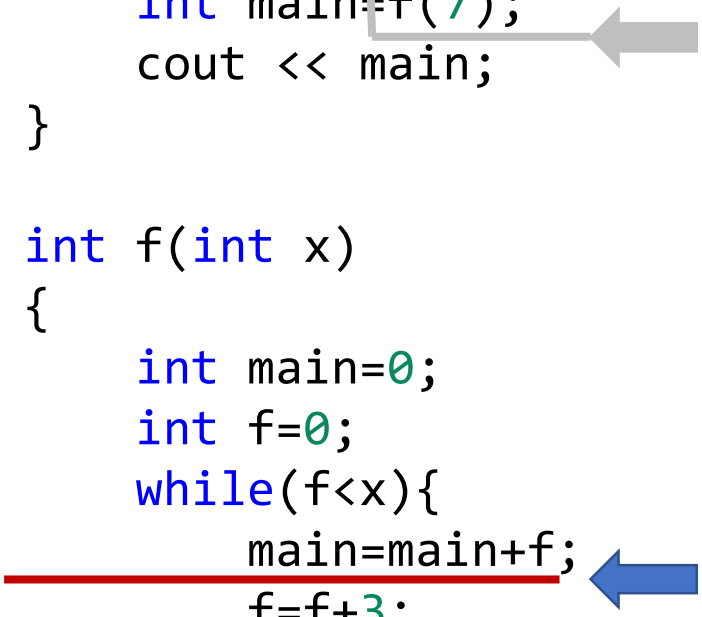


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 0
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

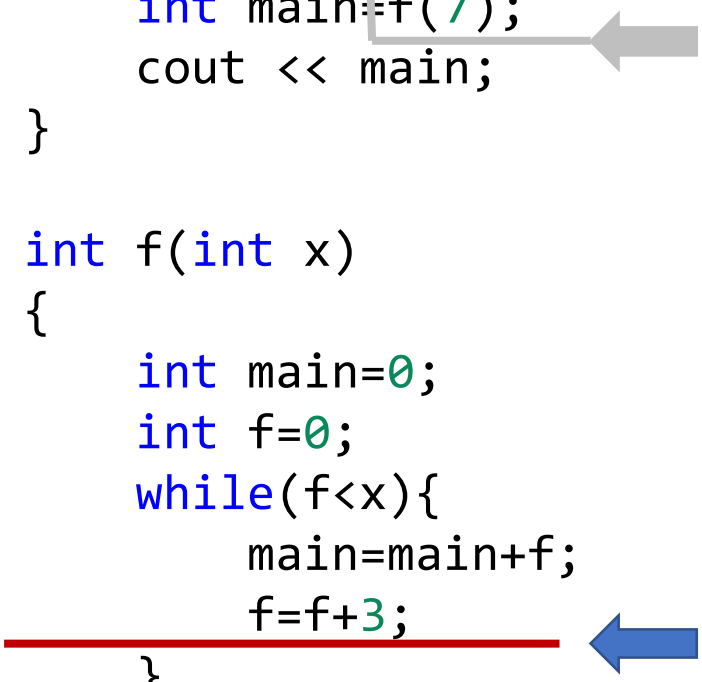


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 3
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 3
```

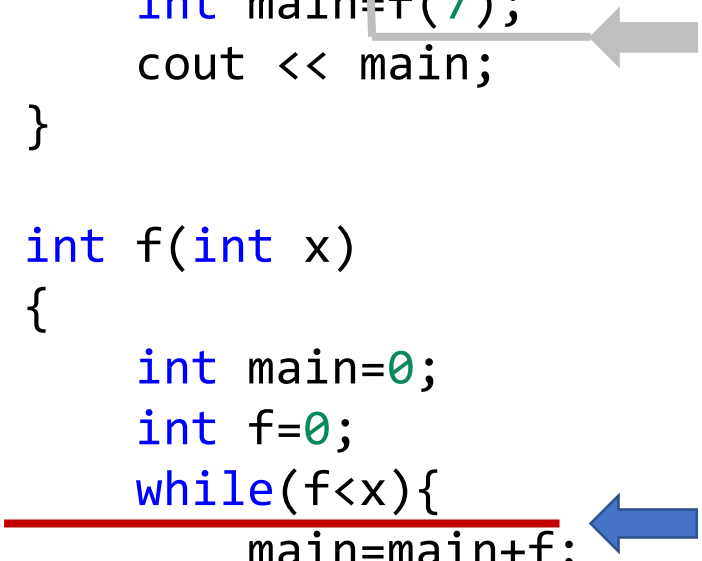
```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 0
         f = 3
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

