C++ course

Sébastien Ponce sebastien.ponce@cern.ch

CERN

ESIPAP 2018

Foreword

What this course is not

- It is not for absolute beginners
- It is not for experts
- It is not complete at all (would need 3 weeks...)
  - although is it already too long for the time we have
  - 303 slides...., 15 exercises

How I see it

Adaptative pick what you want

Interactive tell me what to skip/insist on

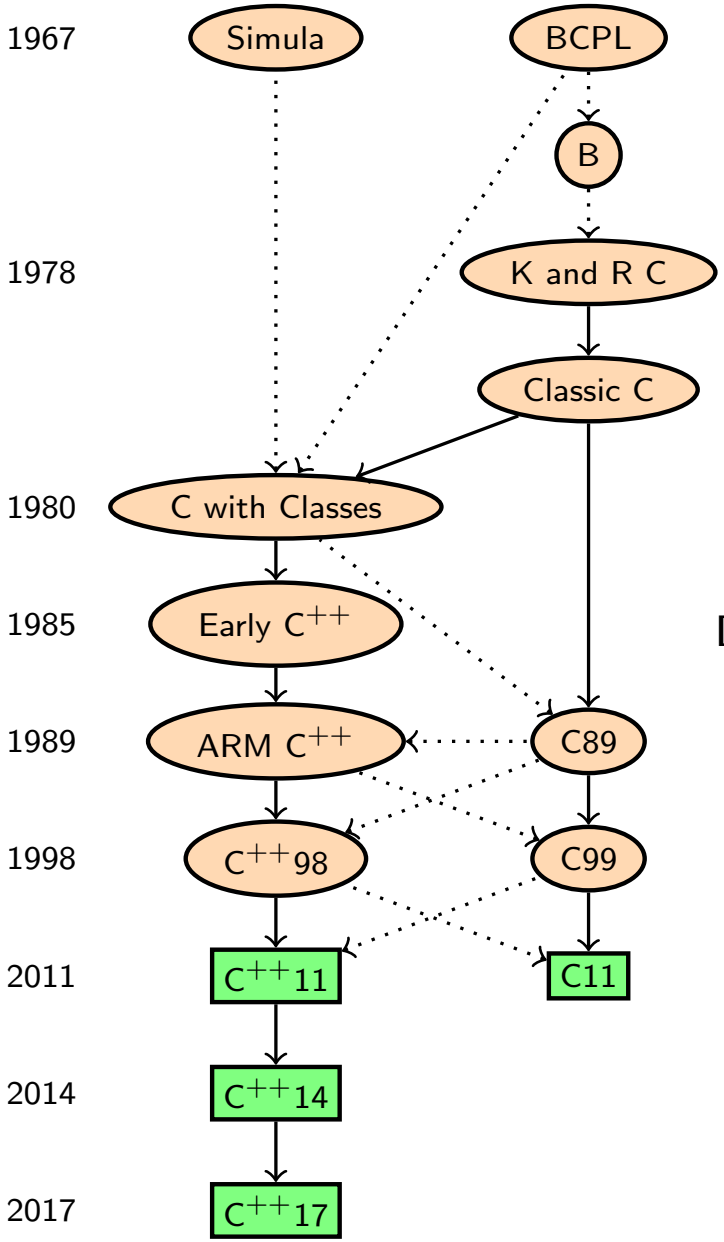
Practical let's spend time on real code

Outline

Detailed outline

History and goals

C/C++ origins



C inventor
Dennis M. Ritchie

C++ inventor
Bjarne Stroustrup

- Both C and C++ are born in Bell Labs
- *C++ almost embeds C*
- *C and C++ are still under development*
- *we will discuss C++98 first*
- *and C++11/14/17 in a second step*

Why is C++our language of choice ?

Why is C++our language of choice ?

Adapted to large projects

- strongly typed
- object oriented
- widely used (and teached)
- many available libraries

Why is C++ our language of choice ?

Adapted to large projects

- strongly typed
- object oriented
- widely used (and teached)
- many available libraries

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed

Why is C++ our language of choice ?

Adapted to large projects

- strongly typed
- object oriented
- widely used (and teached)
- many available libraries

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed

What we get

- the most powerful language
- the most complicated one
- the most error prone ?

Langage basics (C and C++)

Core syntax and types

Hello World

```
1  #include <iostream>
2
3  // This is a function
4  void print(int i) {
5    std::cout << "Hello, world " << i << std::endl;
6  }
7
8  int main(int argc, char** argv) {
9    int n = 3;
10   for (int i = 0; i < n; i++) {
11     print(i);
12   }
13   return 0;
14 }
```

Comments

```
1  // simple comment for integer declaration
2  int i;
3
4  /* multiline comment
5   * in case we need to say more
6   */
7  double d;
8
9  /**
10  * Best choice : doxygen compatible comments
11  * \fn bool isOdd(int i)
12  * \brief checks whether i is odd
13  * \param i input
14  * \return true if i is odd, otherwise false
15  */
16 bool isOdd(int i);
```

Basic types(1)

```
1  bool b = true;                  // boolean, true or false
2
3  char c = 'a';                   // 8 bits ASCII char
4  char* s = "a C string";         // array of chars ended by \0
5  string t = "a C++ string";      // class provided by the STL
6
7  char c = -3;                    // 8 bits signed integer
8  unsigned char c = 4;            // 8 bits unsigned integer
9
10 short int s = -444;             // 16 bits signed integer
11 unsigned short s = 444;         // 16 bits unsigned integer
12 short s = -444;                 // int is optional
```

Basic types(2)

```
1  int i = -123456;           // 32 bits signed integer
2  unsigned int i = 1234567; // 32 bits signed integer
3
4  long l = 0L                // 32 or 64 bits (ptr size)
5  unsigned long l = 0UL;     // 32 or 64 bits (ptr size)
6
7  long long ll = 0LL;        // 64 bits signed integer
8  unsigned long long l = 0ULL; // 64 bits unsigned integer
9
10 float f = 1.23f;           // 32 (23+7+1) bits float
11 double d = 1.23E34;        // 64 (52+11+1) bits float
```

Portable numeric types

One needs to include specific header

```
1  #include <cstdint>
2
3  int8_t c = -3;      // 8 bits, replaces char
4  uint8_t c = 4;      // 8 bits, replaces unsigned char
5
6  int16_t s = -444;   // 16 bits, replaced short
7  uint16_t s = 444;   // 16 bits, replaced unsigned short
8
9  int32_t s = -0674; // 32 bits, replaced int
10 uint32_t s = 0674; // 32 bits, replaced unsigned int
11
12 int64_t s = -0x1bc;// 64 bits, replaced long long
13 uint64_t s = 0x1bc;// 64 bits, replaced unsigned long long
```

Arrays and Pointers

Static arrays

```
1  int ai[4] = {1,2,3,4};
2  int ai[] = {1,2,3,4};  // identical
3
4  char ac[3] = {'a','b','c'};    // char array
5  char ac[4] = "abc";            // valid C string
6  char ac[4] = {'a','b','c',0}; // same valid string
7
8  int i = ai[2];  // i = 3
9  char c = ac[8]; // at best garbage, may segfault
10 int i = ai[4];  // also garbage !
```

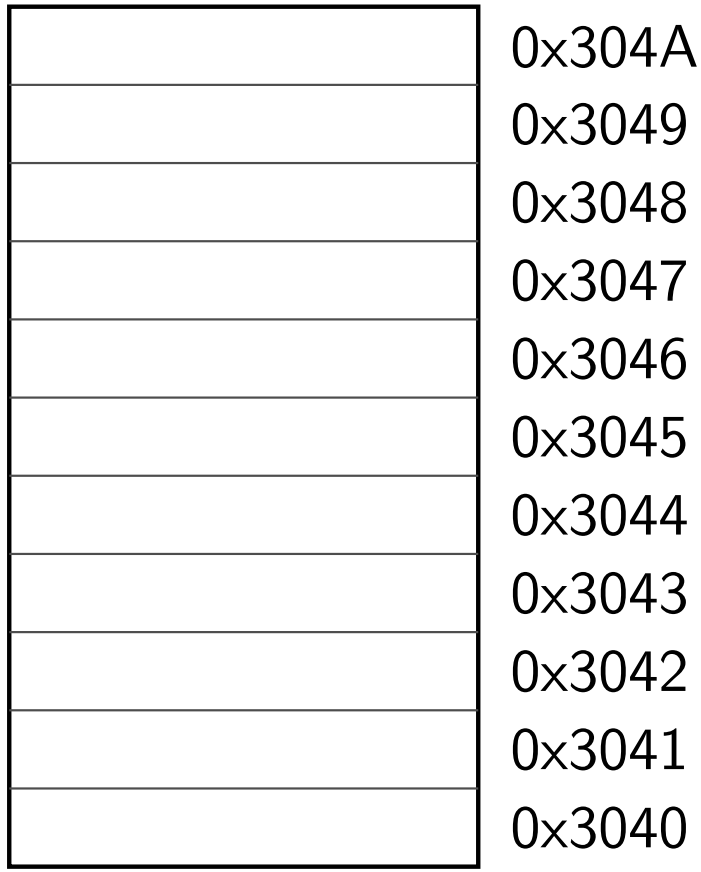
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```

Pointers

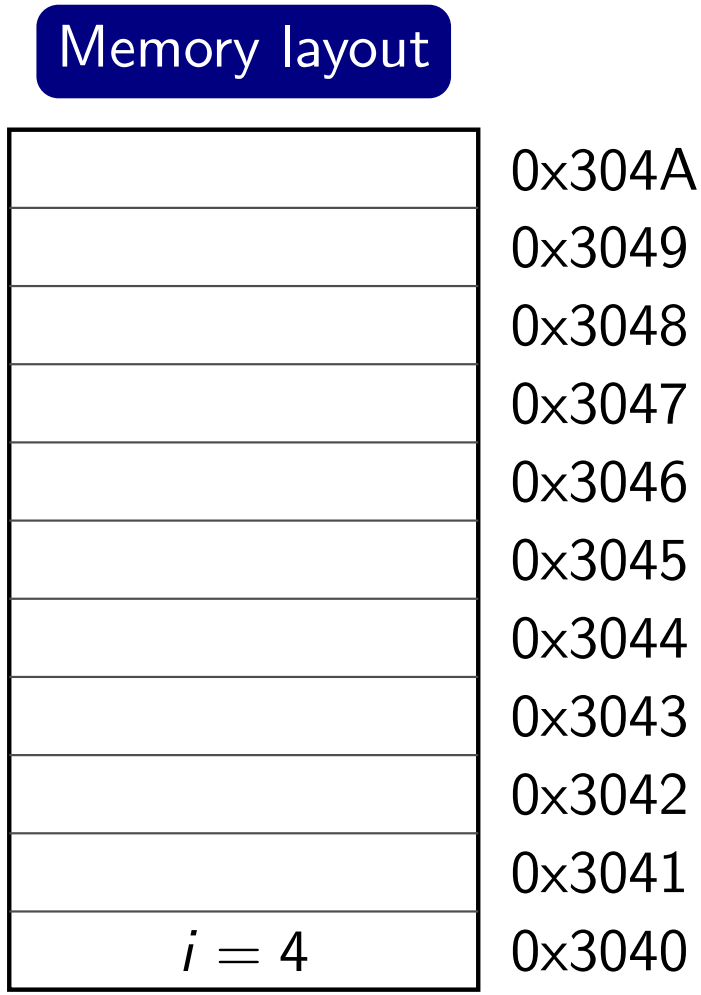
```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```

Memory layout



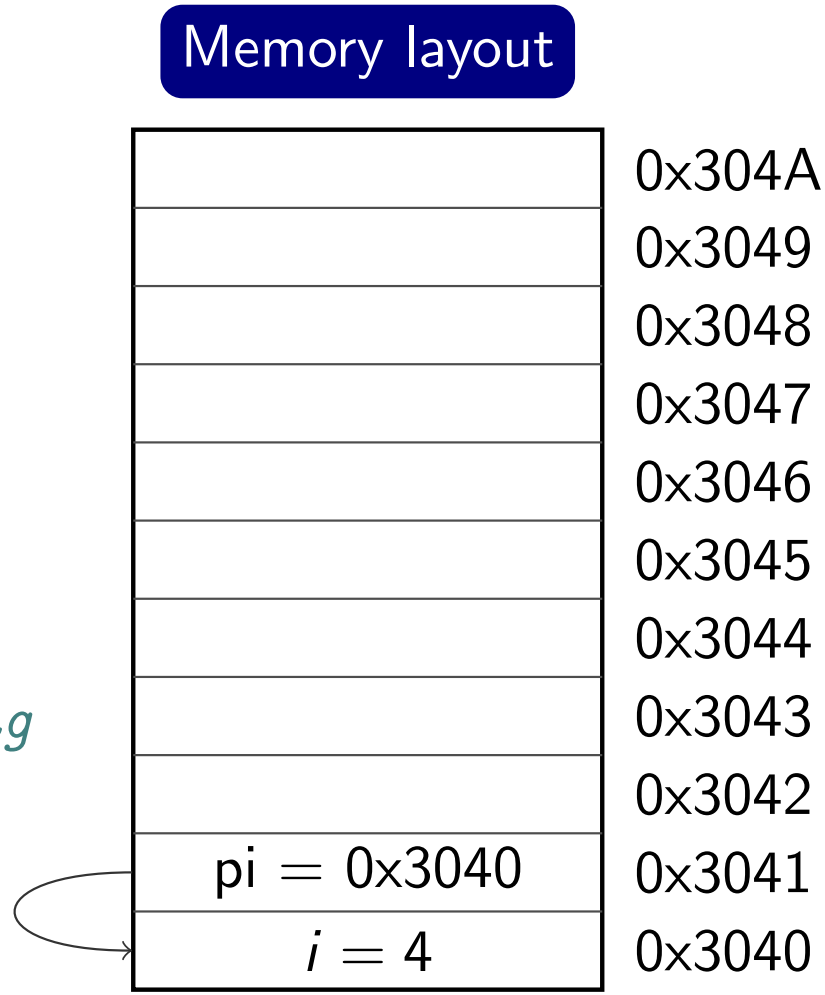
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



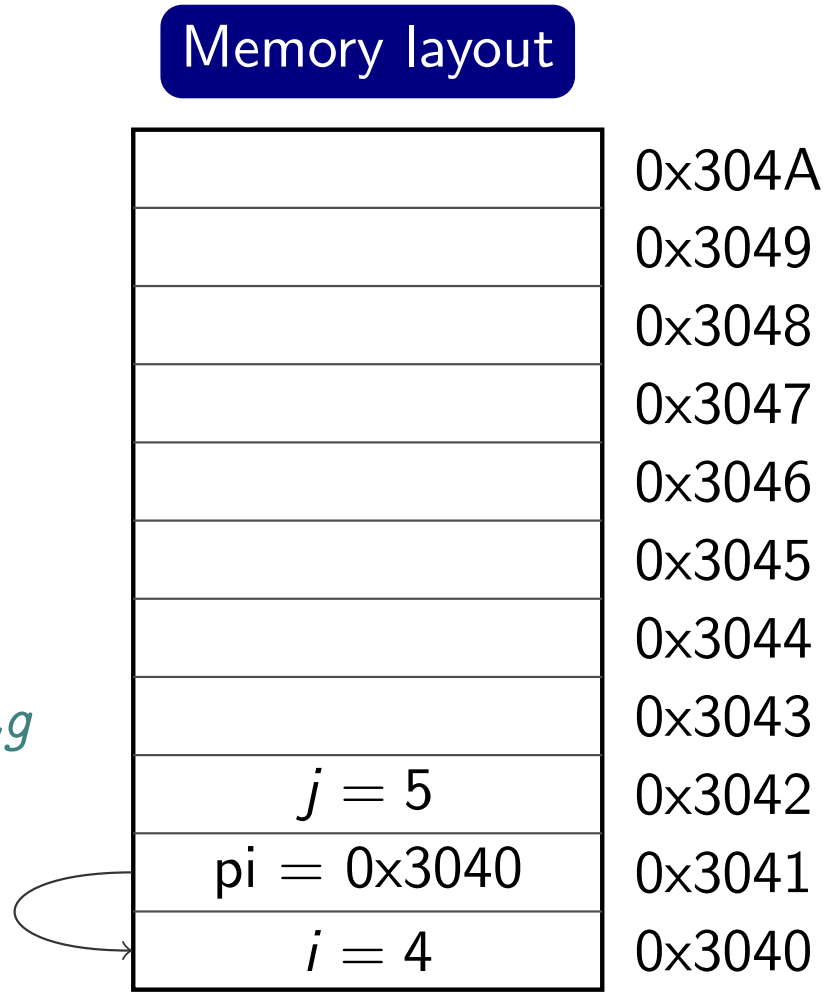
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



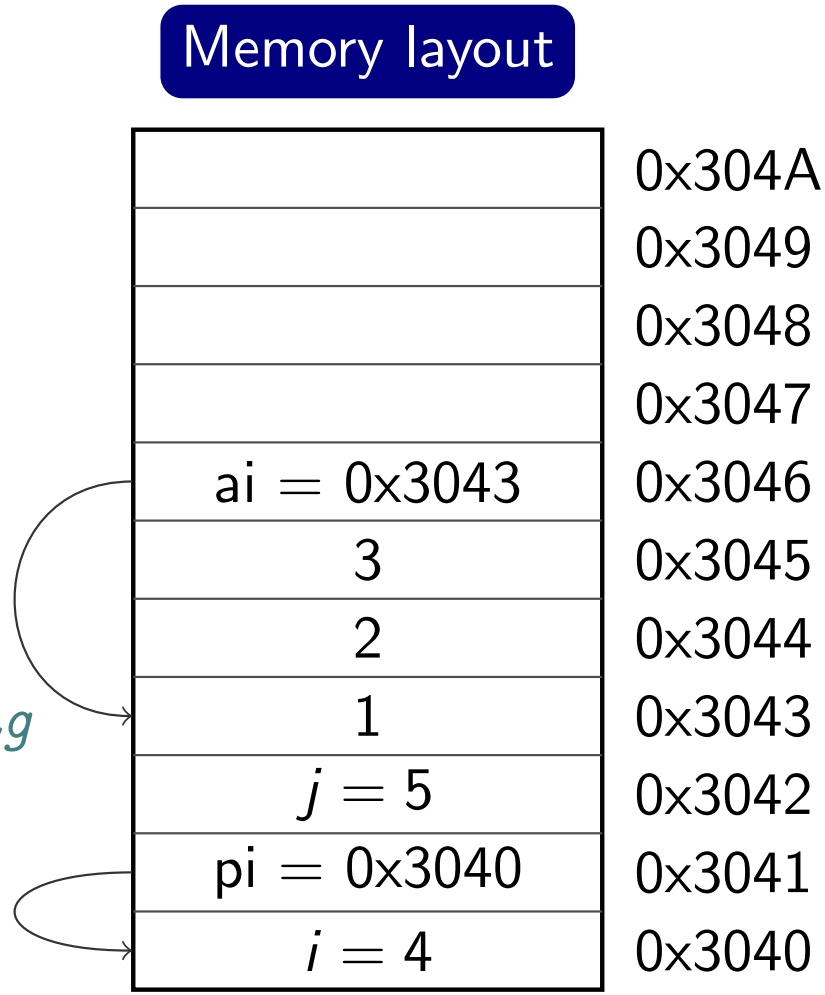
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



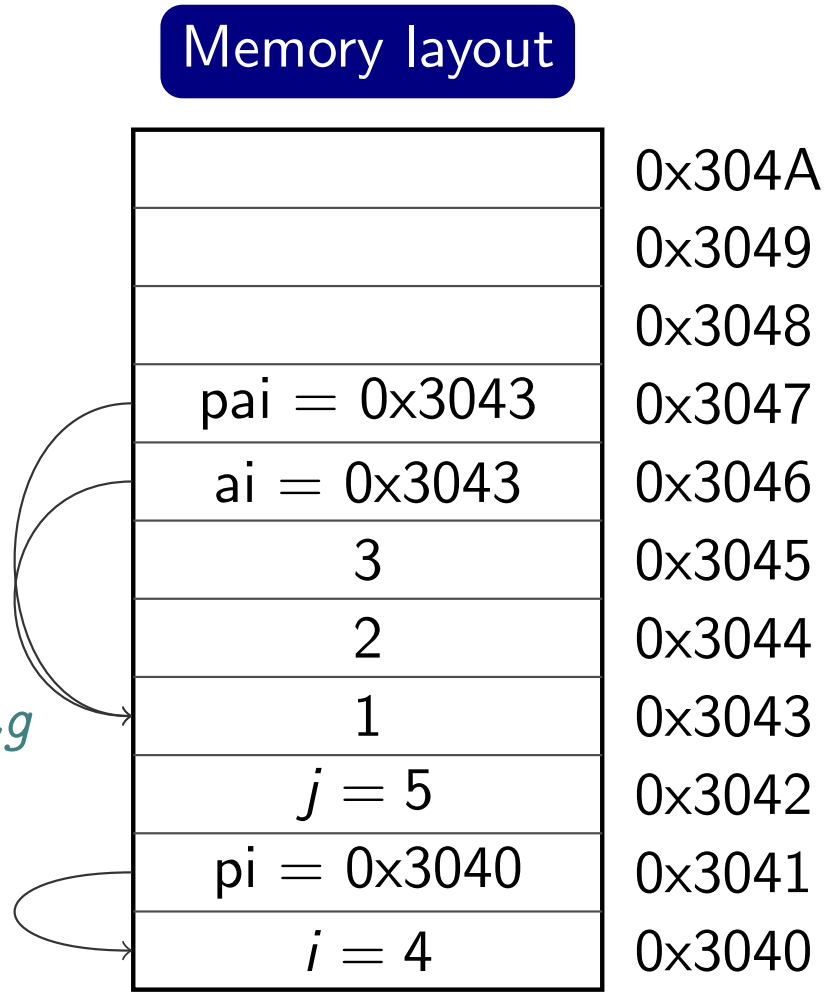
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



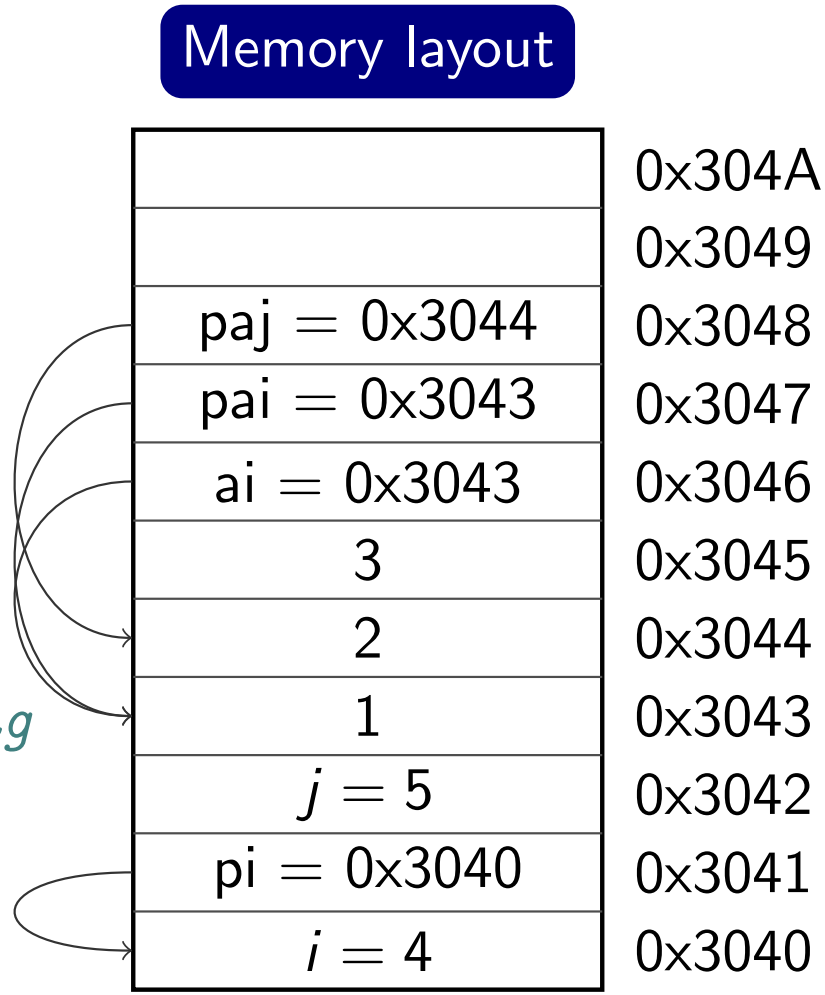
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



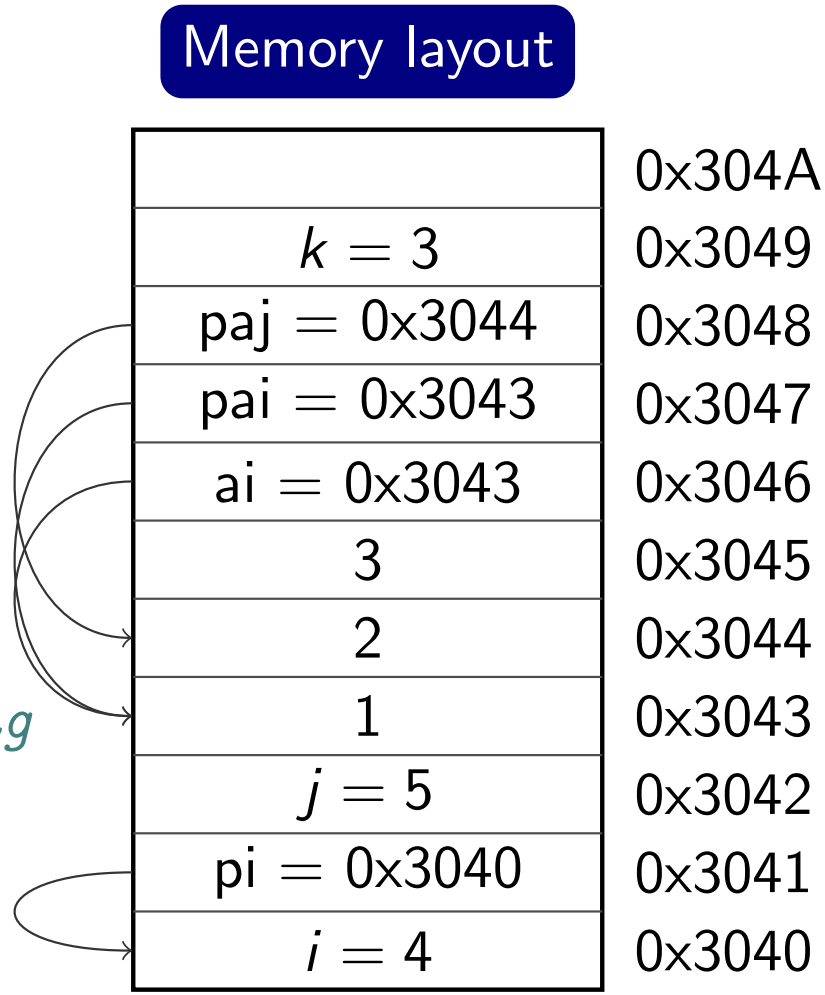
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



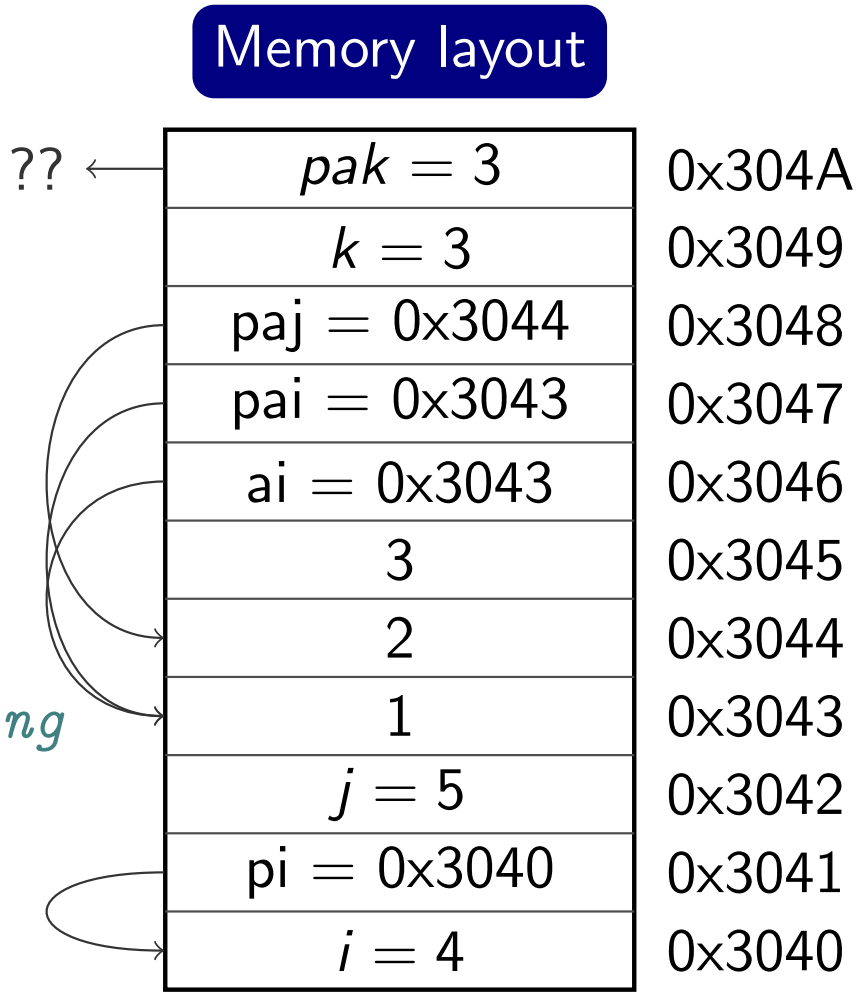
Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



Pointers

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 int *pak = k; // not compiling
11 int *pak = (int*)k;
12 int l = *pak; // seg fault !
```