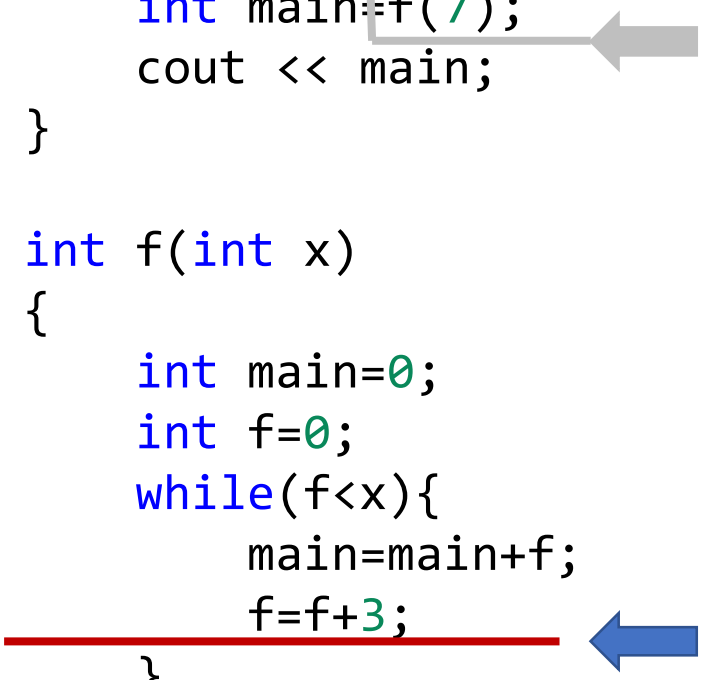


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 3
         f = 6
```

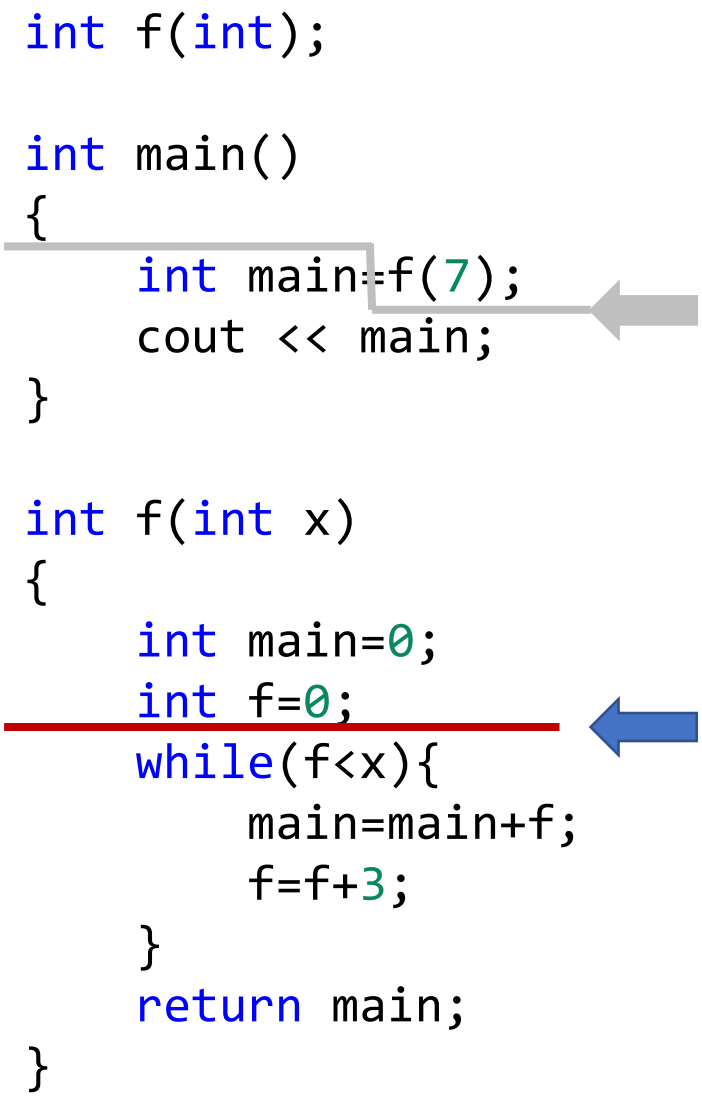
```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 3
         f = 6
```

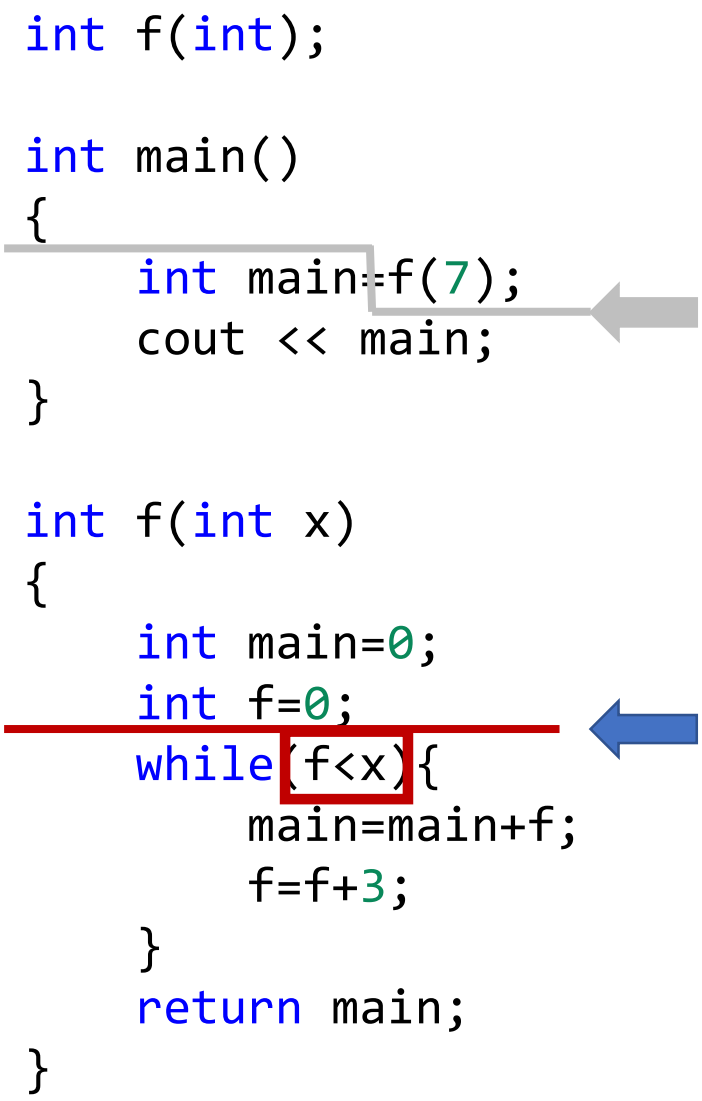


```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 3
         f = 6
```

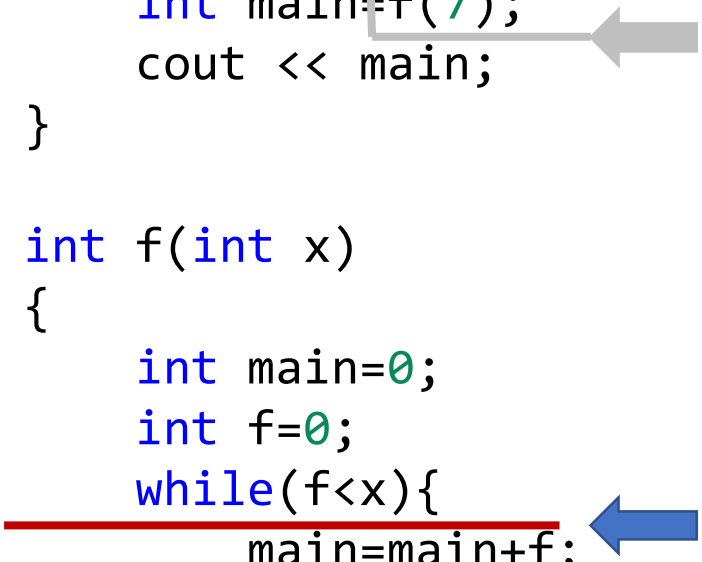


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 3
         f = 6
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

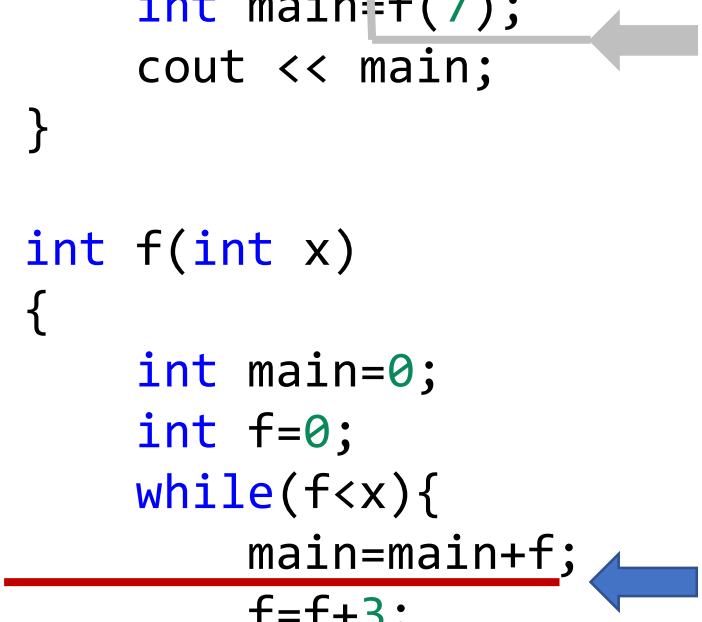


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 9
         f = 6
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

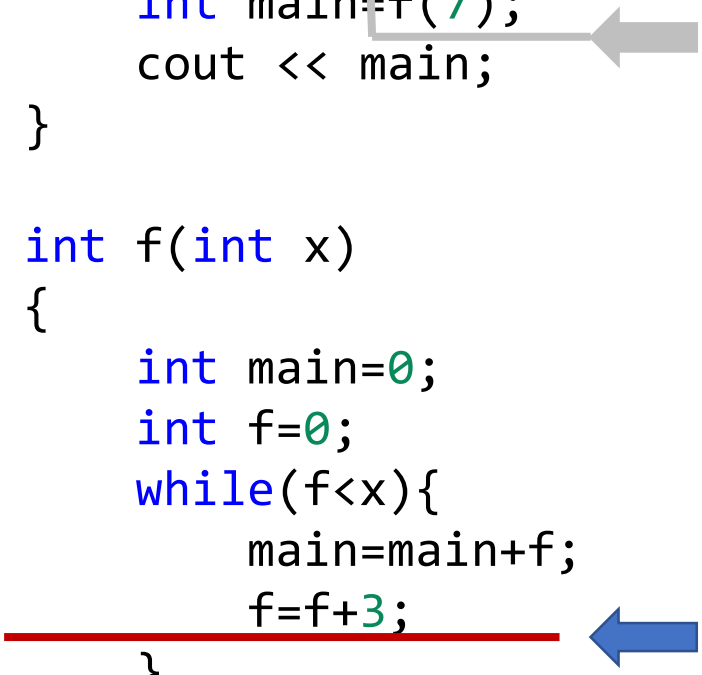


Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 9
         f = 9
```