Dynamic Arrays

```
1  #include <cstdlib>
2  #include <cstring>
3
4  int *bad;     // pointer to random address
5  int *ai = 0; // better. Can be tested
6
7  // allocate array of 10 ints (not initialized)
8  ai = (int*) malloc(10*sizeof(int));
9  // and set them to 0
10 memset(ai, 0, 10*sizeof(int));
11
12 // Both in one go
13 ai = (int*) calloc(10, sizeof(int));
14
15 // liberate memory
16 free(ai);
```

Operators

Operators(1)

Binary & Assignment Operators

```
int i = 1 + 4 - 2;   // 3
i *= 3;              // 9
i /= 2;              // 4
i = 23 % i;          // modulo => 3
```

Operators(1)

Binary & Assignment Operators

```
int i = 1 + 4 - 2;   // 3
i *= 3;              // 9
i /= 2;              // 4
i = 23 % i;          // modulo => 3
```

Increment / Decrement

```
int i = 0; i++; // i = 1
int j = ++i;    // i = 2, j = 2
int k = i++;    // i = 3, k = 2
int l = --i;    // i = 2, l = 2
int m = i--;    // i = 1, m = 2
```

Operators(1)

Binary & Assignment Operators

```
int i = 1 + 4 - 2;   // 3
i *= 3;              // 9
i /= 2;              // 4
i = 23 % i;          // modulo => 3
```

Increment / Decrement

Use wisely

```
int i = 0; i++; // i = 1
int j = ++i;    // i = 2, j = 2
int k = i++;    // i = 3, k = 2
int l = --i;    // i = 2, l = 2
int m = i--;    // i = 1, m = 2
```

Operators(2)

Bitwise and Assignment Operators

```
int i = 0xee & 0x55;   // 0x44
i |= 0xee;             // 0xee
i ^= 0x55;             // 0xbb
int j = ~0xee;         // 0xffffff11
int k = 0x1f << 3;     // 0x78
int l = 0x1f >> 2;     // 0x7
```

Operators(2)

Bitwise and Assignment Operators

```
int i = 0xee & 0x55;   // 0x44
i |= 0xee;             // 0xee
i ^= 0x55;             // 0xbb
int j = ~0xee;         // 0xffffff11
int k = 0x1f << 3;     // 0x78
int l = 0x1f >> 2;     // 0x7
```

Boolean Operators

```
bool a = true;
bool b = false;
bool c = a && b;     // false
bool d = a || b;     // true
bool e = !d;         // false
```

Operators(3)

Comparison Operators

```
bool a = (3 == 3);   // true
bool b = (3 != 3);   // false
bool c = (4 < 4);    // false
bool d = (4 <= 4);   // true
bool e = (4 > 4);    // false
bool f = (4 >= 4);   // true
```

Operators(3)

Comparison Operators

```
bool a = (3 == 3);   // true
bool b = (3 != 3);   // false
bool c = (4 < 4);    // false
bool d = (4 <= 4);   // true
bool e = (4 > 4);    // false
bool f = (4 >= 4);   // true
```

Precedences

```
c &= 1+(++b)|(a--)*4%5^7; // ???
```

Operators(3)

Comparison Operators

```
bool a = (3 == 3);   // true
bool b = (3 != 3);   // false
bool c = (4 < 4);    // false
bool d = (4 <= 4);   // true
bool e = (4 > 4);    // false
bool f = (4 >= 4);   // true
```

Precedences

Don't use

```
c &= 1+(++b)|(a--)*4%5^7; // ???
```

Operators(3)

Comparison Operators

```
bool a = (3 == 3);   // true
bool b = (3 != 3);   // false
bool c = (4 < 4);    // false
bool d = (4 <= 4);   // true
bool e = (4 > 4);    // false
bool f = (4 >= 4);   // true
```

Precedences

Don't use - use parenthesis

```
c &= 1+(++b)|(a--)*4%5^7; // ???
```

Compound data types

struct

"members" grouped together under one name

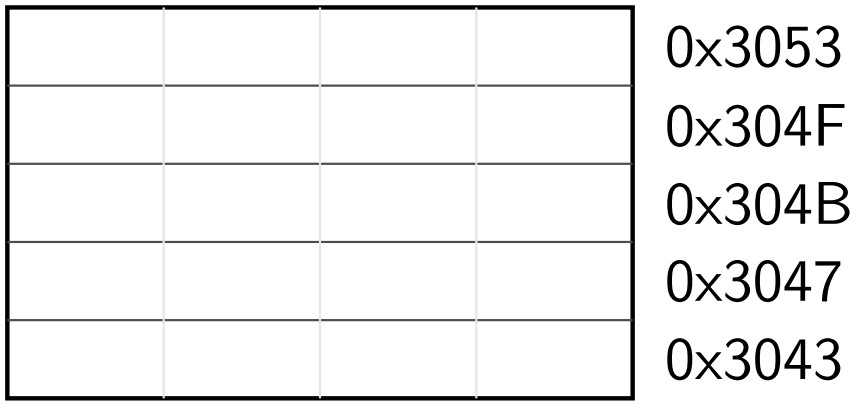
```
1  struct Individual {
2    unsigned char age;
3    float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11   .age = 45,
12   .weight=67
13 };
```

struct

"members" grouped together under one name

```
1  struct Individual {
2    unsigned char age;
3    float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11   .age = 45,
12   .weight=67
13 };
```

Memory layout

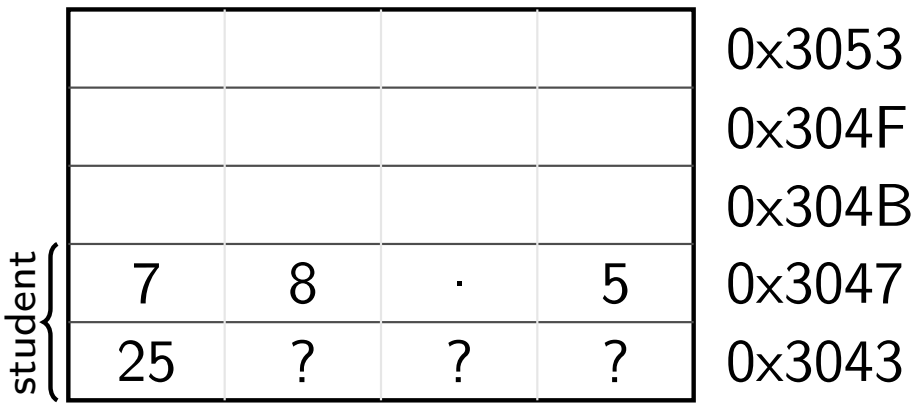


struct

"members" grouped together under one name

```
1  struct Individual {
2    unsigned char age;
3    float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11   .age = 45,
12   .weight=67
13 };
```

Memory layout

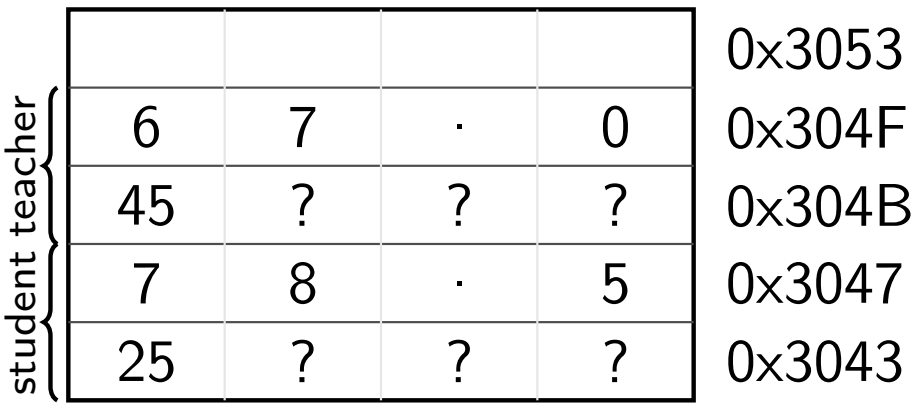


struct

"members" grouped together under one name

```
1  struct Individual {
2    unsigned char age;
3    float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11   .age = 45,
12   .weight=67
13 };
```

Memory layout



union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

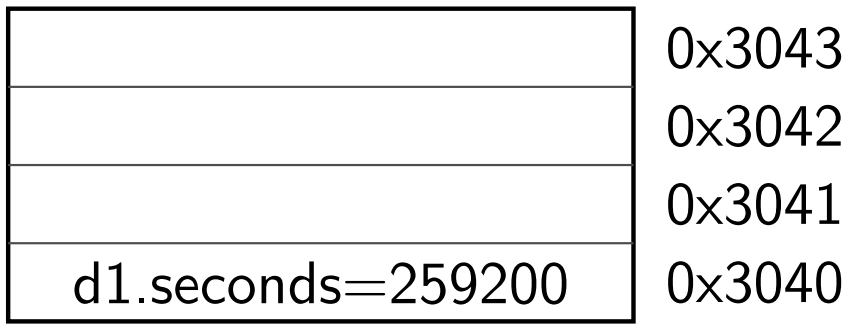
Memory layout

union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

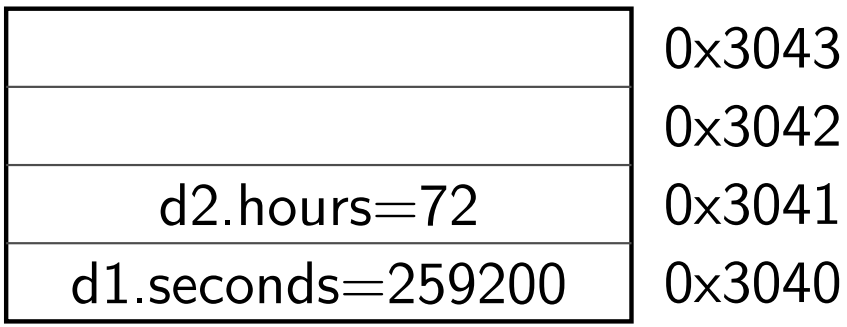


union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

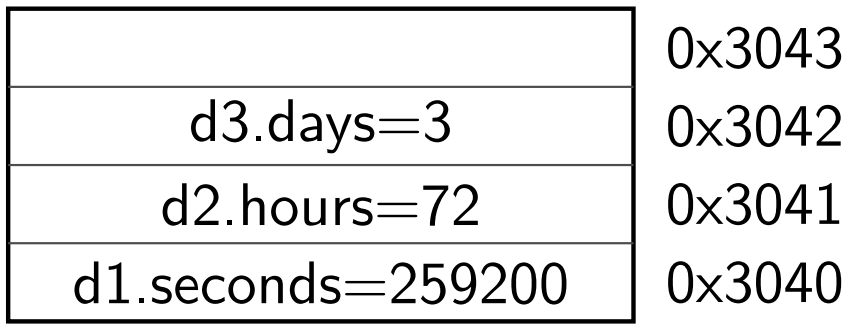


union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout



union

"members" packed together at same memory location

```
1  union Duration {
2    int seconds;
3    short hours;
4    char days;
5  };
6
7  Duration d1, d2, d3;
8  d1.seconds = 259200;
9  d2.hours = 72;
10 d3.days = 3;
11 d1.days = 3; // d1.seconds overwritten
12 int a = d1.seconds; // d1.seconds is garbage
```