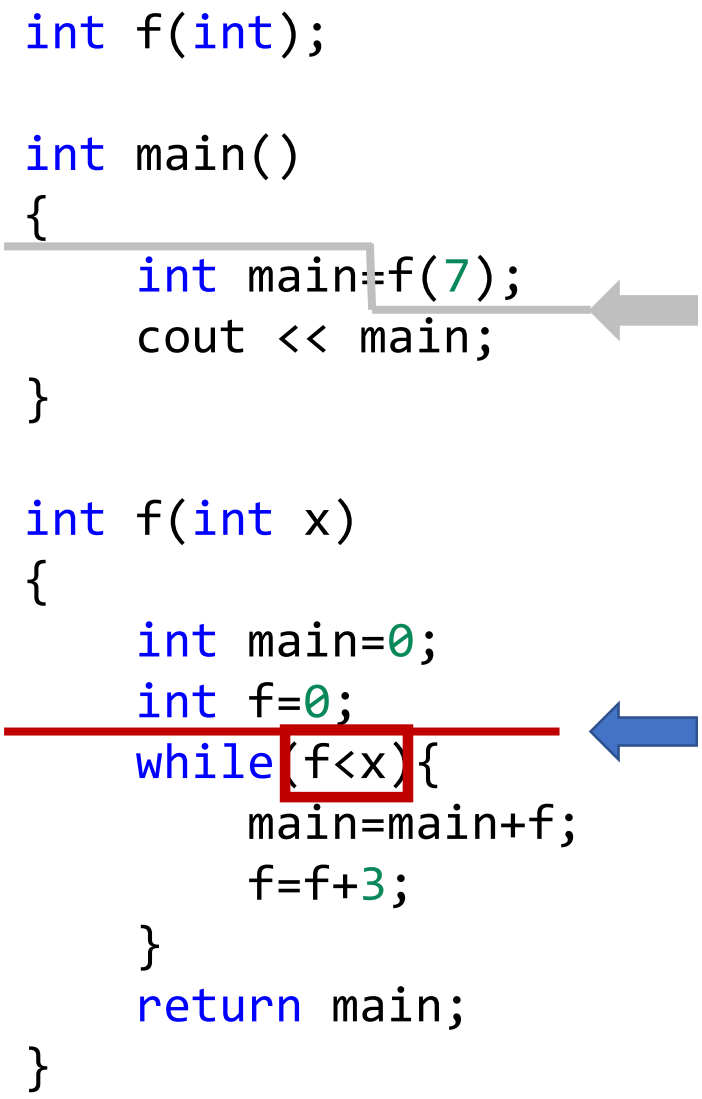
```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

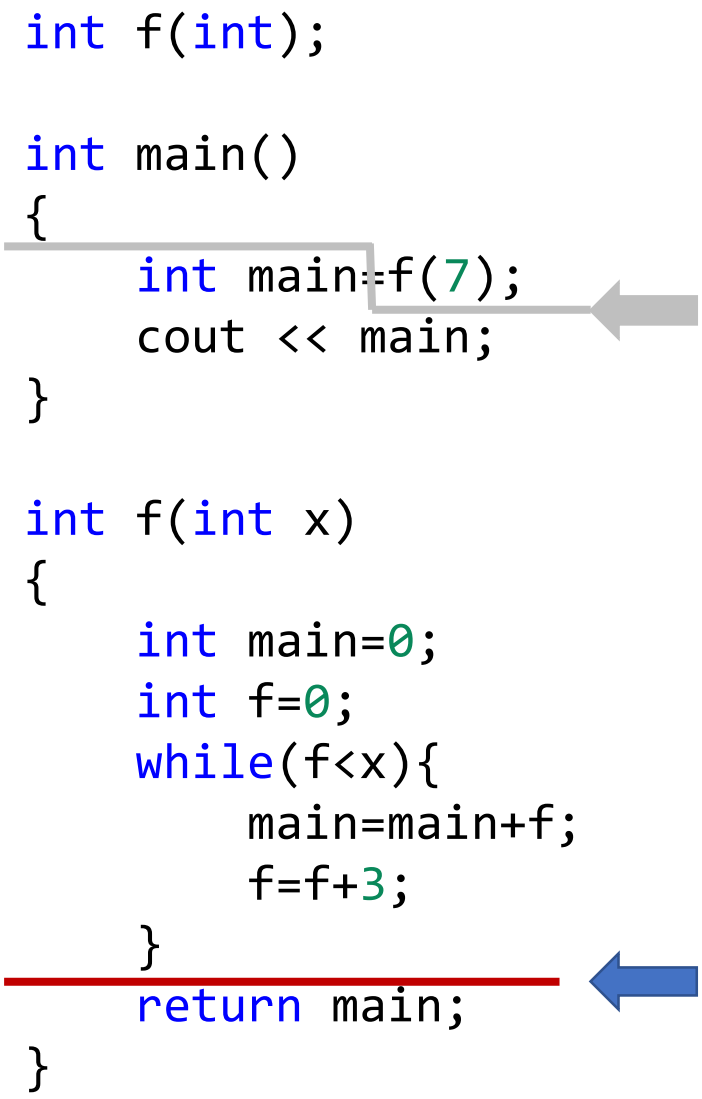
```
main() : main = ?
f(7)   : x = 7
         main = 9
         f = 9
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 9
         f = 9
```

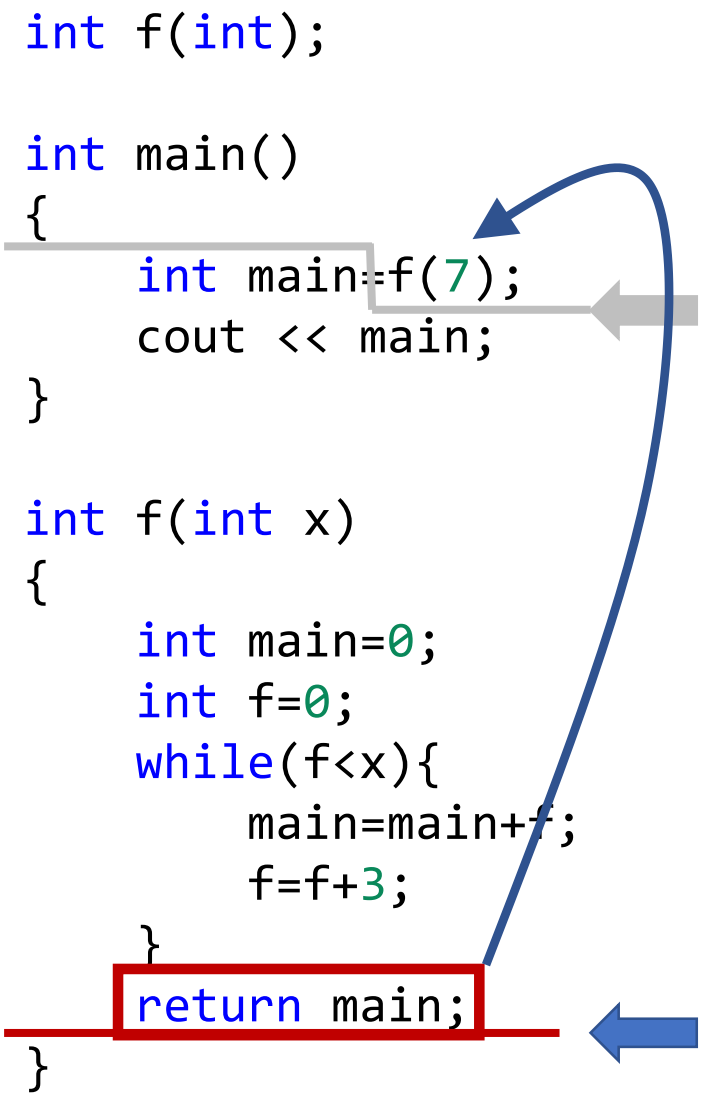