Memory layout

	0x3043
d3.days=3	0x3042
d2.hours=72	0x3041
d1.days=3	0x3040

Enums and typedefs

```
1  enum VehicleType {
2    BIKE,  // 0
3    CAR,   // 1
4    BUS,   // 2
5  };
6  VehicleType t = CAR;
7
8  typedef uint64_t myint;
9  myint toto = 17;
```

```
1  enum VehicleType {
2    BIKE = 3,
3    CAR = 5,
4    BUS = 7,
5  };
6  VehicleType t2 = BUS;
```

More sensible example

```
1  enum ShapeType {
2    CIRCLE,
3    RECTANGLE
4  };
5
6  struct Rectangle {
7    float width;
8    float height;
9  };
```

More sensible example

```
1  enum ShapeType {
2    CIRCLE,
3    RECTANGLE
4  };
5
6  struct Rectangle {
7    float width;
8    float height;
9  };
```

```
10 struct Shape {
11   ShapeType type;
12   union {
13     float radius;
14     Rectangle rect;
15   };
16 };
```

More sensible example

```
1  enum ShapeType {
2    CIRCLE,
3    RECTANGLE
4  };
5
6  struct Rectangle {
7    float width;
8    float height;
9  };

17 Shape s;
18 s.type = CIRCLE;
19 s.radius = 3.4;
```

```
10 struct Shape {
11   ShapeType type;
12   union {
13     float radius;
14     Rectangle rect;
15   };
16 };

20 Shape t;
21 t.type = RECTANGLE;
22 t.rect.width = 3;
23 t.rect.height = 4;
```

Functions

Functions

```
1  // with return type
2  int square(int a) {
3    return a * a;
4  }
5
6  // multiple parameters
7  int mult(int a,
8           int b) {
9    return a*b;
10 }
```

```
11 // no parameter
12 void hello() {
13   printf("Hello World");
14 }
15
16 // no return
17 void log(char* msg) {
18   printf("%s", msg);
19 }
```

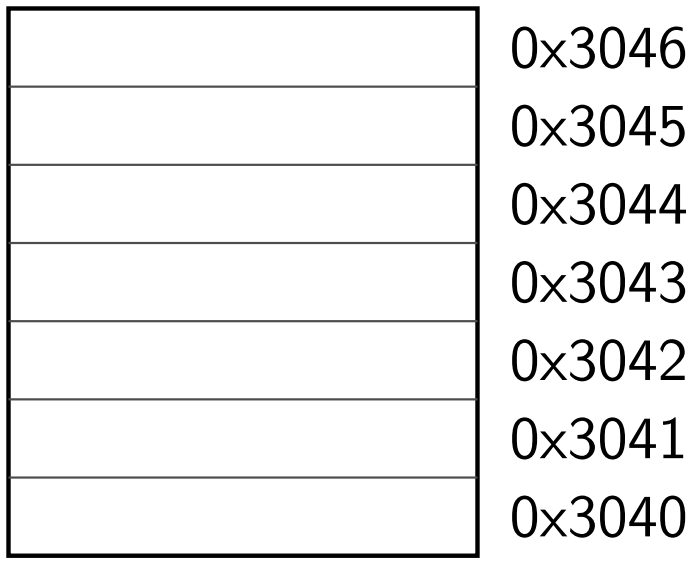
Parameter are passed by value

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```

Parameter are passed by value

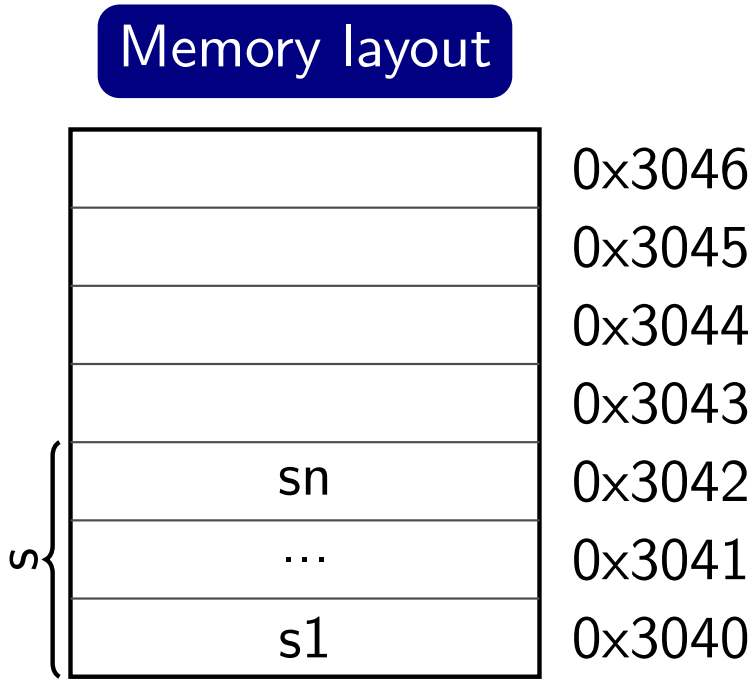
```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```

Memory layout



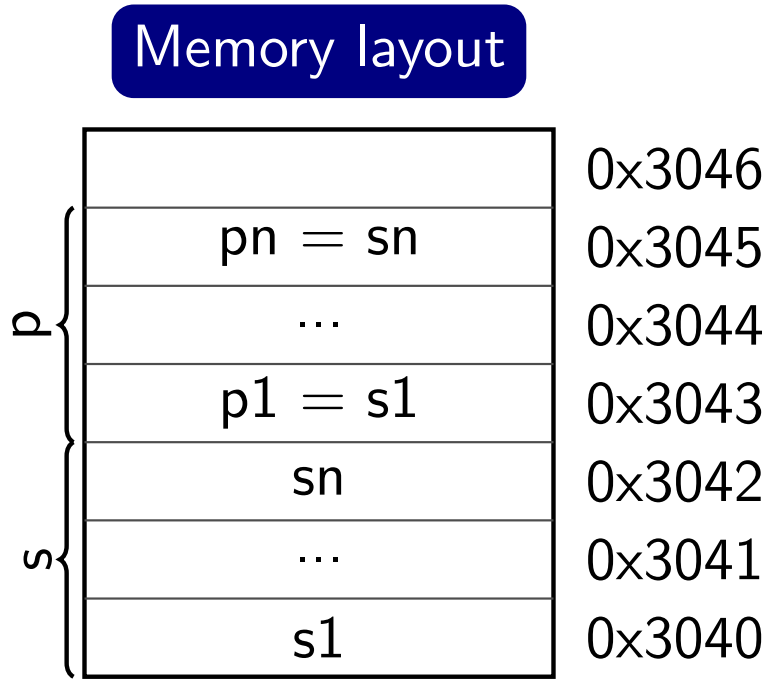
Parameter are passed by value

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```



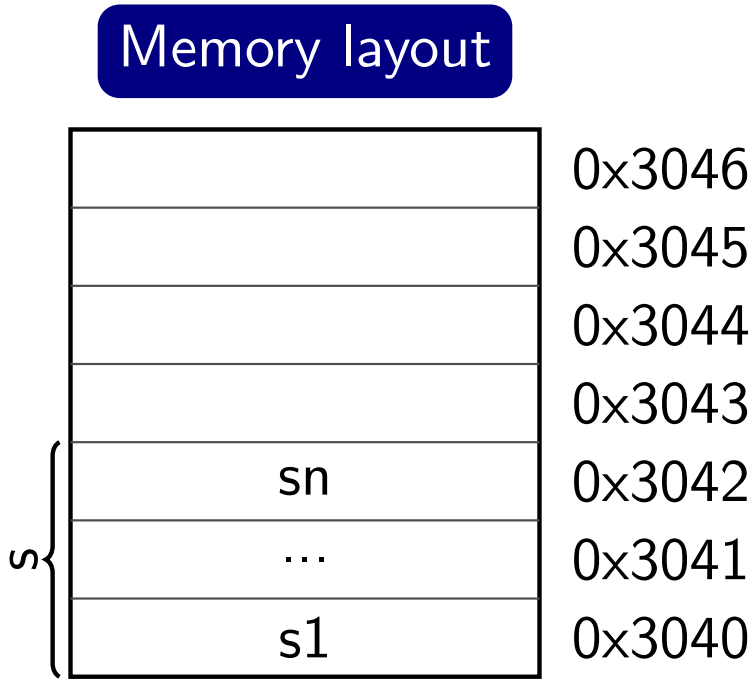
Parameter are passed by value

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```



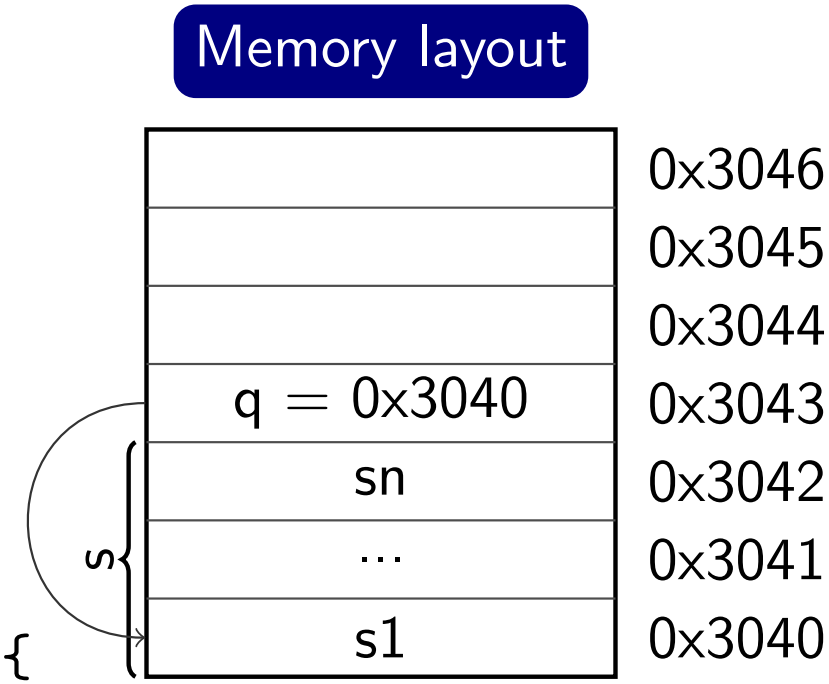
Parameter are passed by value

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```



Parameter are passed by value

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printBS(BigStruct p) {
6     ...
7  }
8  printBS(s); // replication
9
10 // parameter by pointer
11 void printBSp(BigStruct *q) {
12    ...
13 }
14 printBSp(&s); // not replication
```



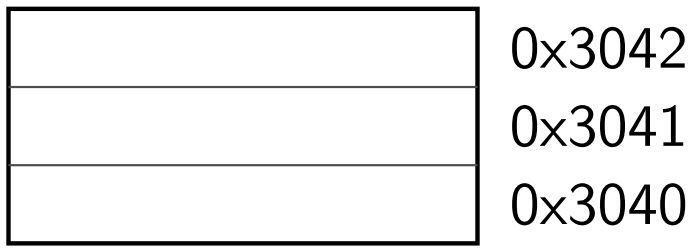
Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

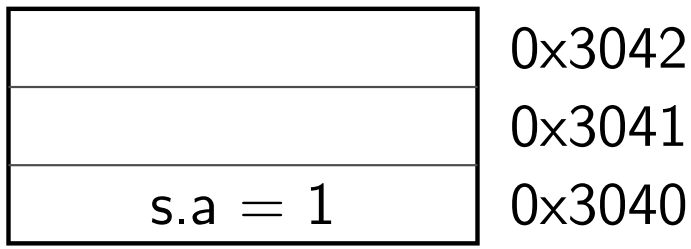
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

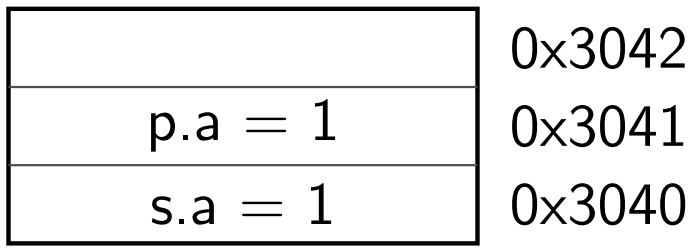
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

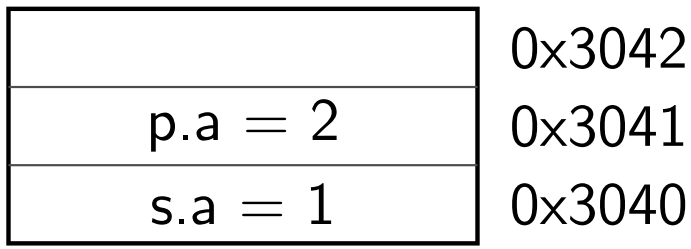
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

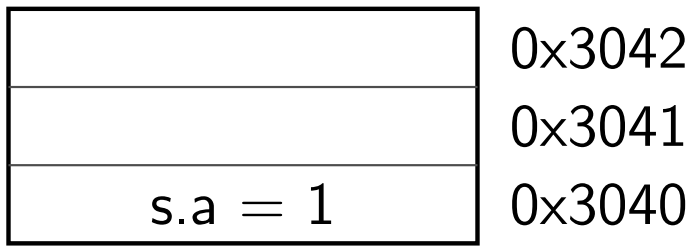
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

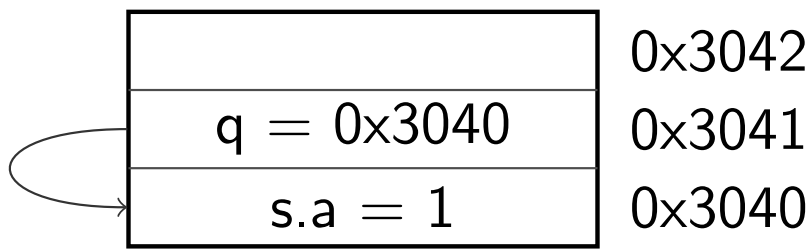
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

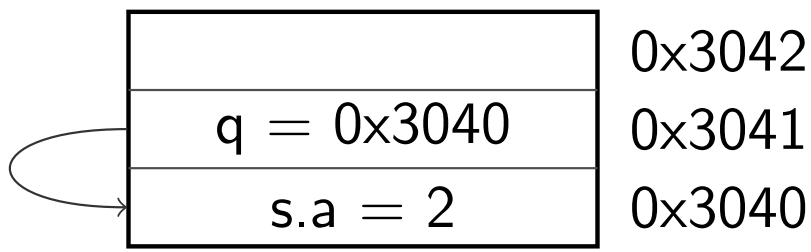
Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {.a = 1};
3
4  void changeSS(SmallStruct p) {
5    p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11   q->a = 2;  // i.e. (*q).a
12 }
13 changeSS2(s);
14 // s.a = 2
```

Memory layout



Control instructions

2. Langage basics (C and C++)
   - Core syntax and types
   - Arrays and Pointers
   - Operators
   - Compound data types
   - Functions
   - Control instructions
   - Headers and interfaces

Control instructions : if

if syntax

```
if (condition1) {
  Instructions1;
} else if (condition2) {
  Instructions2;
} else {
  Instructions3;
}
```

- *else* and *else if* part are optional
- *else if* part can be repeated
- braces are optional if there is a single instruction

Control instructions : if

Practical example

```
int collatz(int a) {
  if (a <= 0) {
    std::cout << "not supported";
    return 0;
  } else if (a == 1) {
    return 1;
  } else if (a%2 == 0) {
    return collatz(a/2);
  } else {
    return collatz(3*a+1);
  }
}
```

Control instructions : conditional operator

Syntax

```
test ? expression1 : expression2;
```

- if test is *true* expression1 is returned
- else expression 2 is returned

Control instructions : conditional operator

Syntax

```
test ? expression1 : expression2;
```

- if test is *true* expression1 is returned
- else expression 2 is returned

Practical example

```
int collatz(int a) {
  return a==1 ? 1 : collatz(a%2 ? 3*a+1 : a/2);
}
```

Control instructions : conditional operator

Syntax

```
test ? expression1 : expression2;
```

- if test is *true* expression1 is returned
- else expression 2 is returned

Practical example

```
int collatz(int a) {
  return a==1 ? 1 : collatz(a%2 ? 3*a+1 : a/2);
}
```

Do not abuse

explicit ifs are easier to read

to be used only when obvious and not nested

Control instructions : switch

Syntax

```
switch(identifier) {
  case c1 : instructions1; break;
  case c2 : instructions2; break;
  case c3 : instructions3; break;
  ...
  default : instructiond; break;
}
```

- *break* is not mandatory but...
- cases are entry points, not independant pieces
- execution carries on with the next case if no *break* is present !
- *default* may be omitted

Control instructions : switch

Syntax

```
switch(identifier) {
  case c1 : instructions1; break;
  case c2 : instructions2; break;
  case c3 : instructions3; break;
  ...
  default : instructiond; break;
}
```

- *break* is not mandatory but...
- cases are entry points, not independant pieces
- execution carries on with the next case if no *break* is present !
- *default* may be omitted

Use break

Do not try to make use of non breaking cases

Control instructions : switch

Practical example

```
enum Lang { FRENCH, GERMAN, ENGLISH, OTHER };
...
switch (language) {
case FRENCH:
  printf("Bonjour");
  break;
case GERMAN:
  printf("Guten tag");
  break;
case ENGLISH:
  printf("Good morning");
  break;
default:
  printf("I do not talk your langage");
}
```

Control instructions : for loop

for loop syntax

```
for(initializations; condition; increments) {
  instructions;
}
```

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Control instructions : for loop

for loop syntax

```
for(initializations; condition; increments) {
  instructions;
}
```

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Practical example

```
for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
  std::cout << i << "^2 is " << j << "\n";
}
```

Control instructions : for loop

for loop syntax

```
for(initializations; condition; increments) {
  instructions;
}
```

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Practical example

```
for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
  std::cout << i << "^2 is " << j << "\n";
}
```

Do not abuse the syntax

The for statement should fit in 1-3 lines

Control instructions : while loop

while loop syntax

```
while(condition) {
  instructions;
}
do {
  Instructions;
} while(condition);
```

- braces are optional if there is a single instruction

Control instructions : while loop

while loop syntax

```
while(condition) {
  instructions;
}
do {
  Instructions;
} while(condition);
```

- braces are optional if there is a single instruction

Practical example

```
while (n != 1)
  if (0 == n%2) n /= 2;
  else n = 3 * n + 1;
```

Control instructions : commands

control commands

break	goes out of the loop
continue	goes immediately to next iteration
return	goes out of current function

Control instructions : commands

control commands

break	goes out of the loop
continue	goes immediately to next iteration
return	goes out of current function

Practical example

```
while (1) {
  if (n == 1) break;
  if (0 == n%2) {
    std::cout << n << "\n";
    n /= 2;
    continue;
  }
  n = 3 * n + 1;
}
```

Headers and interfaces

Headers and interfaces

Interface

Set of declarations defining some functionnality

- defined in a "header file"
- no implementation defined

Header : Hello.hpp