```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = ?
f(7)   : x = 7
         main = 9
         f = 9
```



Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = 9
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives

```
main() : main = 9
```

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Lifetimes : storage at run-time

- Variables need to be stored
  - At run-time they are placed somewhere in RAM
  - The lifetime determines how long the storage lives
- Scopes determine lifetime
  - Local variables live as long as the scope they are in
  - They may get shadowed, but they are still alive

```
int f(int);

int main()
{
    int main=f(7);
    cout << main;
}

int f(int x)
{
    int main=0;
    int f=0;
    while(f<x){
        main=main+f;
        f=f+3;
    }
    return main;
}
```

Recursion

Recursion versus iteration

$$x! = \begin{cases} x(x-1)!, & if\ x > 1 \\ 1 & otherwise \end{cases}$$

```
int factorial(int x)
{
    if(x>1){
        return x*factorial(x-1);
    }else{
        return 1;
    }
}
```

$$x! = \prod_{i=1}^{x} i$$

```
int factorial(int x)
{
    int acc = 1;
    for(int i = 1; i<=x ; i=i+1 ){
        acc = acc * i;
    }
    return acc;
}
```

Recursion is not special

- Recursive functions are completely normal
  - Follow the same rules we've already seen
  - No new syntax or meaning is needed
- (Any?) problems with recursion are conceptual
  - Sometimes people find recursion difficult to grasp
  - Usually means understanding of normal functions is unclear
- The benefits of recursion are conceptual
  - Can express functions in a more natural way
  - May be clearer how code relates to maths
  - Often useful for data-structures

Lifetimes and recursion

```
int factorial(int x)
{
    if(x>1){
        return x * factorial(x-1);
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

main() :

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

main() :

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
```

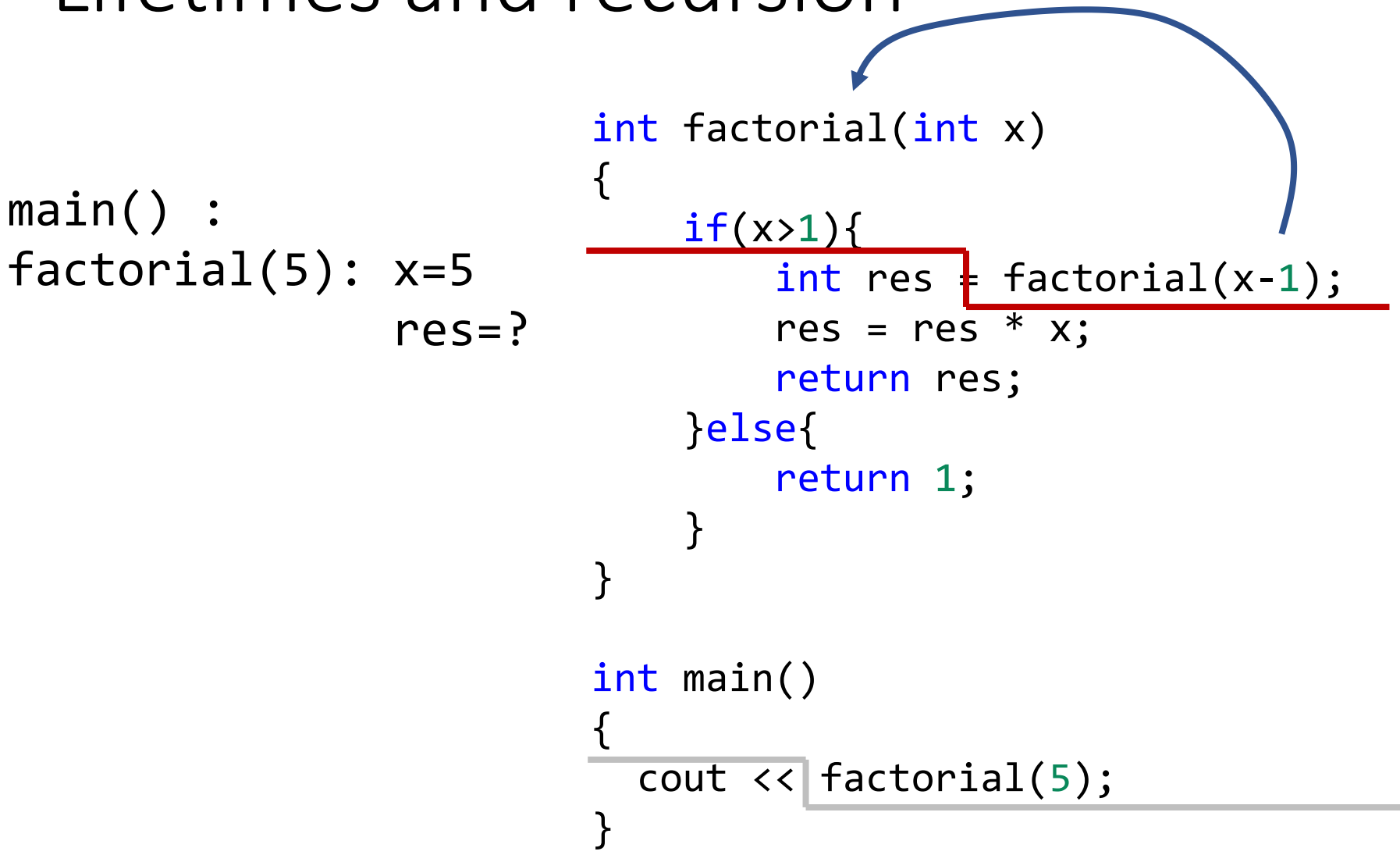
```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion



Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

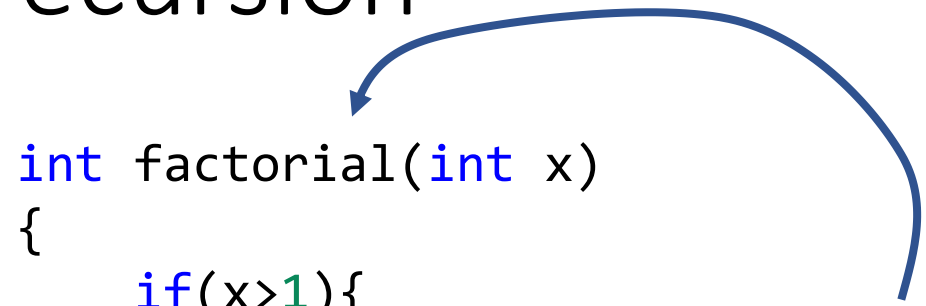
Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
```



```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

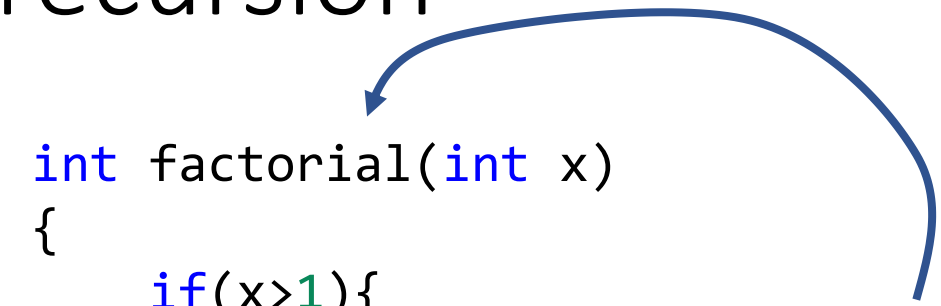
Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
```



```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

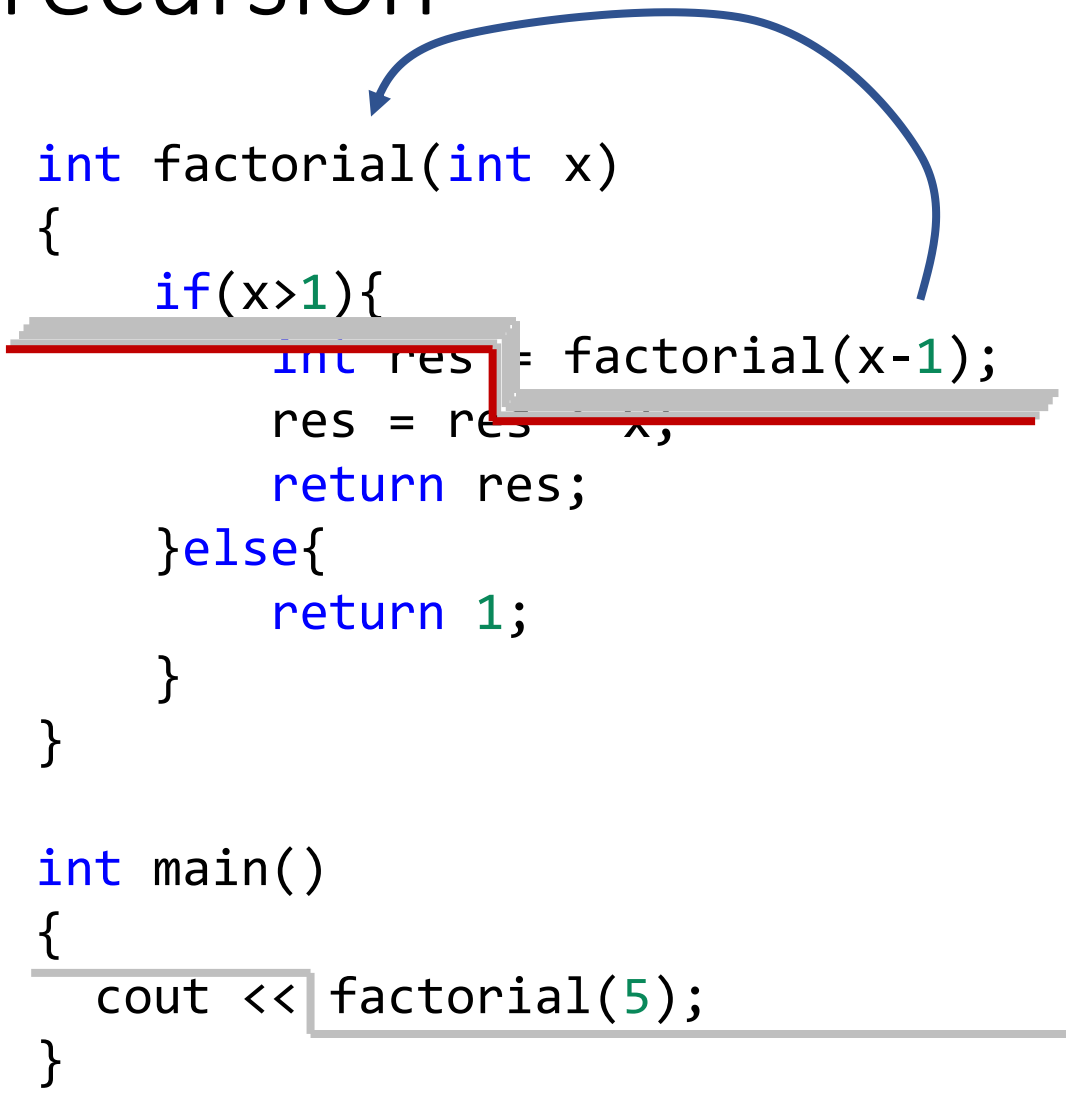
Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
factorial(1): x=1
```



```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
factorial(1): x=1
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