```
void printHello();
```

Usage : myfile.cpp

```
#include "hello.hpp"
int main() {
  printHello();
}
```

Preprocessor

```
1  // file inclusion
2  #include "hello.hpp"
3  // macros
4  #define MY_GOLDEN_NUMBER 1746
5  // compile time decision
6  #ifdef USE64BITS
7    typedef uint64_t myint;
8  #else
9    typedef uint32_t myint;
10 #endif
```

Preprocessor

```
1  // file inclusion
2  #include "hello.hpp"
3  // macros
4  #define MY_GOLDEN_NUMBER 1746
5  // compile time decision
6  #ifdef USE64BITS
7    typedef uint64_t myint;
8  #else
9    typedef uint32_t myint;
10 #endif
```

Use only in very restricted cases

- include of headers
- hardcoded constants (should this happen at all ?)
- portability necessity

Object orientation

Objects and Classes

What are classes and objects

Classes

structs on steroids

- with inheritance
- including methods

Objects

instances of classes

A class encapsulates a concept

- shows an interface
- provides its implementation
  - status, properties
  - possible interactions
  - construction and destruction

My First Class

```
1  struct MyFirstClass {
2    int a;
3    void squareA() {
4      a *= a;
5    };
6    int sum(int b) {
7      return a + b;
8    };
9  };
10
11 MyFirstClass myObj;
12 myObj.a = 2;
13
14 // let's square a
15 myObj.squareA();
```

MyFirstClass
int a;
void squareA();
int sum(int b);

Separating the interface

Header : MyFirstClass.hpp

```
struct MyFirstClass {
  int a;
  void squareA();
  int sum(int b);
};
```

Implementation : MyFirstClass.cpp

```
#include "MyFirstClass.hpp"
void MyFirstClass::squareA() {
  a *= a;
};
void MyFirstClass::sum(int b) {
  return a + b;
};
```

A word on namespaces

- Namespaces allow to segment your code to avoid name clashes
- They can be embedded to create hierarchies (separator is '::')

```
1  int a;
2  namespace n {
3    int a;   // no clash
4  }
5  namespace p {
6    int a;   // no clash
7    namespace inner {
8      int a; // no clash
9    }
10 }
11 int f() {
12   n::a = 2;
13 }
```

```
14 namespace p {
15   int f() {
16     p::a = 2;
17     a = 2;  //same as above
18     p::inner::a = 4;
19     inner::a = 4;
20     n::a = 5;
21   }
22 }
23 using namespace p::inner;
24 int g() {
25   a = 3; // using p::inner
26 }
```

Implementing methods

Standard practice

- usually in .cpp, outside of class declaration
- using the class name as namespace
- when reference to the object is needed, use *this* keyword

```
1  void MyFirstClass::squareA() {
2    a *= a;
3  };
4
5  int MyFirstClass::sum(int b) {
6    int a = 0; // do not do that !
7    a += this->a;
8    a += b;
9    return a;
10 };
```

Method overloading

The rules in C++

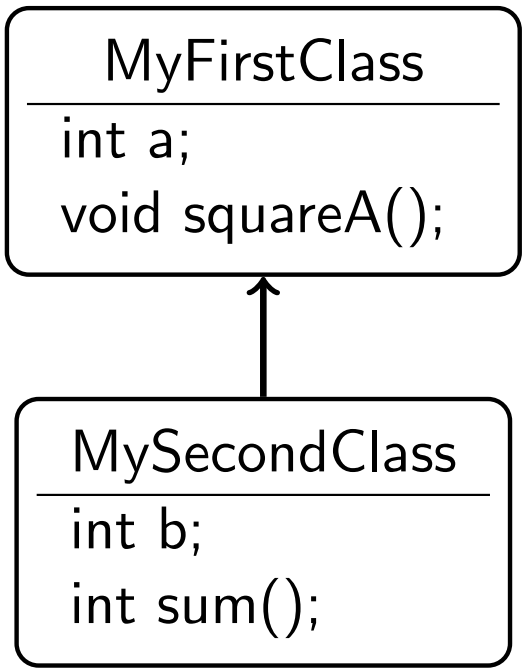
- overloading is authorized and welcome
- signature is part of the method identity
- but not the return code

```
1  struct MyFirstClass {
2    int a;
3    int sum(int b);
4    int sum(int b, int c);
5  }
6
7  int MyFirstClass::sum(int b) { return a + b; };
8
9  int MyFirstClass::sum(int b, int c) {
10   return a + b + c;
11 };
```

Inheritance

First inheritance

```
1  struct MyFirstClass {
2    int a;
3    void squareA() { a *= a; };
4  };
5  struct MySecondClass :
6    MyFirstClass {
7    int b;
8    int sum() { return a + b; };
9  };
10
11 MySecondClass myObj2;
12 myObj2.a = 2;
13 myObj2.b = 5;
14 myObj2.squareA();
15 int i = myObj2.sum(); // i = 9
```



Managing access to class members

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public

Managing access to class members

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public

```
1  class MyFirstClass {
2  public:
3    void setA(int a);
4    int getA();
5    void squareA();
6  private:
7    int a;
8  }
```

```
9  MyFirstClass obj;
10 obj.a = 5;    // error !
11 obj.setA(5); // ok
12 obj.squareA();
13 int b = obj.getA();
```

Managing access to class members

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public

```
1  class MyFirstClass {
2  public:
3    void setA(int a);
4    int getA();
5    void squareA();
6  private:
7    int a;
8  }
```

```
9  MyFirstClass obj;
10 obj.a = 5;    // error !
11 obj.setA(5); // ok
12 obj.squareA();
13 int b = obj.getA();
```

This breaks MySecondClass !

Managing access to class members(2)

Solution is *protected* keyword

Gives access to classes inheriting from base class

```
1  class MyFirstClass {
2  public:
3    void setA(int a);
4    int getA();
5    void squareA();
6  protected:
7    int a;
8  }
```

```
13 class MySecondClass :
14   public MyFirstClass {
15 public:
16   int sum() {
17     return a + b;
18   };
19 private:
20   int b;
21 }
```

Managing inheritance privacy

Inheritance can be public, protected or private

It influences the privacy of inherited members for external code.
The code of the class itself is not affected

public	privacy of inherited members remains unchanged
protected	inherited public members are seen as protected
private	all inherited members are seen as private this is the default if nothing is specified

Managing inheritance privacy

Inheritance can be public, protected or private

It influences the privacy of inherited members for external code.
The code of the class itself is not affected

public	privacy of inherited members remains unchanged
protected	inherited public members are seen as protected
private	all inherited members are seen as private this is the default if nothing is specified

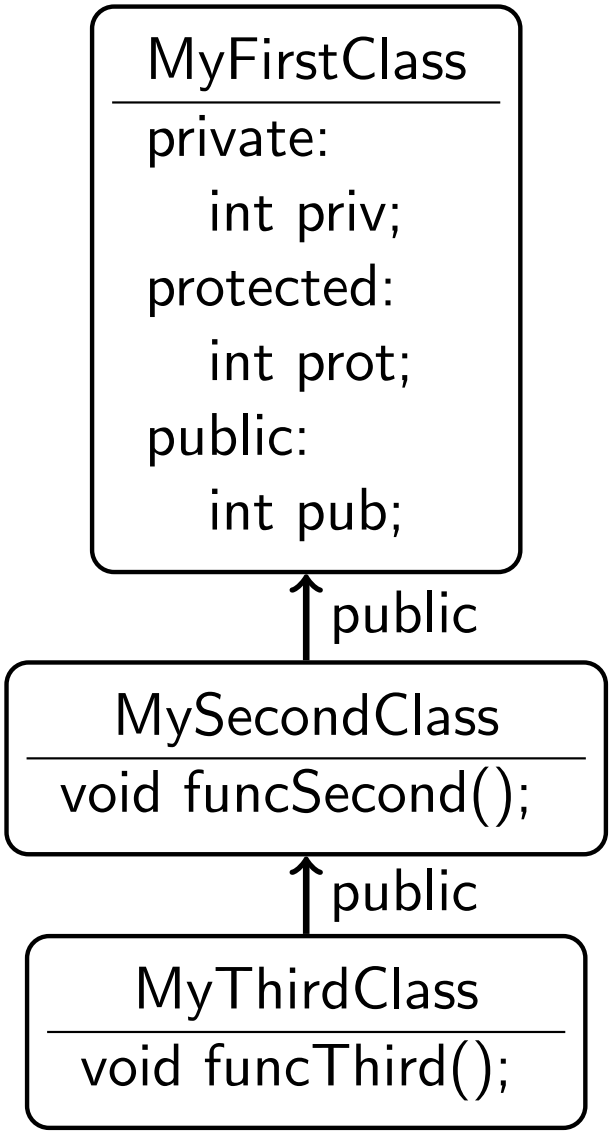
Net result for external code

- only public members of public inheritance are accessible

Net result for grand child code

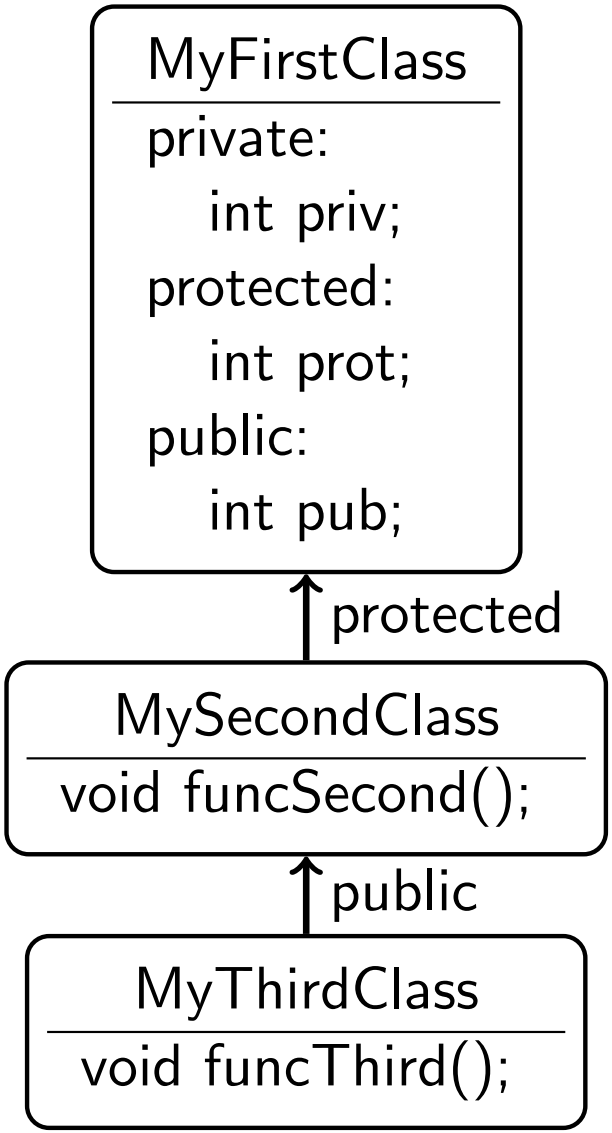
- only public and protected members of public and protected parents are accessible

Managing inheritance privacy - public



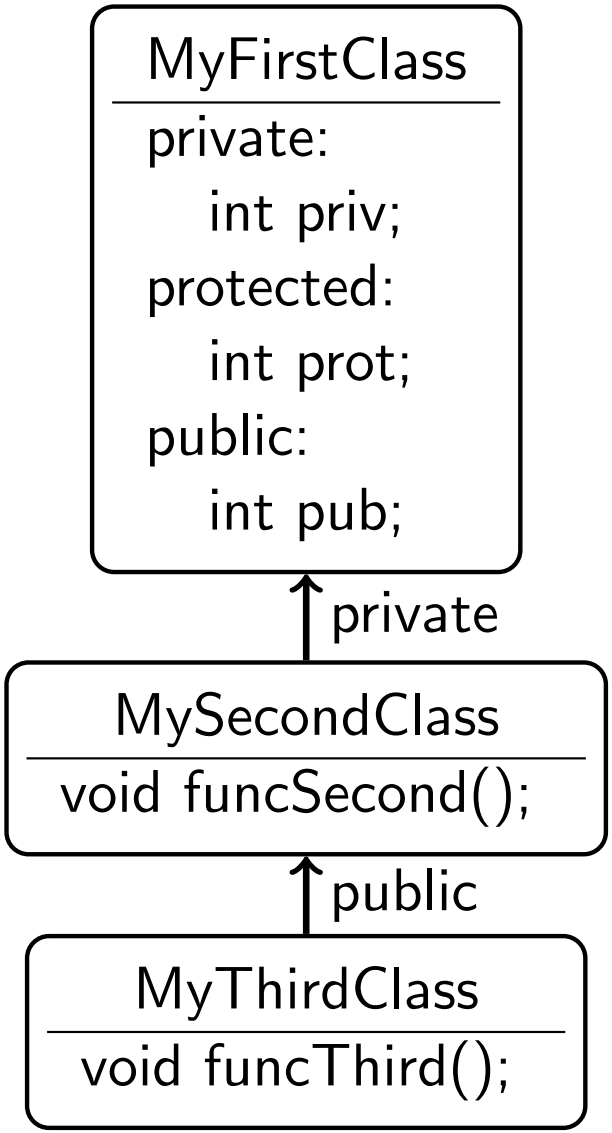

```
1  void funcSecond() {
2    int a = priv;    // Error
3    int b = prot;    // OK
4    int c = pub;     // OK
5  }
6  void funcThird() {
7    int a = priv;    // Error
8    int b = prot;    // OK
9    int c = pub;     // OK
10 }
11 void extFunc(MyThirdClass t) {
12   int a = t.priv; // Error
13   int b = t.prot; // Error
14   int c = t.pub;  // OK
15 }
```

Managing inheritance privacy - protected




```
1  void funcSecond() {
2    int a = priv;    // Error
3    int b = prot;    // OK
4    int c = pub;     // OK
5  }
6  void funcThird() {
7    int a = priv;    // Error
8    int b = prot;    // OK
9    int c = pub;     // OK
10 }
11 void extFunc(MyThirdClass t) {
12   int a = t.priv; // Error
13   int b = t.prot; // Error
14   int c = t.pub;  // Error
15 }
```

Managing inheritance privacy - private




```
1  void funcSecond() {
2    int a = priv;    // Error
3    int b = prot;    // OK
4    int c = pub;     // OK
5  }
6  void funcThird() {
7    int a = priv;    // Error
8    int b = prot;    // Error
9    int c = pub;     // Error
10 }
11 void extFunc(MyThirdClass t) {
12   int a = t.priv; // Error
13   int b = t.prot; // Error
14   int c = t.pub;  // Error
15 }
```

Constructors/destructors

Class Constructors and Destructors

Concept

- special functions building/destroying an object
- a class can have several constructors
- the constructors have the name of the class
- same for the destructor with a leading ~

```
1  class MyFirstClass {
2  public:
3    MyFirstClass();
4    MyFirstClass(int a);
5    ~MyFirstClass();
6    ...
7  protected:
8    int a;
9  };
```

```
10 // note special notation for
11 // initialization of members
12 MyFirstClass() : a(0) {}
13
14 MyFirstClass(int a_):a(a_) {}
15
16 ~MyFirstClass(){};
```

Class Constructors and Destructors

```
1  class Vector {
2  public:
3    Vector(int n);
4    ~Vector();
5    void setN(int n, int value);
6    int getN(int n);
7  private:
8    int len;
9    int* data;
10 }
11 Vector::Vector(int n) : len(n) {
12   data = (int*)malloc(n*sizeof(int));
13 }
14 Vector::~Vector() {
15   free(data);
16 }
```

Constructor and inheritance

```
1  struct MyFirstClass {
2    MyFirstClass();
3    MyFirstClass(int a);
4  }
5
6  struct MySecondClass : MyFirstClass {
7    MySecondClass();
8    MySecondClass(int b);
9    MySecondClass(int a, int b);
10 }
11
12 MySecondClass() : MyFirstClass(), b(0) {};
13 MySecondClass(int b_) : MyFirstClass(), b(b_) {};
14 MySecondClass(int a_,
15                int b_) : MyFirstClass(a_), b(b_) {};
```

Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type const ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, declare a private copy constructor without implementing it

Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type const ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, declare a private copy constructor without implementing it

```
1 struct MySecondClass : MyFirstClass {
2   MySecondClass();
3   MySecondClass(const MySecondClass &other);
4 }
```

Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type const ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, declare a private copy constructor without implementing it

```
1 struct MySecondClass : MyFirstClass {
2   MySecondClass();
3   MySecondClass(const MySecondClass &other);
4 }
```

The rule of 3

- if a class defines a destructor, a copy constructor or an assignment operator (see later), it should define all three

Class Constructors and Destructors

```
1  class Vector {
2  public:
3    Vector(int n);
4    Vector(const Vector &other);
5    ~Vector();
6    ...
7  }
8  Vector::Vector(int n) : len(n) {
9    data = (int*)calloc(n, sizeof(int));
10 }
11 Vector::Vector(const Vector &other) : len(other.len) {
12   data = (int*)malloc(len*sizeof(int));
13   memcpy(data, other.data, len);
14 }
15 Vector::~Vector() { free(data); }
```

Static members

Static members

Concept

- members attached to a class rather than to an object
- usable with or without an instance of the class
- identified by the *static* keyword

```
1  class Text {
2  public:
3    static std::string upper(std::string) {...};
4  private:
5    static int s_nbCallsToUpper;
6  };
7  int Text::s_nbCallsToUpper = 0;
8  std::string s = "my text";
9  std::string uppers = Text::upper("my text");
10 // now Text::s_nbCallsToUpper is 1
```

Allocating objects

Process memory organization

4 main areas

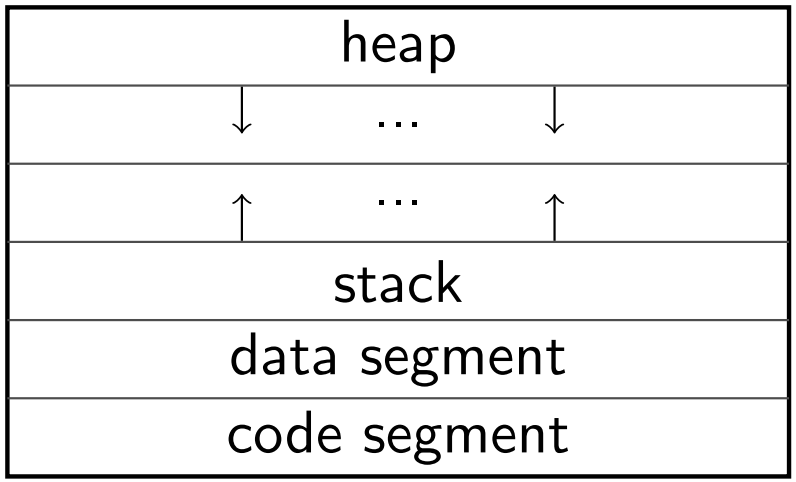
the code segment for the code of the executable

the data segment for global variables

the heap for dynamically allocated variables

the stack for parameters of functions and local variables

Memory layout



The Stack

Main characteristics

- allocation on the stack stays valid for the duration of the current scope. It is destroyed when it is popped off the stack.
- memory allocated on the stack is known at compile time and can thus be accessed through a variable.
- the stack is relatively small, it is not a good idea to allocate large arrays, structures or classes

Object allocation on the stack

On the stack

- object are created when declared (constructor called)
- object are destructed when out of scope (destructor is called)

```
1  int f() {
2    MyFirstClass a(3); // constructor called
3    ...
4  } // destructor called
5
6  {
7    MyFirstClass a; // default constructor called
8    ...
9  }  // destructor called
```

The Heap

Main characteristics

- Allocated memory stays allocated until it is specifically deallocated
  - beware memory leaks
- Dynamically allocated memory must be accessed through pointers
- large arrays, structures, or classes should be allocated here

Object allocation on the heap

On the heap

- object are created by calling *new* (constructor is called)
- object are destructed by calling *delete* (destructor is called)

```
1  {
2    // default constructor called
3    MyFirstClass *a = new MyFirstClass;
4    ...
5    delete a; // destructor is called
6  }
7
8  int f() {
9    // constructor called
10   MyFirstClass *a = new MyFirstClass(3);
11   ...
12 } // memory leak !!!
```

Array allocation on the heap

Arrays on the heap

- arrays of objects are created by calling *new[]*
  default constructor is called for each object of the array
- arrays of object are destructed by calling *delete[]*
  destructor is called for each object of the array

```
1  {
2    // default constructor called 10 times
3    MyFirstClass *a = new MyFirstClass[10];
4    ...
5    delete[] a; // destructor called 10 times
6  }
```

Exceptions

Exceptions

The concept

- exceptional Event breaking linearity of the code
- will be handled in dedicated place

Pratically

- you can throw any object with *throw*
- you handle them using *try ... catch* blocks

```
1  try {
2    if (0 == name) {
3      throw std::string("Expected non empty name");
4    }
5    printf("%s\n", name);
6  } catch (std::string e) {
7    printf("empty name found\n");
8  }
```

Exceptions

Rules

- exception will skip all code until next *catch*
  - still destructors are called when exiting scopes
  - but your own cleanup may not be
- *catch* is selective on the exception type

```
1  class ZeroDivide {};
2
3  int divide(int a, int b) {
4    if (0 == b) {
5      throw ZeroDivide();
6    }
7    return a/b;
8  }
```

```
9  int func(char* value) {
10   try {
11     errno = 0;
12     long l = strtol(value,0,10);
13     if (errno) {
14       throw string("Bad Value");
15     }
16     divide(100, l);
17   } catch (string e) {
18     printf("%s\n", e.c_str());
19   } catch (ZeroDivide e2) {
20     printf("Division error\n");
21   }
22 }
```

Controlling exceptions

Deprecated

Declaring expected exceptions

- each function can declare a set of expected exceptions
- using the *throw* statement in its declaration
- other exceptions won't exit the scope of the function
  - instead, the *unexpected* handler is called
  - by default, it terminates the program

Controlling exceptions

Deprecated

Declaring expected exceptions

- each function can declare a set of expected exceptions
- using the *throw* statement in its declaration
- other exceptions won't exit the scope of the function
  - instead, the *unexpected* handler is called
  - by default, it terminates the program

```
1  int func(int a) throw(int) {
2    if (0 == a) {
3      throw 2;  // ok, goes out
4    } else {
5      throw "hello"; // std::unexpected called
6    }
7  }
```

Controlling exceptions

Deprecated

Good to know

- The check in done at runtime, not at compile time
  - unlike Java
- When the *throw* clause is absent, any exception can go out
- To block all exceptions, use *throw()*

Controlling exceptions

Deprecated

Good to know

- The check in done at runtime, not at compile time
  - unlike Java
- When the *throw* clause is absent, any exception can go out
- To block all exceptions, use *throw()*

```
1  int func(int a) {
2    // any exception can go out
3  }
4  int otherfunc(int a) throw() {
5    // no exception can go out
6  }
```

Advanced Topics

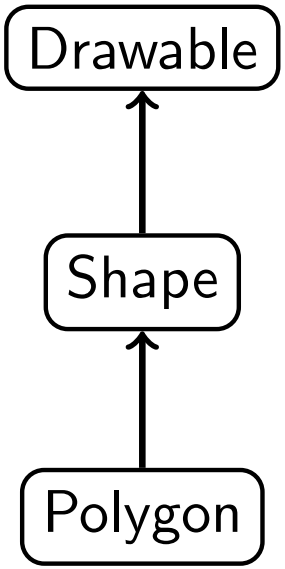
Object orientation

Polymorphism

the concept

- objects actually have multiple types concurrently
- and can be used as any of them

```
1  Polygon *p = new Polygon();
2
3  int f(Drawable *d) {...};
4  f(p);  //ok
5
6  try {
7    throw *p;
8  } catch (Shape e) {
9    // will be caught
10 }
```

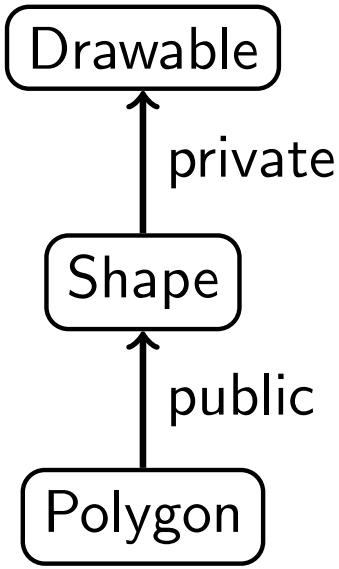


Inheritance privacy and polymorphism

Only public inheritance is visible to code outside the class

- private and protected are not
- this may restrict usage of polymorphism

```
1  Polygon *p = new Polygon();
2
3  int f(Drawable *d) {...};
4  f(p);  // Not ok anymore
5
6  try {
7    throw *p;
8  } catch (Shape e) {
9    // ok, will be caught
10 }
```

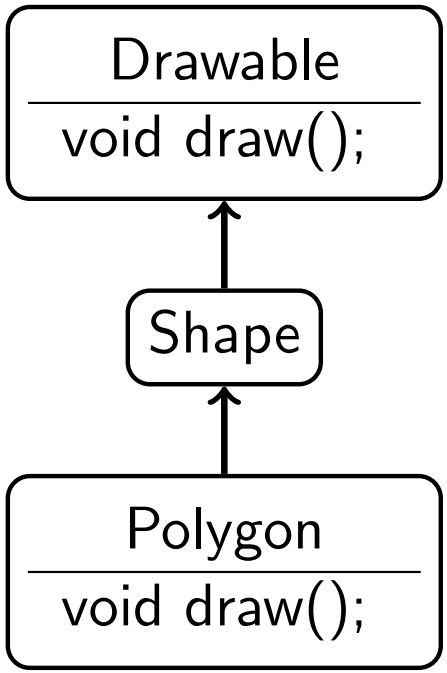


Method overriding

the problem

- a given method of the parent can be overriden in a child
- but which one is called ?

```
1  Polygon *p = new Polygon();
2  p->draw(); // ?
3
4  Shape* s = p;
5  s->draw(); // ?
```



Virtual methods

the concept

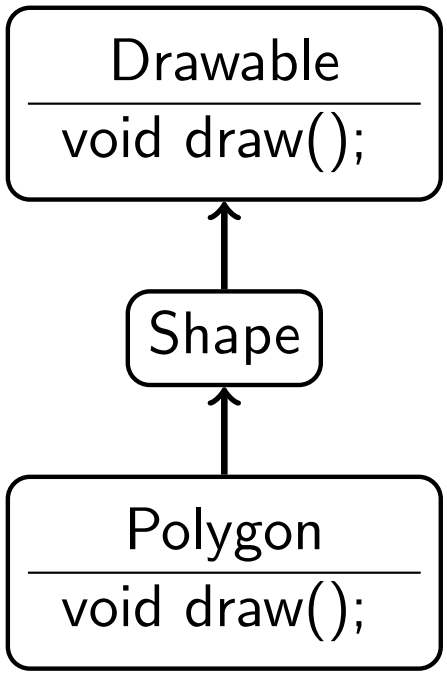
- methods can be declared *virtual*
- for these, the most precise object is always considered
- for others, the type of the variable decides

Virtual methods

the concept

- methods can be declared *virtual*
- for these, the most precise object is always considered
- for others, the type of the variable decides

```
1  Polygon *p = new Polygon();
2  p->draw(); // Polygon.draw
3
4  Shape* s = p;
5  s->draw(); // Drawable.draw
```

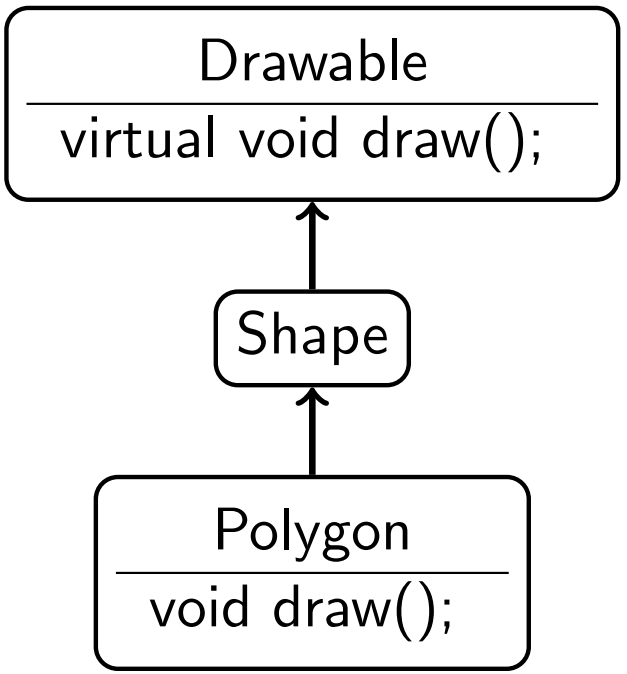


Virtual methods

the concept

- methods can be declared *virtual*
- for these, the most precise object is always considered
- for others, the type of the variable decides

```
1  Polygon *p = new Polygon();
2  p->draw(); // Polygon.draw
3
4  Shape* s = p;
5  s->draw();  // Polygon.draw
```



Pure Virtual methods

Concept

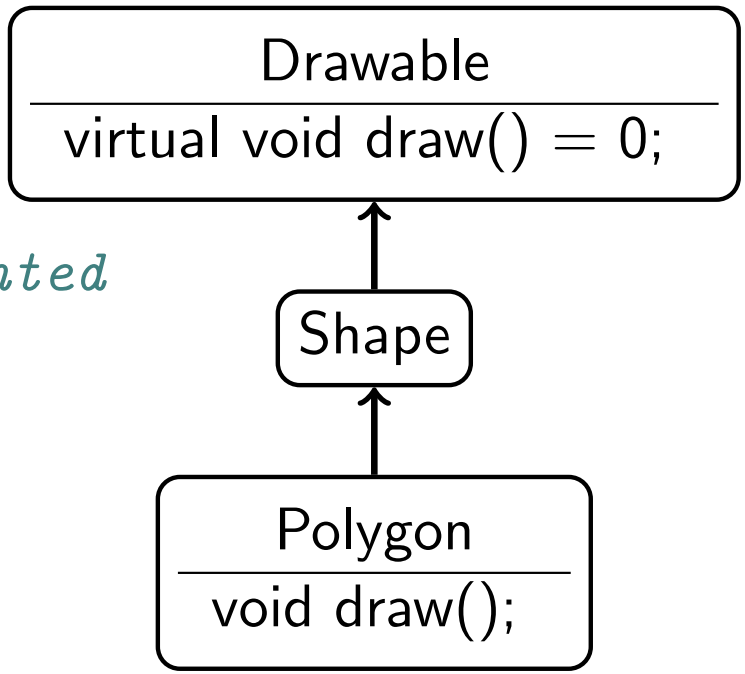
- methods that exist but are not implemented
- marked by an "$= 0$" in the declaration
- makes their class abstract
- an object can only be instantiated for a non abstract class

Pure Virtual methods

Concept

- methods that exist but are not implemented
- marked by an "$= 0$" in the declaration
- makes their class abstract
- an object can only be instantiated for a non abstract class

```
1  // Error : abstract class
2  Shape *s = new Shape();
3
4  // ok, draw has been implemented
5  Polygon *p = new Polygon();
6
7  // Shape type still usable
8  Shape* s = p;
9  s->draw();
```



Pure Abtract Class aka Interface

Definition of pure abstract class

- a class that has
  - no data member
  - all its methods pure virtual
- the equivalent of an Interface in Java

```
1  struct Drawable {
2    virtual void draw() = 0;
3  }
```

Drawable
virtual void draw() = 0;

Overriding overloaded methods

Concept

- overriding an overloaded method will hide the others
- unless you inherit them using *using*

```
1  struct BaseClass {
2    int foo(std::string);
3    int foo(int);
4  }
5  struct DerivedClass : BaseClass {
6    using BaseClass::foo;
7    int foo(std::string);
8  }
9  DerivedClass dc;
10 dc.foo(4);       // error if no using
```

Polymorphism

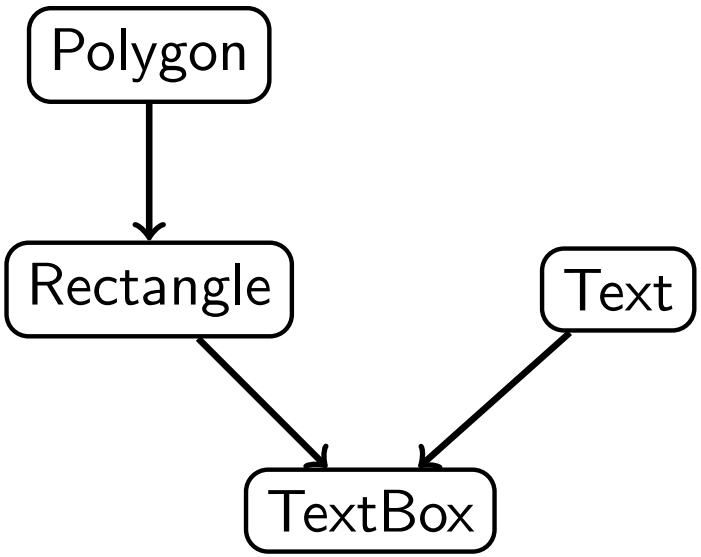
Exercise Time

- go to code/polymorphism
- look at the code
- open test.cpp
- create a Pentagon, call its perimeter method
- create an Hexagon, call its perimeter method
- create an Hexagon, call its parent's perimeter method
- retry with virtual methods

Multiple Inheritance

Concept

- one class can inherit from multiple parents




```
1  class TextBox :
2    public Rectangle, Text {
3    // inherits of both
4  }
```

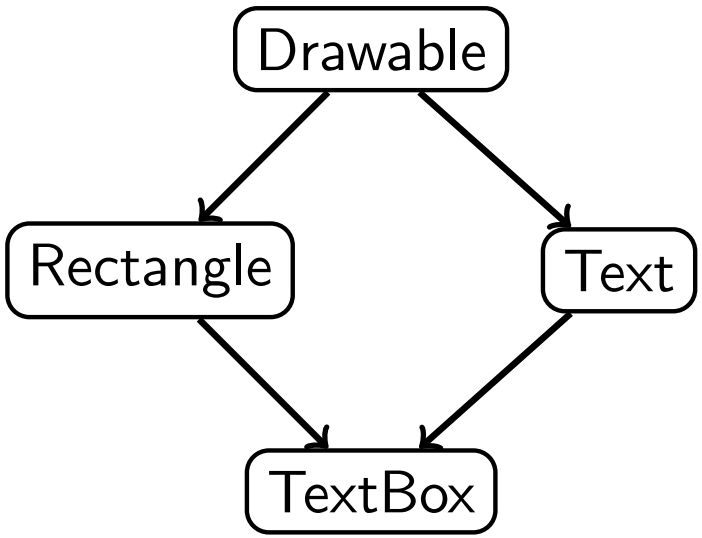
The diamond shape

Definition

- situation when one class inherits several times from a given grand parent

Problem

- are the members of the grand parent replicated ?

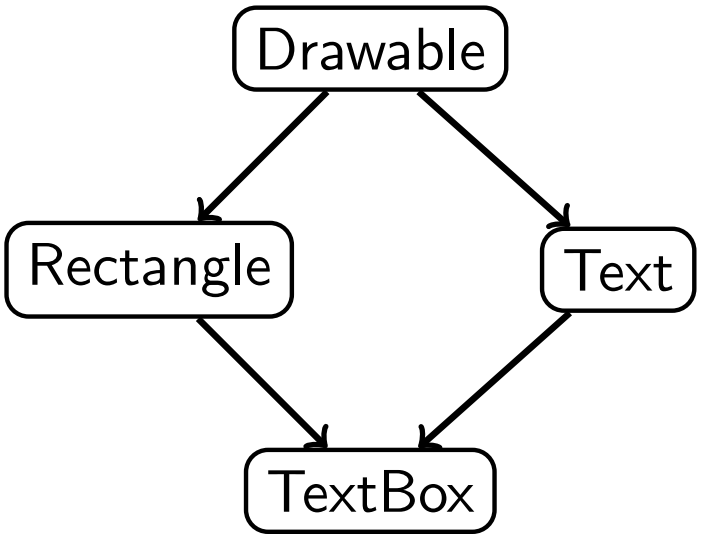


Virtual inheritance

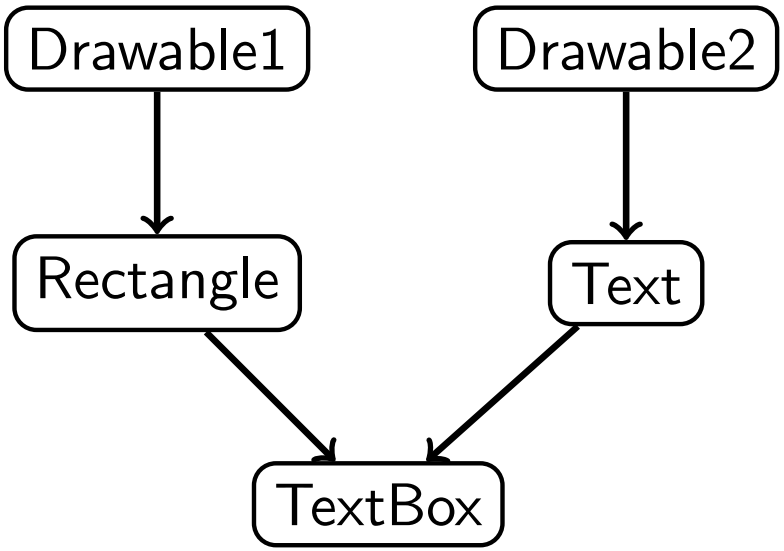
Solution

- inheritance can be *virtual* or not
- *virtual* inheritance will "share" parents
- standard inheritance will replicate them

virtual



standard



Multiple inheritance advice

Do not use multiple inheritance

- Except for inheriting from interfaces
- and for very seldom special cases

Multiple inheritance advice

Do not use multiple inheritance

- Except for inheriting from interfaces
- and for very seldom special cases

Do not use diamond shapes

- This is a sign that your architecture is not correct
- In case you are tempted, think twice and change mind

Virtual inheritance

Exercise Time

- go to code/virtual_inheritance
- look at the code
- open test.cpp
- create a TextBox and call draw
- Fix the code to call both draws by using types
- retry with virtual inheritance

Virtual inheritance

Warning

in case of virtual inheritance it is the most derived class that calls the virtual base class' constructor

Operators

Operators' example

```
1  struct Complex {
2    float m_real, m_imaginary;
3    Complex(float real, float imaginary);
4    Complex operator+(Complex& other) {
5      return Complex(m_real + other.m_real,
6                     m_imaginary + other.m_imaginary);
7    }
8  }
9
10 Complex c1(2, 3), c2 (4, 5);
11 Complex c3 = c1 + c2; // (6, 8)
```

Operators

Definition for operators of a class

- implemented as a regular method
  - either inside the class, as a member function
  - or outside the class (not all)
- with a special name (replace @ by anything)

Expression	As member	As non-member
@a	(a).operator@()	operator@(a)
a@b	(a).operator@(b)	operator@(a,b)
a=b	(a).operator=(b)	cannot be non-member
a(b...)	(a).operator()(b...)	cannot be non-member
a[b]	(a).operator[](b)	cannot be non-member
a->	(a).operator->()	cannot be non-member
a@	(a).operator@(0)	operator@(a,0)

Why to have non-member operators ?

Symetry

```
1  struct Complex {
2    float m_real, m_imaginary;
3    Complex operator+(float other) {
4      return Complex(m_real + other, m_imaginary);
5    }
6  }
7  Complex c1(2, 3);
8  Complex c2 = c1 + 4;  // ok
9  Complex c3 = 4 + c1;  // not ok !!
```

Why to have non-member operators ?

Symetry

```
1  struct Complex {
2    float m_real, m_imaginary;
3    Complex operator+(float other) {
4      return Complex(m_real + other, m_imaginary);
5    }
6  }
7  Complex c1(2, 3);
8  Complex c2 = c1 + 4;  // ok
9  Complex c3 = 4 + c1;  // not ok !!