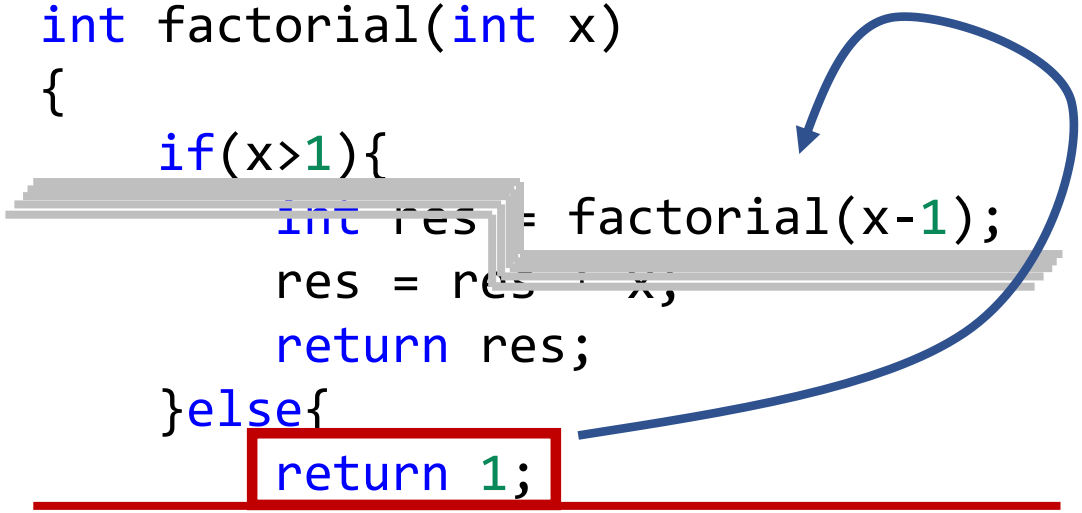
```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
factorial(1): x=1
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
factorial(1): x=1
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```



Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=1
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

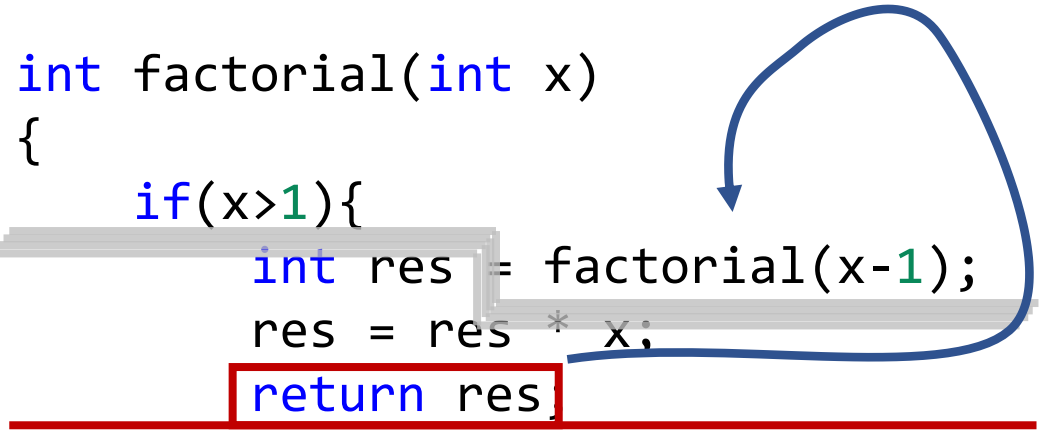
```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=2
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=2
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```



Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=2
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

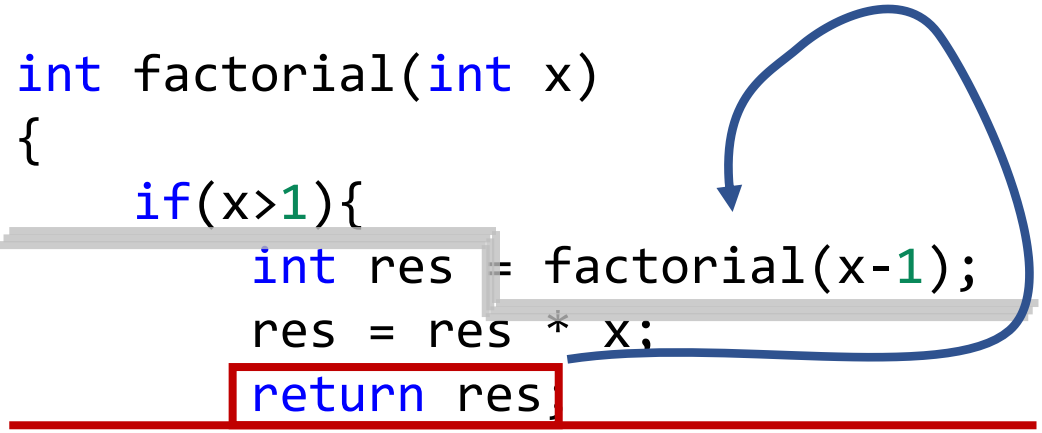
Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=6
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