10 Complex operator+(float a, const Complex& obj) {
11   return Complex(a + obj.m_real, obj.m_imaginary);
12 }
```

Other reason to have non-member operators ?

Extending existing classes

```
1  struct Complex {
2    float m_real, m_imaginary;
3    Complex(float real, float imaginary);
4  }
5
6  std::ostream& operator<<(std::ostream& os,
7                           const Complex& obj) {
8    os << "(" << obj.m_real << ", "
9              << obj.m_imaginary << ")";
10   return os;
11 }
12 Complex c1(2, 3);
13 std::cout << c1 << std::endl;
```

Value, pointers and references

4 Advanced Topics
- Object orientation
- Operators
- Value, pointers and references
- Constness
- Functors
- Templates
- The STL

Value, pointers and references

Different ways to pass arguments to a function

- by default arguments are passed by value
- but pointers can be used
- and references are also available

```
1 struct T {...};
2 void func    (T value);  // by value
3 void funcPtr (T *value); // pointer
4 void funcRef (T &value); // reference
```

Value versus pointers/reference

Identical to C

- by value, a copy is created
  - calling the copy constructor for objects
- using pointers, the memory address of value is passed
- using reference, a reference to value is passed

```
1  T a;        // constructor called
```

Value versus pointers/reference

Identical to C

- by value, a copy is created
  - calling the copy constructor for objects
- using pointers, the memory address of value is passed
- using reference, a reference to value is passed

```
1 T a;          // constructor called
2 funct(a); // copy constructor called on enter
3           // destructor called on exit
```

Value versus pointers/reference

Identical to C

- by value, a copy is created
  - calling the copy constructor for objects
- using pointers, the memory address of value is passed
- using reference, a reference to value is passed

```
1 T a;          // constructor called
2 funct(a); // copy constructor called on enter
3           // destructor called on exit
4 functPtr(&a); // no copy, but we pass a pointer
5 functRef(a);  // no copy, and standard syntax
```

Pointers vs References

Specificities of reference

- natural syntax
- will never be NULL
- cannot reference temporary object

Advantages of pointers

- can be NULL
- clearly indicates that argument may be modified

Pointers vs References

Specificities of reference

- natural syntax
- will never be NULL
- cannot reference temporary object

Advantages of pointers

- can be NULL
- clearly indicates that argument may be modified

Good practice

- Always use references when you can
- Consider that a reference will be modified
- Use const when it's not the case

Constness

Constness

The *const* keyword

- indicate that the element to the left is constant
- this element won't be modifiable in the future
- this is all checked at compile time

```
1  // standard syntax
2  int const i = 6;
3
4  // error : i is constant
5  i = 5;
6
7  // also ok, when nothing on the left,
8  // const applies to element on the right
9  const int j = 6;
```

Constness and pointers

```
1  // pointer to a constant integer
2  int a = 1, b = 2;
3  int const *i = &a;
4  *i = 5; // error, int is const
5  i = &b; // ok, pointer is not const
6
7  // constant pointer to an integer
8  int * const j = &a;
9  *j = 5; // ok, value can be changed
10 j = &b; // error, pointer is const
11
12 // constant pointer to a constant integer
13 int const * const k = &a;
14 *k = 5; // error, value is const
15 k = &b; // error, pointer is const
```

Function constness

The *const* keyword for class functions

- indicate that the function does not modify the object
- in other words, *this* is a pointer to constant object

```
1  struct Exemple {
2    void foo() const  {
3      m_member = 0; // Error : function is constant
4    }
5    int m_member;
6  };
```

Function constness

Constness is part of the type

- const T and T are different type
- however, T is automatically casted in const T when needed

```
1  void func(int *a);
2  void funcConst(const int *a);
3
4  int *a = 0;
5  const int *b = 0;
6
7  func(a);       // ok
8  func(b);       // error : no cannot cast int* to const
9  funcConst(a); // ok
10 funcConst(b); // ok
```

constness

Exercise Time

- go to code/constness
- open test.cpp
- try pointer to constant
- try constant pointer
- try constant pointer to constant
- try constant arguments of functions
- try constant method in a class

Functors

Functors

Concept

- a class that implements the () operator
- allows to use objects in place of functions
- and as objects have constructors, allow to construct functions

```
1  struct Adder {
2    int m_increment;
3    Adder(int increment) : m_increment(increment) {}
4    int operator()(int a) { return a + m_increment; }
5  };
6
7  Adder inc1(1), inc10(10);
8  int i = 3;
9  int j = inc1(i);  // 4
10 int k = inc10(i); // 13
```

Functors

Typical usage

- pass a function to another one
- or to an STL algorithm (see later)

```
1  struct BinaryFunction {
2    virtual double operator() (double a, double b) = 0;
3  };
4  struct Add : public BinaryFunction {
5    double operator() (double a, double b) { return a+b; }
6  };
7  double binary_op(double a, double b, BinaryFunction &func)
8    return func(a, b);
9  }
10 Add addfunc;
11 double c = binary_op(a, b, addfunc);
```

Templates

Templates

Concept

- The C++ way to write reusable code
  - aka macros on steroids
- Applicable to functions and objects

```
1  template<class T>
2  const T & Max(const T &A, const T &B) {
3    return A > B ? A : B;
4  }
5
6  template<class T>
7  struct Vector {
8    int m_len;
9    T* m_data;
10 }
```

Templates

Warning

These are really like macros

- they are compiled n times
- they need to be defined before used
  - so all templated code has to be in headers
- this may lead to longer compilation times and bigger libraries

```
template<class T>
T func(T a) {
  return a;
}
```

func(3)

```
int func(int a) {
  return a;
}
```

func(5.2)

```
double func(double a) {
  return a;
}
```

Templates

Arguments

- can be a class,
- you can have several
- last ones can have a default value

```
1  template<class KeyType=int, class ValueType=KeyType>
2  struct Map {
3    void set(KeyType &key, ValueType value);
4    ValueType get(KeyType &key);
5  }
6
7  Map<std::string, int> m1;
8  Map<float> m2;    // Map<float, float>
9  Map<> m3;         // Map<int, int>
```

Templates implementation

```
1  template<class KeyType=int, class ValueType=KeyType>
2  struct Map {
3    void set(KeyType &key, ValueType value);
4    ValueType get(KeyType &key);
5  }
6
7  template<class KeyType, class ValueType>
8  void Map<KeyType, ValueType>::set
9     (KeyType &key, ValueType value) {
10   ...
11 }
12
13 template<class KeyType, class ValueType>
14 ValueType Map<KeyType, ValueType>::get(KeyType &key) {
15   ...
16 }
```

Templates

Specialization

templates can be specialized for given values of their parameter

```
1  template<unsigned int N> struct Polygon {
2    Polygon(float radius);
3    float perimeter();
4    float m_radius;
5  };
6
7  template<>
8  struct Polygon<6> {
9    Polygon(float radius);
10   float perimeter() {return 6*m_radius;};
11   float m_radius;
12 };
```

The full power of templates

Exercise Time

- go to code/template
- look at the OrderedVector code
- compile and run test.cpp. See the ordering
- modify test.cpp and reuse OrderedVector with Complex
- improve OrderedVector to template the ordering
- test reverse ordering of strings (from the last letter)
- test manhattan order with complex type
- check the implementation of Complex
- try ordering complex of complex

The STL

The Standard Template Library

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient

The Standard Template Library

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient

Just use it

and adapt it to your needs, thanks to templates

STL in practice

```
1  #include <vector>
2  #include <algorithm>
3
4  std::vector<int> vi, vr(3);
5  vi.push_back(5); vi.push_back(3); vi.push_back(4);
6
7  std::transform(vi.begin(), vi.end(),      // range1
8                 vi.begin(),                // start range2
9                 vr.begin(),                // start result
10                std::multiplies<int>()); // function
11
12 for(std::vector<int>::iterator it = vr.begin();
13     it != vr.end();
14     it++) {
15   std::cout << *it << " ";
16 };
```

STL's concepts

containers

- a structure containing data
- with a given way of handling it
- irrespective of
  - the data itself (templated)
  - the memory allocation of the structure (templated)
  - the algorithms that may use the structure
- examples
  - string
  - list, vector, deque
  - map, set, multimap, multiset, hash_map, hash-set, ...
  - stack, queue, priority_queue

STL's concepts

iterators

- generalization of pointers
- allowing iteration over some data
- irrespective of
  - the container used (templated)
  - the data itself (container is templated)
  - the consumer of the data (templated algorithm)
- examples
  - iterator
  - reverse_iterator
  - const_iterator

STL's concepts

algorithms

- implementation of an algorithm working on data
- with a well defined behavior (defined complexity)
- irrespective of
  - the data handled
  - the container where data live
  - the iterator used to go through data
- examples
  - for_each, find, find_if, count, count_if, search
  - copy, swap, transform, replace, fill, generate
  - remove, remove_if
  - unique, reverse, rotate, random, partition
  - sort, partial_sort, merge, min, max
  - lexicographical_compare, iota, accumulate, partial_sum

STL's concepts

functions / functors

- generic utility functions/functors
- mostly useful to be passed to STL algorithms
- implemented independently of
  - the data handled (templated)
  - the context (algorithm) calling it
- examples
  - plus, minus, multiply, divide, modulus, negate
  - equal_to, less, greater, less_equal, ...
  - logical_and, logical_or, logical_not
  - identity, project1st, project2nd
  - binder1st, binder2nd, unary_compose, binary_compose

STL in practice

```
1  #include <vector>
2  #include <algorithm>
3
4  std::vector<int> vi, vr(3);
5  vi.push_back(5); vi.push_back(3); vi.push_back(4);
6
7  std::transform(vi.begin(), vi.end(),      // range1
8                 vi.begin(),                // start range2
9                 vr.begin(),                // start result
10                std::multiplies<int>()); // function
11
12 for(std::vector<int>::iterator it = vr.begin();
13     it != vr.end();
14     it++) {
15   std::cout << *it << " ";
16 };
```

More complex STL code

```
1  // Finds the first element in a list that lies in
2  // the range from 1 to 10.
3  list<int> L;
4  ...
5  list<int>::iterator in_range =
6    find_if(L.begin(), L.end(),
7            compose2(logical_and<bool>(),
8                     bind2nd(greater_equal<int>(), 1),
9                     bind2nd(less_equal<int>(), 10)));
10
11 // Computes sin(x)/(x + DBL_MIN) for elements of a range.
12 transform(first, last, first,
13           compose2(divides<double>(),
14                    ptr_fun(sin),
15                    bind2nd(plus<double>(), DBL_MIN)));
```

Welcome to lego programming !

Using the STL

Exercise Time

- go to code/stl
- look at the non STL code in test.nostl.cpp
  - it creates a vector of ints at regular intervals
  - it randomizes them
  - it computes differences between consecutive ints
  - and the mean and variance of it
- open test.cpp and complete the "translation" to STL
- see how easy it is to reuse the code with complex numbers

Using the STL

Some last warning

You may find the STL quite difficult to use.

- template syntax is simply awful
- it is hard to debug (compilers spit out mostly garbage)
- the standard is not well defined (SGI vs C++98 vs C++11)

However, this has improved a lot with C++11

Useful tools

C++editor

C++editor

Choose it wisely

- it can improve dramatically your efficiency by
  - coloring the code for you to "see" the structure
  - helping indenting properly
  - allowing you to navigate easily in the source tree
  - helping for compilation/debugging

A few tools

- Visual Studio: the Microsoft way
- Eclipse: similar, but open source and portable
- NetBeans: similar again, also portable
- Emacs: the expert way. Extremely powerful. Programmable
  It is to IDEs what latex is to PowerPoint

Choosing one is mostly a matter of taste

Code management tools

5 Useful tools

- C++editor
- Code management tools
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis

Code management tool

Please use one !

- even locally
- even on a single file
- even if you are the only commiter

It will soon save your day

A few tools

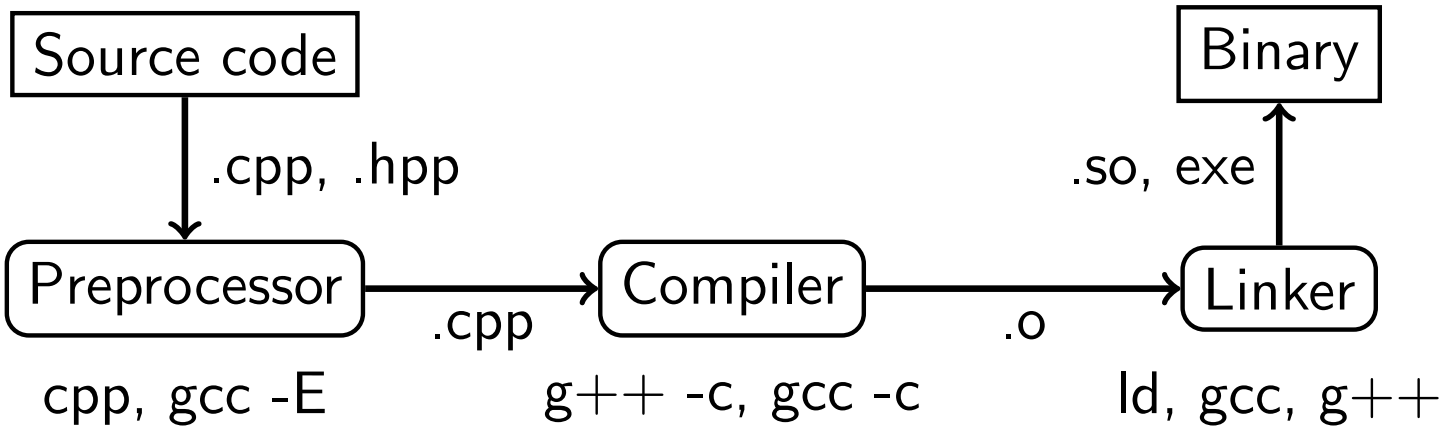
git	THE best choice. Fast, light, easy to use
mercurial	the main alternative
Bazzar	another alternative
svn	historical, not distributed - DO NOT USE
CVS	archeological, not distributed - DO NOT USE

GIT crash course

```
# mkdir myProject; cd myProject; git init
Initialized empty Git repository in /tmp/myProject/.git/

# vim file.cpp; vim file2.cpp
# git add file.cpp file2.cpp
# git commit -m "commiting first 2 files"
[master (root-commit) c481716] commiting first 2 files
...

# git log --oneline
d725f2e Better STL test
f24a6ce Reworked examples + added stl one
bb54d15 implemented template part
...

# git diff f24a6ce bb54d15
```

The Compiling Chain

The compiling chain



The steps

cpp the preprocessor
handles the # directives (macros, includes)
creates "complete" source code

g++ the compiler
creates assembly code from C++ code

ld the linker
links several binary files into libraries and executables

Compilers

Available tools

- gcc: the most common and most used
  free and open source
- clang: drop-in replacement of gcc
  better output and error reporting
  free and open source
- icc: the intel compiler
  proprietary
  optimized for Intel hardware
- Visual C++: the Windows way

My prefered choice today

- clang for its error reporting
- gcc is not far and tries to catch up

Useful compiler options (gcc/clang)

Get more warnings

-Wall -Wextra the way to get all warnings

-Werror the way to force yourself to look at warnings

Around optimization

-g add debug symbols

-O0, -O2 0 = no optimization, -O2 = optimized

Compilation environment

-I <path> where to find header files

-L <path> where to find libraries

-l <name> link with libname.so

-E / -c stop after preprocessing / compilation

Makefiles

Why to use them

- an organized way of describing building steps
- avoids a lot of typing

Several implementations

- raw Makefiles : suitable for small projects
- cmake : portable, the current best choice
- automake : portable but complex

```
test : test.cpp libpoly.so
    $(CXX) -Wall -Wextra -o $@ $^
libpoly.so: Polygons.cpp
    $(CXX) -Wall -Wextra -shared -fPIC -o $@ $^
clean:
    rm -f *o *so *~ test test.sol
```

Compiler chain

Exercise Time

- go to code/polymorphism
- preprocess Polygons.cpp (cpp or gcc -E -o output)
- compile Polygons.o and test.o (g++ -c -o output)
- use nm to check symbols
- see link statement using g++ -v
- see library dependencies with ldd
- look at the Makefile
- try make clean; make

Debugging

5 Useful tools
- C++editor
- Code management tools
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis

Debugging

The problem

- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue

Debugging

The problem

- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue

The solution : debuggers

- dedicated program able to stop execution at any time
- and show you where you are and what you have

Debugging

The problem

- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue

The solution : debuggers

- dedicated program able to stop execution at any time
- and show you where you are and what you have

Existing tools

- gdb THE main player
- lldb the debugger coming with clang
  still young, no stable version
- idb the intel debugger, proprietary

gdb crash course

start gdb

- gdb <program>
- gdb <program><core file>

inspect state

bt	prints a backtrace
print <var>	prints current content of the variable
list	show code around current point
up/down	go up or down in call stack

breakpoints

break <function>	puts a breakpoint on function entry
break <file>:<line>	puts a breakpoint on that line

gdb

Exercise Time

- go to code/debug
- compile, run, see the crash
- run it in gdb
- inspect backtrace, variables
- find problem and fix bug
- try stepping, breakpoints
- use -Wall -Wextra and see warning

The Valgrind family

The valgrind family

Valgrind fundamentals

- valgrind is a framework for different tools
- a processor simulator allowing checks in between instructions
- slow (10-50 times slower than normal execution)
- easy to use : "valgrind <your executable>"
  - no recompilation
  - better with -g -O0, but not strictly needed
- it is free and open source

The valgrind family

Valgrind fundamentals

- valgrind is a framework for different tools
- a processor simulator allowing checks in between instructions
- slow (10-50 times slower than normal execution)
- easy to use : "valgrind <your executable>"
  - no recompilation
  - better with -g -O0, but not strictly needed
- it is free and open source

Main tools

memcheck a memory checker (default tool) and leak detector

callgrind a call graph builder

helgrind a race condition detector

memcheck

- keeps track of all memory allocations and deallocations
- is able to detect accesses to non allocated memory
- and even tell you when it was deallocated if it was
- or what it the closest array in case of overflow
- is able to list still allocated memory when program exits (memory leaks detection)

valgrind

Exercise Time

- go to code/valgrind
- compile, run, it should work
- run with valgrind, see the problem
- fix the problem

- go back to the code/debug exercise
- check it with valgrind
- analyze the issue, see that the variance was biaised
- fix the issue

memcheck

Exercise Time

- go to code/memcheck
- compile, run, it should work
- run with valgrind, see LEAK summary
- run with --leak-check=full to get details
- analyze and correct it

callgrind and kcachegrind

callgrind

- keeps track of all function calls
- and time spent in each function
- build statistics on calls, CPU usages and more
- outputs flat statistics file, quite unreadable

kcachegrind

- a gui exploiting statistics built by callgrind
- able to browse graphically the program calls
- able to "graph" CPU usage on the program structure

callgrind

Exercise Time

- go to code/callgrind
- compile, run, it will be slow
- change nb iterations to 20
- run with valgrind --tool=callgrind
- look at output with kcachegrind
- change fibo call to fibo2
- observe the change in kcachegrind

helgrind

- keeps track of all pthreads activity
- in particular keeps track of all mutexes
- builds a graph of dependencies of the different actions
- works on the resulting graph to detect:
  - possible dead locks
  - possible data races

helgrind

- keeps track of all pthreads activity
- in particular keeps track of all mutexes
- builds a graph of dependencies of the different actions
- works on the resulting graph to detect:
  - possible dead locks
  - possible data races

Note the "possible". It finds future problems !

helgrind

Exercise Time

- go to code/helgrind
- compile, run
- check it with valgrind. See strange behavior but no explanation
- check it with valgrind --tool=helgrind
- understand issue and fix

Static code analysis

5 Useful tools
- C++ editor
- Code management tools
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

A (partial) solution : analyzing the source code

- build a graph of dependencies of the calls
- use graph tools to detect potential memory corruptions, memory leaks ot missing initializations

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

A (partial) solution : analyzing the source code

- build a graph of dependencies of the calls
- use graph tools to detect potential memory corruptions, memory leaks ot missing initializations

Existing tools

- Coverity — proprietary tool, the most complete
- cppcheck — free and opensource, but less complete
- scan-build — the clang source analyzer, still young

cppcheck

Exercise Time

- go to code/cppcheck
- compile, run, see that it works
- use valgrind : no issue
- use cppcheck, see the problem
- analyze the issue, and fix it
- bonus : understand why valgrind did not complain and how the standard deviation could be biased
  hint : use gdb and check addresses of v and diffs

C++11/14 features

Introduction

Introduction to C++11 and C++14

status

- Latest C++specifications
- Bringing a lot of goodies

Introduction to C++11 and C++14

status

- Latest C++specifications
- Bringing a lot of goodies

How to use C++11 features

- Use a compatible compiler gcc $\geq$ 4.8, clang $\geq$ 3.3
- add -std=c++11 to compilation flags

How to use C++14 features

- Use a compatible compiler gcc $\geq$ 4.9, clang $\geq$ 3.4
- add -std=c++14 to compilation flags

How to use C++17 features

- Use a compatible compiler gcc $\geq$ 7.3, clang $\geq$ 5
- add -std=c++17 to compilation flags

Constant Expressions

Generalized Constant Expressions

Reason of being

- compute constant expressions at compile time
- even if non trivial

Generalized Constant Expressions

Reason of being

- compute constant expressions at compile time
- even if non trivial

Example

```
constexpr int f(int x) {
  return x > 1 ? x * f(x - 1) : 1;
}
int a = f(5); // now computed at compile time
```

Generalized Constant Expressions(2)

Few limitations

- function's body cannot contain try-catch or static variables
- arguments should be constexpr or literals in order to benefit from compile time computation

Notes

- classes can have constexpr functions
- objects can be constexpr
  - if the constructor of their class is
- a constexpr function can also be used normally

Real life example

```
1  constexpr float toSI(const float v, const char unit) {
2    switch (unit) {
3    case 'k': return 1000*v;
4    case 'm': return 0.001*v;
5    case 'y': return 0.9144*v;
6    case 'i': return 0.0254*v;
7    ...
8    default: return v;
9    }
10 }
11 constexpr float fromSI(const float v, const char unit) {
12   switch (unit) {
13   case 'k': return 0.001*v;
14   ...
15   }
16 }
```

Real life example(2)

```
1  class DimLength {
2    const float m_value;
3  public:
4    constexpr DimLength(const float v, const char unit):
5      m_value(convertToSI(v, unit)) {
6    }
7    constexpr float get(const char unit) const {
8      return convertFromSI(m_value, unit);
9    }
10 };
11 constexpr DimLength km(1, 'k');
12 constexpr float km_y = km.get('y');
13 constexpr float km_i = km.get('i');
14 std::cout << "1 km = " << km_y << " yards\n"
15           << "     = " << km_i << " inches\n";
```

Range based loops

Range based loops

Reason of being

- simplifies loops tremendously
- especially with STL containers

Syntax

```
for ( type iteration_variable : container ) {
  // body using iteration_variable
}
```

Example code

```
std::vector<int> v;
int sum = 0;
for (int a : v) { sum += a; }
```

auto keyword

Auto keyword

Reason of being

- many type declarations are redundant
- and lead to compiler error if you mess up

```
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug ? unsigned to signed
```

Auto keyword

Reason of being

- many type declarations are redundant
- and lead to compiler error if you mess up

```
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug ? unsigned to signed
```

New way

```
std::vector<int> v;
auto a = v[3];
auto b = v.size();
```

Loops and auto keyword with the STL

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5    sum += *it;
6  }
```

Loops and auto keyword with the STL

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5    sum += *it;
6  }
```

New way

```
1  std::vector<int> v = ...;
2  int sum = 0;
3  for (auto a : v) { sum += a; }
```

Loops and auto keyword with the STL

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5    sum += *it;
6  }
```

New way

```
1  std::vector<int> v = ...;
2  int sum = 0;
3  for (auto a : v) { sum += a; }
```

STL way

```
1  std::vector<int> v = ...;
2  int sum = std::accumulate(v.begin(), v.end(), 0);
```

override and final

Override keyword

Idea

- make sure you do not accidentally create a new virtual function, when intending to override

```
1  struct Base {
2    virtual void some_func(float);
3  };
4  struct Derived : Base {
5    virtual void some_func(double); // oups !
6  };
```

Practically

```
1  struct Derived : Base {
2    virtual void some_func(double) override; // error
3  };
```

Final keyword

Idea

- make sure you cannot override a given method
- by declaring it final

Practically

```
1  struct Base {
2    virtual void some_func(float) final;
3  };
4  struct Derived : Base {
5    virtual void some_func(float) override; // error
6  };
```

non-member begin/end

non-member begin and end

The problem

STL containers and arrays have different syntax for loop

```
1  std::vector<int> v;
2  for(it = v.begin(); it != v.end(); it++) {...}
3  int a[] = {1,2,3};
4  for(i = 0; i < 3; i++) {...}
```

non-member begin and end

The problem

STL containers and arrays have different syntax for loop

```
1  std::vector<int> v;
2  for(it = v.begin(); it != v.end(); it++) {...}
3  int a[] = {1,2,3};
4  for(i = 0; i < 3; i++) {...}
```

The new syntax

```
1  std::vector<int> v;
2  for(auto it = begin(v); it != end(v); it++) {...}
3  int a[] = {1,2,3};
4  for(int* i = begin(a); i != end(a); i++) {...}
```

Initializers

Initializers - objects

C++98 nightmare

```
1  struct A {
2    int a;
3    float b;
4    A();
5    A(int);
6    A(int, int);
7  };
```

```
8  struct B {
9    int a;
10   float b;
11 };
```

Initializers - objects

C++98 nightmare

```
1  struct A {
2    int a;
3    float b;
4    A();
5    A(int);
6    A(int, int);
7  };

8  struct B {
9    int a;
10   float b;
11 };

12 A a(1,2);          // A::A(int, int)
13 A a(1);           // A::A(int)
14 A a();            // declaration of a function !
15 A a;              // A::A()
16 B b = {1, 2.3}; // OK
17 A a = {1,2};      // not allowed
```

Initializers - objects

C++11 uniformization

```
1  struct A {
2    int a;
3    float b;
4    A();
5    A(int);
6    A(int, int);
7  };

8  struct B {
9    int a;
10   float b;
11 };

12 A a{1,2};          // A::A(int, int)
13 A a{1};            // A::A(int)
14 A a{};             // A::A()
15 A a;               // A::A()
16 B b = {1, 2.3};    // OK
17 A a = {1,2};       // OK, A::A(int, int)
```

Initializers - arrays and vectors

C++98 nightmare

```
10 int ip[3] = {1,2,3};                // OK
11 int* ip = new int[3]{1,2,3};        // not allowed
12 std::vector<int> v = {1,2,3}; // not allowed
```

Initializers - arrays and vectors

C++98 nightmare

```
10  int ip[3] = {1,2,3};              // OK
11  int* ip = new int[3]{1,2,3};      // not allowed
12  std::vector<int> v = {1,2,3}; // not allowed
```

C++11 uniformization

```
10  int ip[3] = {1,2,3};              // OK
11  int* ip = new int[3]{1,2,3};      // OK
12  std::vector<int> v = {1,2,3}; // OK
```

Constructors

6 C++11/14 features

- Introduction
- Constant Expressions
- Range based loops
- auto keyword
- override and final
- non-member begin/end
- Initializers
- Constructors
- Exceptions
- Lambdas
- Move semantic
- pointers and RAII
- Threads and async
- Mutexes

Default Constructor

Idea

- avoid writing explicitely default constructors
- by declaring them as default

Details

- when no user defined constructor, a default is provided
- any user defined constructor disables default ones
- but they can be enforced.
- rule can be more subtle depending on members

Practically

```
1  ClassName() = default;   // provide/force default
2  ClassName() = delete;    // do not provide default
```

Constructor delegation

Idea

- avoid replication of code in several constructors
- by delegating to another constructor, in the initializer list

Practically

```
1  struct Delegate {
2    int m_i;
3    Delegate() { ... complex initialization ...};
4    Delegate(int i) : Delegate(), m_i(i) {};
5  }
```

Constructor inheritance

Idea

- avoid declaring empty constructors inheriting parent's ones
- by stating that we inherit all parent constructors

Practically

```
1  struct BaseClass {
2    BaseClass(int value);
3  };
4  struct DerivedClass : BaseClass {
5    using BaseClass::BaseClass;
6  };
7  DerivedClass a{5};
```

Member initialization

Idea

- avoid redefining same default value for members in all constructors
- by defining it once at member declaration time

Practically

```
1  struct BaseClass {
2    int a{5};
3    BaseClass() = default;
4  };
5  struct DerivedClass : BaseClass {
6    int b{6};
7    using BaseClass::BaseClass;
8  };
9  DerivedClass *a = new DerivedClass{};
```

Exceptions

6 C++11/14 features

- Introduction
- Constant Expressions
- Range based loops
- auto keyword
- override and final
- non-member begin/end
- Initializers
- Constructors
- Exceptions
- Lambdas
- Move semantic
- pointers and RAII
- Threads and async
- Mutexes

Deprecation of C++98 exceptions

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

Deprecation of C++98 exceptions

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

Some of the reasons

- throw specification is runtime only
  - does not allow compiler optimizations
  - on the contrary forces extra checks
  - generally terminates your program if violated
- throw specification clashes with templates
  - one cannot "template" the throw clause

What remains C++11/14

throw is dead

- throw statements are deprecated
- even throw() (no exceptions)

What remains C++11/14

throw is dead

- throw statements are deprecated
- even throw() (no exceptions)

long live noexcept

- noexcept a somehow equivalent to throw()
- but is checked at compile time
- so allows compiler optimizations

Full power of noexcept

3 uses of noexcept

- standalone

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

- as an expression saying whether exceptions can be sent

```
int f() noexcept(sizeof(long) == 8);
```

- as an operator to know whether a function launches exceptions

```
template <class T> void foo()
    noexcept(noexcept(T())) {};
```

Lambdas

Function return type

A new way to specify function's return type

```
ReturnType fn_name(ArgType1, ArgType2);  //old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```

Function return type

A new way to specify function's return type

```
ReturnType fn_name(ArgType1, ArgType2);  //old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```

Advantages

- Allows to simplify inner type definition

```
class TheClass {
  typedef int inner_type;
  inner_type func();
}
TheClass::inner_type TheClass::func() {...}
auto TheClass::func() -> inner_type {...}
```

- will be used for lambdas

Lambdas

Definition

a lambda is a function with no name

Lambdas

Definition

a lambda is a function with no name

Python example

```
1  data = [1,9,3,8,3,7,4,6,5]
2
3  # without lambdas
4  def isOdd(n):
5    return n%2 == 1
6  print filter(isOdd, data)
7
8  # with lambdas
9  print filter(lambda n:n%2==1, data)
```

C++ Lambdas

Simplified syntax

```
1 [] (args) -> type {
2   code;
3 }
```

The type specification is optionnal

Usage example

```
1 std::vector<int> data{1,2,3,4,5};
2 for_each(begin(data), end(data),
3           [](int i) {
4             std::cout << "The square of " << i
5                       << " is " << i*i << std::endl;
6           });
```

Capture

Python code

```
1 increment = 3
2 data = [1,9,3,8,3,7,4,6,5]
3 map(lambda x : x + increment, data)
```

Capture

Python code

```
1  increment = 3
2  data = [1,9,3,8,3,7,4,6,5]
3  map(lambda x : x + increment, data)
```

First attempt in C++

```
1  int increment = 4;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [](int x) { std::cout << x+increment; });
5  std::cout << std::endl;
```

Capture

Python code

```
1  increment = 3
2  data = [1,9,3,8,3,7,4,6,5]
3  map(lambda x : x + increment, data)
```

First attempt in C++

```
1  int increment = 4;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [](int x) { std::cout << x+increment; });
5  std::cout << std::endl;
```

Error

```
error: 'increment' is not captured
  [](int x) { std::cout << x+increment; });
                                  ^
```

Capture

Variable capture

- external variables need to be explicitely captured
- captured variables are listed within initial []

Capture

Variable capture

- external variables need to be explicitely captured
- captured variables are listed within initial []

Example

```
1  int increment = 4;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [increment](int x) {
5             std::cout << x+increment;
6           });
7  std::cout << std::endl;
```

Default capture is by value

Code example

```
1  int sum = 0;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [sum](int x) { sum += x; });
```

Default capture is by value

Code example

```
1  int sum = 0;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [sum](int x) { sum += x; });
```

Error

```
error: assignment of read-only variable 'sum'
           [sum](int x) { sum += x; });
```

Default capture is by value

Code example

```
1  int sum = 0;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [sum](int x) { sum += x; });
```

Error

```
error: assignment of read-only variable 'sum'
           [sum](int x) { sum += x; });
```

Explanation

By default, variables are captured by value

Capture by reference

Simple example

In order to capture by reference, add '&' before the variable

```
1  int sum = 0;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [&sum](int x) { sum += x; });
```

Capture by reference

Simple example

In order to capture by reference, add '&' before the variable

```
1  int sum = 0;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [&sum](int x) { sum += x; });
```

Mixed case

One can of course mix values and references

```
1  int sum = 0, offset = 1;
2  std::vector<int> data{1,9,3,8,3,7,4,6,5};
3  for_each(begin(data), end(data),
4           [&sum, offset](int x) {
5             sum += x + offset;
6           });
```

Capture all

by value

```
[=](...) { ... };
```

Capture all

by value

```
[=](...) { ... };
```

by reference

```
[&](...) { ... };
```

Capture all

by value

```
[=](...) { ... };
```

by reference

```
[&](...) { ... };
```

exceptions

```
[&, b](...) { ... };
[=, &b](...) { ... };
```

Closures

Example

```
1  auto build_incrementer = [](int inc) {
2    return [inc](int value) { return value + inc; };
3  };
4  auto inc1 = build_incrementer(1);
5  auto inc10 = build_incrementer(10);
6  int i = 0;
7  i = inc1(i);   // i = 1
8  i = inc10(i);  // i = 11
```

How it works

- build_incrementer returns a function object
- this function's behavior depends on a parameter
- note how *auto* is useful here !

C++11 makes the STL usable

Before lambdas

```
1  struct Incrementer {
2    int m_inc;
3    Incrementer(int inc) : m_inc(inc) {}
4    int operator() (int value) {
5      return value + m_inc;
6    };
7  };
8  std::vector<int> v;
9  v.push_back(1); v.push_back(2); v.push_back(3);
10 std::transform(begin(v), end(v), begin(v),
11                  Incrementer(1));
12 for (std::vector<int>::iterator it = begin(v);
13       it != end(v);
14       it++) std::cout << *it << " ";
```

C++11 makes the STL usable

With lambdas

```
1  std::vector<int> v = {1, 2, 3};
2  int inc = 1;
3  std::transform(begin(v), end(v), begin(v),
4                 [inc](int value) {
5                   return value + inc;
6                 });
7  for (auto a : v) std::cout << a << " ";
```

C++11 makes the STL usable

With lambdas

```
1  std::vector<int> v = {1, 2, 3};
2  int inc = 1;
3  std::transform(begin(v), end(v), begin(v),
4                 [inc](int value) {
5                   return value + inc;
6                 });
7  for (auto a : v) std::cout << a << " ";
```

Conclusion

Use the STL !

Lambdas

Exercise Time

- go to code/lambdas
- look at the code (it's the solution to the stl exercise)
- use lambdas to simplify it

Move semantic

Move semantics : the problem

Non efficient code

```
1  template <class T>
2  void swap(T &a, T &b) {
3    T c = a;
4    a = b;
5    b = c;
6  }
7  std::vector<int> v, w;
8  for (int i = 0; i < 10000; i++) v.push_back(i);
9  for (int i = 0; i < 10000; i++) w.push_back(i);
10 swap(v, w);
```

Move semantics : the problem

Non efficient code

```
1  template <class T>
2  void swap(T &a, T &b) {
3    T c = a;
4    a = b;
5    b = c;
6  }
7  std::vector<int> v, w;
8  for (int i = 0; i < 10000; i++) v.push_back(i);
9  for (int i = 0; i < 10000; i++) w.push_back(i);
10 swap(v, w);
```

What really happens during swap

- 10k allocations + 10k releases
- 30k copies

Move semantics : the problem

Dedicated efficient code

```
1  std::vector<int> v, w;
2  for (int i = 0; i < 10000; i++) v.push_back(i);
3  for (int i = 0; i < 10000; i++) w.push_back(i);
4  v.swap(w);
```

Move semantics : the problem

Dedicated efficient code

```
1  std::vector<int> v, w;
2  for (int i = 0; i < 10000; i++) v.push_back(i);
3  for (int i = 0; i < 10000; i++) w.push_back(i);
4  v.swap(w);
```

What probably happens during swap

- 1 allocations + 1 releases
- 3 copies

only the pointers to underlying arrays were swapped

Move semantics : the problem

Another non efficient code

```
1  std::vector<int> vrandom(unsigned int n) {
2    std::vector<int> result;
3    for (int i = 0; i < n; i++) {
4      result.push_back(rand());
5    }
6    return result;
7  }
8  std::vector<int> v = vrandom(10000);
```

Move semantics : the problem

Another non efficient code

```
1  std::vector<int> vrandom(unsigned int n) {
2    std::vector<int> result;
3    for (int i = 0; i < n; i++) {
4      result.push_back(rand());
5    }
6    return result;
7  }
8  std::vector<int> v = vrandom(10000);
```

What really happens during assignment

- 10k allocations + 10k releases
- 10k copies

Move semantics : the problem

Dedicated efficient way

```
1  void vrandom(unsigned int n, std::vector<int> &v) {
2    for (int i = 0; i < n; i++) {
3      v.push_back(rand());
4    }
5  }
6  std::vector<int> v;
7  vrandom(10000, v);
```

Move semantics : the problem

Dedicated efficient way

```
1  void vrandom(unsigned int n, std::vector<int> &v) {
2    for (int i = 0; i < n; i++) {
3      v.push_back(rand());
4    }
5  }
6  std::vector<int> v;
7  vrandom(10000, v);
```

The ideal situation

Have a way to express that we move the vector's content

Move semantics

The idea

- a new type of reference : rvalue references
  - used for move semantic
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

Move semantics

The idea

- a new type of reference : rvalue references
  - used for move semantic
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

Practically

```
1  T(const T&  other); // copy construction
2  T(      T&& other); // move construction
3  T& operator=(const T&  other); // copy assignment
4  T& operator=(      T&& other); // move assignment
```

Move semantics

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
  - still, I advice to never leave inconsistent objects
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - e.g. when passing temporary

Move semantics

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
  - still, I advice to never leave inconsistent objects
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - e.g. when passing temporary

Practically

```
1  T a;
2  T b = a;       // 1. Copy assign
3  T c = T(2);    // 2. Move assign
4  T d = func(); // 3. Move assign
```

Move semantics

In some cases, you want to force a move

```
1  template <class T> void swap(T &a, T &b) {
2    T c = a;  // copy
3    a = b;    // copy
4    b = c;    // copy
5  }
```

Move semantics

In some cases, you want to force a move

```
1  template <class T> void swap(T &a, T &b) {
2    T c = a;  // copy
3    a = b;    // copy
4    b = c;    // copy
5  }
```

There are mainly two ways

- casting to an rvalue reference
- using the std::move function

```
1  T a;
2  T b = a;                          // Copy assign
3  T c = static_cast<T&&>(a);        // Move assign
4  T d = std::move(a);               // Move assign
```

Move semantics : the easy way

Use copy and swap idiom

- implement an efficient swap method to your class
  - preferably outside the class so that it is symetric
- use swap for move constructor
  - create empty object with constructor delegation
  - swap it with source
- use swap in move assignment
  - pass parameter by value
  - this should force creation of a local replica of source
  - as we are in the move assignment
    our move constructor will be called
    and source will be filled with an empty object
  - swap local object with *this
  - let local object be destructed when exiting the method
    this will actually destroy the original content of the target

Move semantics : the easy way

Practically

```
1  class Movable {
2    Movable();
3    Movable(Movable &&other) :
4      Movable() {          // constructor delegation
5      swap(*this, other);
6    }
7    Movable& operator=(Movable other) { // by value
8      swap(*this, other);
9      return *this;
10   }
11   friend void swap(Movable &a, Movable &b);
12 };
13 void swap(Movable &a, Movable &b);
```

Move Semantic

Exercise Time

- go to code/move
- look at the code and run it with callgrind
- understand how inefficient it is
- implement move semantic the easy way in NVector
- run with callgrind and see no improvement
- understand why and fix test.cpp
- see efficiency improvements

pointers and RAII

nullptr

Finally a C++ NULL pointer

- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

nullptr

Finally a C++ NULL pointer

- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

Example code

```
1  void* vp = nullptr;
2  int* ip = nullptr;
3  int i = NULL;       // OK -> bug ?
4  int i = nullptr;    // ERROR
```

Pointers : why they are error prone ?