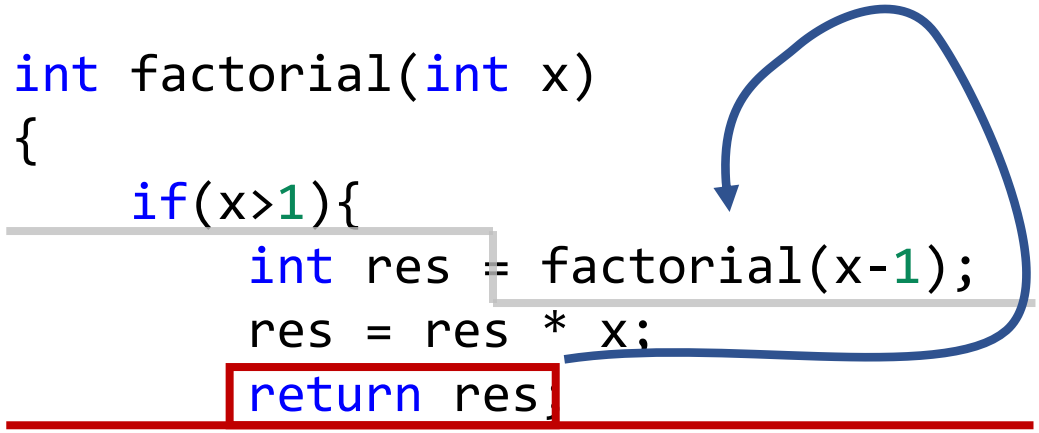
```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=6
```



```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=24
```



```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=5
              res=24
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

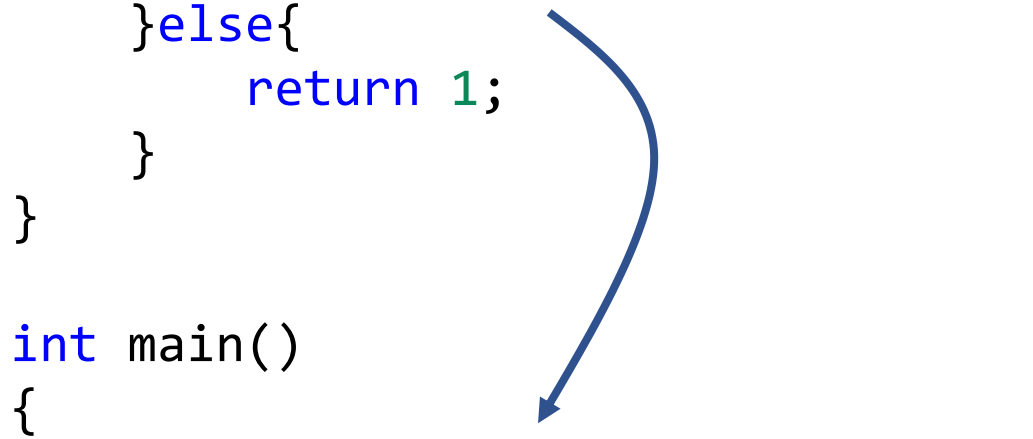
```
main() :
factorial(5): x=120
              res=24
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Lifetimes and recursion

```
main() :
factorial(5): x=120
              res=24
```

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```



Lifetimes and recursion

main() :

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Recursion : concepts

- *Stack* : a place for live but inactive functions
  - For each inactive function call we must remember:
    1. Function parameters
    2. Local variables
    3. Next point of execution in the function
- Recursive functions need at least two cases
  - *Base case* : a non-recursive return path
  - *Recursive step* : a path calling itself

Recursion : concepts

```
main() :
factorial(5): x=5
              res=?
factorial(4): x=4
              res=?
factorial(3): x=3
              res=?
factorial(2): x=2
              res=?
factorial(1): x=1
```

Stack of function calls

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Recursion : concepts

$$x! = \begin{cases} x(x-1)!, & if\ x > 1 \\ 1 & otherwise \end{cases}$$

Base case

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Recursion : concepts

$$x! = \begin{cases} x(x-1)!, & if\ x > 1 \\ 1 & otherwise \end{cases}$$

Recursive case

```
int factorial(int x)
{
    if(x>1){
        int res = factorial(x-1);
        res = res * x;
        return res;
    }else{
        return 1;
    }
}

int main()
{
  cout << factorial(5);
}
```

Stack overflow

- What if the base-case never happens?

$$x! = \begin{cases} 1, & if\ x = 1 \\ x(x-1)! & otherwise \end{cases}$$

```
int factorial(int x)
{
  if(x==1){
    return 1;
 }else{
    return x*factorial(x-1);
  }
}

int main()
{
  cout << factorial(5);
}
```

Stack overflow

- What if the base-case never happens?

$$x! = \begin{cases} 1, & if\ x = 1 \\ x(x-1)! & otherwise \end{cases}$$

```
main()
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

```
int factorial(int x)
{
  if(x==1){
    return 1;
 }else{
    return x*factorial(x-1);
  }
}

int main()
{
  cout << factorial(5);
}
```

Stack overflow

- What if the base-case never happens?

$$x! = \begin{cases} 1, & if\ x = 1 \\ x(x-1)! & otherwise \end{cases}$$

```
main()
factorial(0)
factorial(-1)
factorial(-2)
factorial(-3)
factorial(-4)
factorial(-5)
factorial(-6)
factorial(-7)
```

```
int factorial(int x)
{
  if(x==1){
    return 1;
 }else{
    return x*factorial(x-1);
  }
}

int main()
{
  cout << factorial(0);
}
```

Avoiding stack overflow

1. Always have a base-case
   - If all paths are recursive it will always stack overflow
2. Make recursion conditions exhaustive
   - Every possible input leads to a defined case
3. Something should get “smaller” on each call
   - E.g a number getting smaller, a vector getting shorter
   - Each sub-problem should be “easier”

Real-world stack overflow

- The “stack” is something implemented at run-time
  - Designed to make function calls very fast
  - Often has special support in the CPU
- Stack storage is fairly limited in size
  - ~1000-10000 recursive calls may cause stack overflow
- Your program will crash in a (fairly) obvious way

- Sometimes you might want to rewrite recursion into iteration due to stack size limitations

Recursion vs iteration

- All recursive functions can be rewritten to iteration
  - Sometimes you need to implement a stack (use a vector)
  - The conversion process can be quite tough

- All iterative functions can be rewritten to recursion
  - May not work in practice due to stack size limitations
  - May introduce overhead due to function calls

Dual recursion

```
int fibonacci(int n)
{
    if(n<=1){
        return 1;
    }else{
        int a=fibonacci(n-1);
        int b=fibonacci(n-2);
        return a+b;
    }
}
```

Most non-trivial recursive function recurse twice or more

Can this be rewritten as iteration?

Recursion: hints and tips

- Use recursion if it fits the problem
  - Many mathematical definitions are recursive
- Use recursion if it is simpler to describe
  - Data-structures often fit well with recursions
- Try to guess roughly how many times it will recurse
  - More than ~1000: maybe rewrite as iteration?

- In general: choose functional and simple first
  - Then think about optimisation and re-writing

Summary of where we are

- You now know all the basics of programming
  - IO + variables + if/while/for + types + functions
- This is “programming” up till about the 1980s
  - Linux is still written using a subset of these constructs
  - You can write very substantial programs and solutions
  - You might find it difficult to write fast programs though
- Next lecture: examples of doing useful stuff
- Next quarter: low-level aspects of C++