They need initialization

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
  strncpy(s, ...);
} catch (...) { ... }
bar(s);
```

Pointers : why they are error prone ?

They need initialization — Seg Fault

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
  strncpy(s, ...);
} catch (...) { ... }
bar(s);
```

Pointers : why they are error prone ?

They need initialization — Seg Fault

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
```

They need to be released

```
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strncmp(s, ...)) return;
foo(s);
free(s);
```

Pointers : why they are error prone ?

They need initialization — Seg Fault

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
```

They need to be released — Memory leak

```
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strncmp(s, ...)) return;
foo(s);
free(s);
```

Pointers : why they are error prone ?

They need initialization — Seg Fault

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
```

They need to be released — Memory leak

```
char *s = (char*) malloc(...);
strncpy(s, ...);
```

They need clear ownership

```
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```

Pointers : why they are error prone ?

They need initialization — Seg Fault

```
char *s;
try {
  foo();
  s = (char*) malloc(...);
```

They need to be released — Memory leak

```
char *s = (char*) malloc(...);
strncpy(s, ...);
```

They need clear ownership — Who should release ?

```
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```

This problem exists for any resource

For example with a file

```
1  try {
2    FILE *handle = std::fopen(path, "w+");
3    if (0 == handle) { throw ... }
4    if (std::fputs(str, handle) == EOF) {
5      throw ...
6    }
7    fclose(handle);
8  } catch (...) { ... }
```

Resource Acquisition Is Initialization

Practically

Use object semantic to acquire/release resources

- wrap the resource inside an object
- acquire resource via object constructor
- release resource in destructor
- create this object on the stack so that it is automatically destructed when leaving the scope

RAII in practice

File class

```
1  class File {
2  public:
3    File(const char* filename) :
4      m_file_handle(std::fopen(filename, "w+")) {
5      if (m_file_handle == NULL) { throw ... }
6    }
7    ~File() { std::fclose(m_file_handle); }
8    void write (const char* str) {
9      if (std::fputs(str, m_file_handle) == EOF) {
10       throw ...
11     }
12   }
13 private:
14   FILE* m_file_handle;
15 };
```

RAII usage

Usage of File class

```
1  void log_function() {
2    // file opening, aka resource acquisition
3    File logfile("logfile.txt") ;
4
5    // file usage
6    logfile.write("hello logfile!") ;
7
8    // file is automatically closed by the call to
9    // its destructor, even in case of exception !
10 }
```

std::unique_ptr

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in <memory> header

std::unique_ptr

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in <memory> header

Usage

```
1  Foo *p = new Foo{};  // allocation
2  std::unique_ptr<Foo> uptr(p);
3  std::cout << uptr.get() << " points to "
4            << uptr->someMember << std::endl;
5  void f(std::unique_ptr<Foo> ptr);
6  f(std::move(uptr)); // transfer of ownership
7  // deallocation when exiting f
8  std::cout << uptr.get() << std::endl; // 0
```

Quizz

```
1  Foo *p = new Foo{};  // allocation
2  std::unique_ptr<Foo> uptr(p);
3  void f(std::unique_ptr<Foo> ptr);
4  f(uptr); // transfer of ownership
```

What do you expect ?

Quizz

```
1  Foo *p = new Foo{};  // allocation
2  std::unique_ptr<Foo> uptr(p);
3  void f(std::unique_ptr<Foo> ptr);
4  f(uptr); // transfer of ownership
```

What do you expect ?

Compilation Error

```
test.cpp:15:5: error: call to deleted constructor
of 'std::unique_ptr<Foo>'
  f(uptr);
    ^~~~

/usr/include/c++/4.9/bits/unique_ptr.h:356:7: note:
 'unique_ptr' has been explicitly marked deleted here
 unique_ptr(const unique_ptr&) = delete;
 ^
```

std::make_unique

- allocates directly a unique_ptr
- no new or delete calls anymore !

std::make_unique

- allocates directly a unique_ptr
- no new or delete calls anymore !

make_unique usage

```
1  // allocation of one Foo object,
2  // calling constructor with one argument
3  auto a = std::make_unique<Foo>(memberValue);
4  std::cout << a.get() << " points to "
5            << a->someMember << std::endl;
6  // allocation of an array of Foos
7  // calling default constructor
8  auto b = std::make_unique<Foo[]>(10);
9  // deallocations
```

RAII or raw pointers

When to use what ?

- Always use RAII for allocations
- You thus never have to deallocate !
- Use raw pointers for observer functions (or references)
  - remember that unique_ptr is move only

RAII or raw pointers

When to use what ?

- Always use RAII for allocations
- You thus never have to deallocate !
- Use raw pointers for observer functions (or references)
  - remember that unique_ptr is move only

A question of ownership

```
1  unique_ptr<T> producer();
2  void observer(T*);
3  void consumer(unique_ptr<T>);
4
5  unique_ptr<T> pt{producer()};
6  observer(pt.get());          // Keep ownership
7  consumer(std::move(pt));     // Transfer ownership
```

unique_ptr usage summary

It's about lifetime management

- Use unique_ptr in functions taking part to the lifetime management
- Otherwise use raw pointers or references

shared_ptr, make_shared

shared_ptr : a reference counting pointers

- wraps a regular pointer like unique_ptr
- has move and copy semantic
- uses internally reference counting
  - "Would the last person out, please turn off the lights ?"
- is thread safe, thus the reference counting is costly

make_shared : creates a shared_ptr

```
1  {
2    auto sp = std::make_shared<Foo>(); // #ref = 1
3    vector.push_back(sp);              // #ref = 2
4    set.insert(sp);                    // #ref = 3
5  } // #ref 2
```

Threads and async

Basic concurrency

Threading

- new object std::thread in <thread> header
- takes a function as argument of its constructor
- must be called on join or program is terminated

Basic concurrency

Threading

- new object std::thread in <thread> header
- takes a function as argument of its constructor
- must be called on join or program is terminated

Example code

```
1  void doSth() {...};
2  void doSthElse() {...};
3  int main() {
4    std::thread t1(doSth);
5    std::thread t2(doSthElse);
6    for (auto t: {&t1,&t2}) t->join();
7  }
```

The thread constructor

Can take a function and its arguments

```
1  void function(int j, double j) {...};
2  std::thread t1(function, 1, 2.0);
```

The thread constructor

Can take a function and its arguments

```
1  void function(int j, double j) {...};
2  std::thread t1(function, 1, 2.0);
```

Can take any function like object

```
1  struct AdderFunctor {
2    AdderFunctor(int i): m_i(i) {}
3    int operator() (int j) { return i+j; };
4    int m_i;
5  };
6  std::thread t2(AdderFunctor(2), 5);
7  int a;
8  std::thread t3([](int i) { return i+2; }, a);
9  std::thread t4([a]       { return a+2; });
```

Basic asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Basic asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Pratically

- std::async function launches and asynchronous task
- std::future template allows to handle the result

Basic asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Pratically

- std::async function launches and asynchronous task
- std::future template allows to handle the result

Example code

```
1 int computeSth() {...}
2 std::future<int> res = std::async(computeSth);
3 std::cout << res->get() << std::endl;
```

Mixing the two

Is async running concurrent code ?

- it depends !
- you can control this with a launch policy argument

  std::launch::async spawns a thread for immediate execution
  std::launch::deferred causes lazy execution in current thread

  - execution starts when get() is called
- default is not specified !

Mixing the two

Is async running concurrent code ?

- it depends !
- you can control this with a launch policy argument

  std::launch::async spawns a thread for immediate execution
  std::launch::deferred causes lazy execution in current thread
  - execution starts when get() is called
- default is not specified !

Usage

```
1 int computeSth() {...}
2 auto res = std::async(std::launch::async,
3                       computeSth);
4 auto res2 = std::async(std::launch::deferred,
5                        computeSth);
```

Fine grained control on asynchronous execution

std::packaged_task template

- creates an asynchronous version of any function like object
  - identical arguments
  - returns a std::future
- provides access to the returned future
- associated with threads, gives full control on execution

Fine grained control on asynchronous execution

std::packaged_task template

- creates an asynchronous version of any function like object
  - identical arguments
  - returns a std::future
- provides access to the returned future
- associated with threads, gives full control on execution

Usage

```
1  int task() { return 42; }
2  std::packaged_task<int()> pckd_task(task);
3  auto future = pckd_task.get_future();
4  pckd_task();
5  std::cout << future.get() << std::endl;
```

Mutexes

6 C++11/14 features
- Introduction
- Constant Expressions
- Range based loops
- auto keyword
- override and final
- non-member begin/end
- Initializers
- Constructors
- Exceptions
- Lambdas
- Move semantic
- pointers and RAII
- Threads and async
- Mutexes

Races

Example code

```
1  int a = 0;
2  void inc() { a++; };
3  void inc100() {
4    for (int i=0; i < 100; i++) inc();
5  };
6  int main() {
7    std::thread t1(inc100);
8    std::thread t2(inc100);
9    for (auto t: {&t1,&t2}) t->join();
10   std::cout << a << std::endl;
11 }
```

Races

Example code

```
1  int a = 0;
2  void inc() { a++; };
3  void inc100() {
4    for (int i=0; i < 100; i++) inc();
5  };
6  int main() {
7    std::thread t1(inc100);
8    std::thread t2(inc100);
9    for (auto t: {&t1,&t2}) t->join();
10   std::cout << a << std::endl;
11 }
```

What do you expect ? Try it in code/race

Races

Example code

```
1  int a = 0;
2  void inc() { a++; };
3  void inc100() {
4    for (int i=0; i < 100; i++) inc();
5  };
6  int main() {
7    std::thread t1(inc100);
8    std::thread t2(inc100);
9    for (auto t: {&t1,&t2}) t->join();
10   std::cout << a << std::endl;
11 }
```

What do you expect ? Try it in code/race

Anything between 100 and 200 !!!

Atomicity

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Atomicity

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Atomicity

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Usually not. It behaves like :

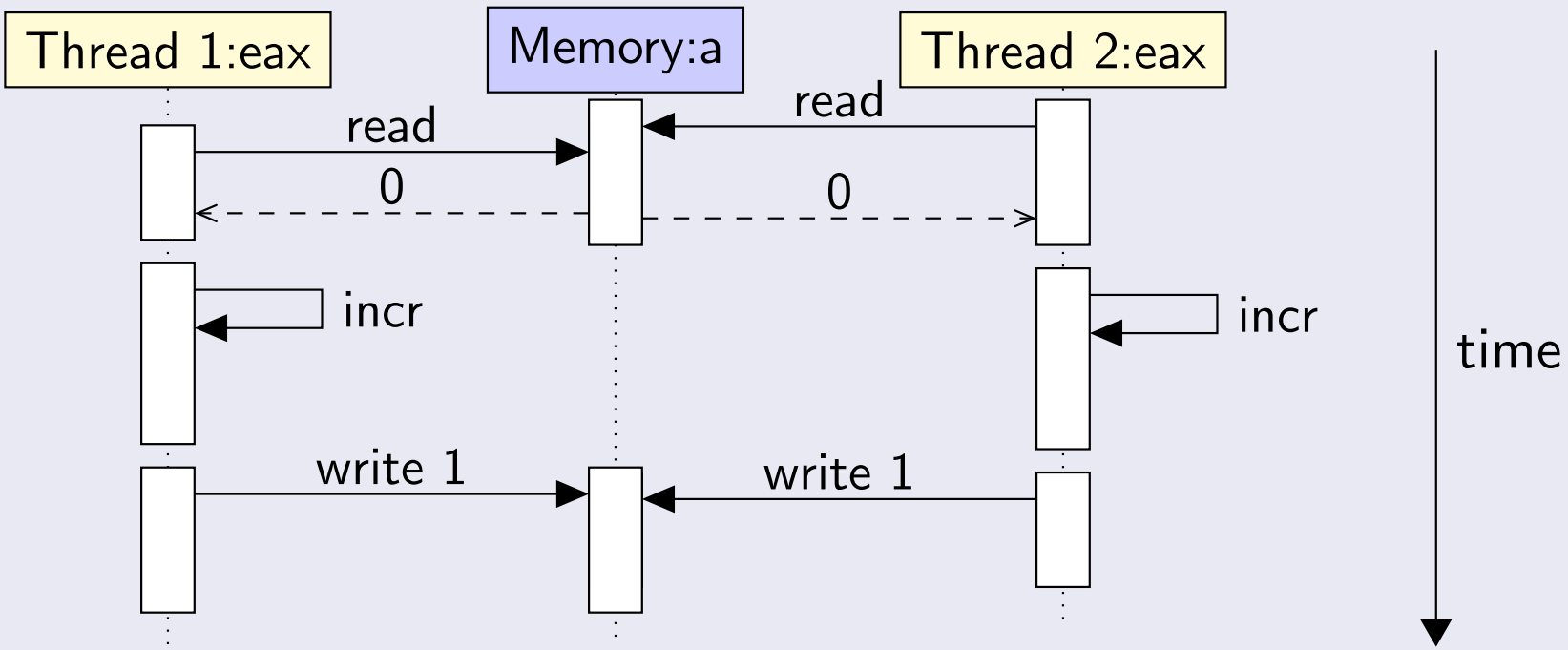
```
eax = a          // memory to register copy
increase eax     // increase (atomic CPU instruction)
a = eax          // copy back to memory
```

Timing

Code

```
eax = a          // memory to register copy
increase eax     // increase (atomic CPU instruction)
a = eax          // copy back to memory
```

For 2 threads



Mutexes

Concept

- a lock to serialize access to a non atomic piece of code

Mutexes

Concept

- a lock to serialize access to a non atomic piece of code

The objects

std::mutex	in the mutex header
std::lock_guard	for an RAII version of it
std::unique_lock	same and can be released/relocked

Mutexes

Concept

- a lock to serialize access to a non atomic piece of code

The objects

std::mutex	in the mutex header
std::lock_guard	for an RAII version of it
std::unique_lock	same and can be released/relocked

Practically

```
1  int a = 0;
2  std::mutex m;
3  void inc() {
4    std::lock_guard<std::mutex> guard(m);
5    a++;
6  };
```

Mutexes

Exercise Time

- Go to code/race
- Look at the code and try it
  See that it has a race condition
- Use a mutex to fix the issue
- See the difference in execution time

Dead lock

Scenario

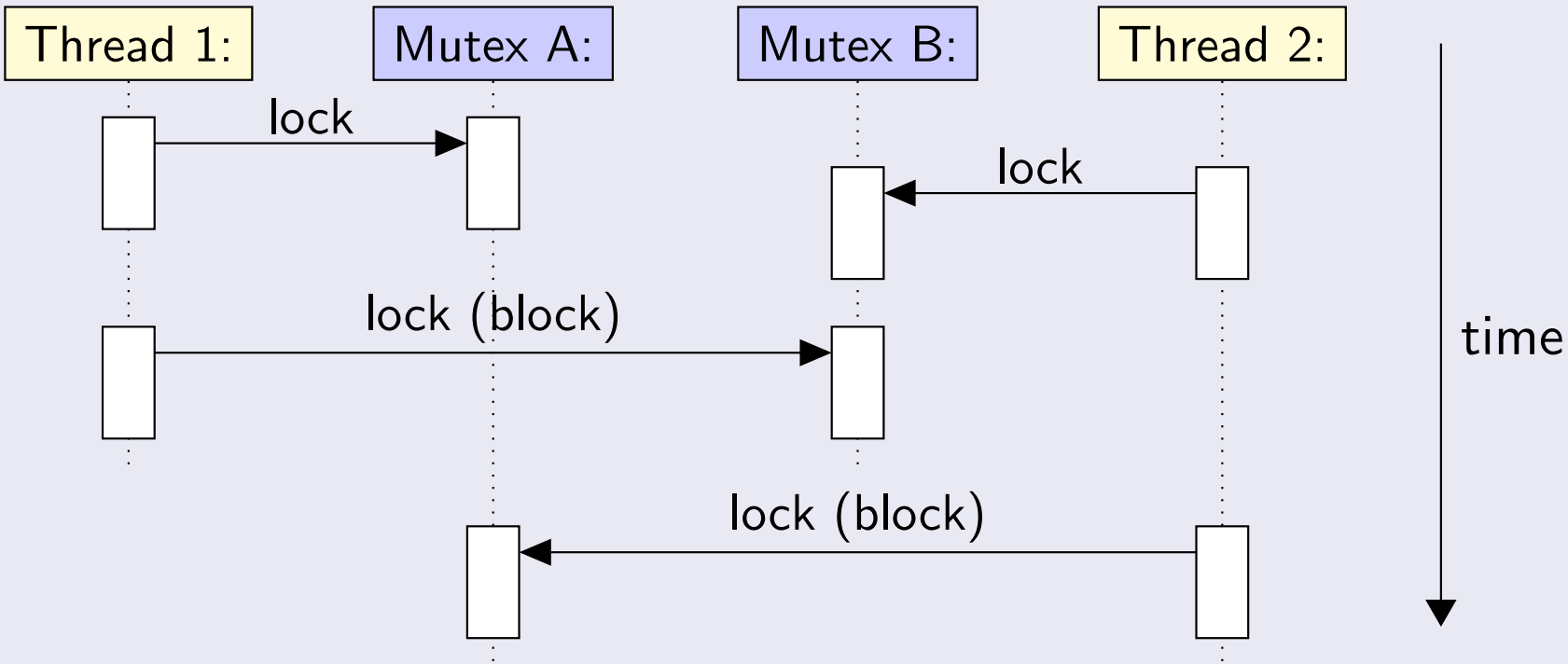
- 2 mutexes, 2 threads
- locking order different in the 2 threads

Dead lock

Scenario

- 2 mutexes, 2 threads
- locking order different in the 2 threads

Sequence diagram



How to avoid dead locks

Possible solutions

- Never take several locks
  - Or add master lock protecting the locking phase
- Respect a strict order in the locking across all threads
- Do not use locks
  - Use other techniques, e.g. queues

Condition variables

How to express thread dependencies

- Allows a thread to sleep until a given condition is satisfied
- std::condition_variable object from condition_variable header

Condition variables

How to express thread dependencies

- Allows a thread to sleep until a given condition is satisfied
- std::condition_variable object from condition_variable header

Usage

- wraps an RAII lock around a mutex
- wait() will hang until the condition is met
  - you can have several waiters sharing the same mutex
- notify_one() will wake up on waiter
- notify_all() will wake up all waiters

Condition variable usage

Example code

```
1  int value = -1;
2  std::mutex mutex;
3  std::condition cond;
4  auto t = std::thread([] () {
5    value = ... long process ...;
6    cond.notify_all();
7  });
8  auto t = std::thread([] () {
9    std::unique_lock<std::mutex> lock{mutex};
10   cond.wait(lock, [] { return value > 0; });
11   ... use value ...
12 });
13 { std::unique_lock<std::mutex> lock{mutex};
14   cond.wait(lock, [] { return value > 0; });
15   std::cout << value << std::endl; }
```

C++17 features

Nested namespace

7 C++17 features
- Nested namespace
- Copy elision
- [[fallthrough]]
- Structured Binding Declarations
- init-statements for if and switch
- new STL types

Nested namespaces

Easier way to declare nested namespaces

C++14

```
1  namespace A {
2    namespace B {
3      namespace C {
4        //...
5      }
6    }
7  }
```

C++17

```
1  namespace A::B::C {
2    //...
3  }
```

Copy elision

7 C++17 features
- Nested namespace
- Copy elision
- [[fallthrough]]
- Structured Binding Declarations
- init-statements for if and switch
- new STL types

Guaranteed copy elision

What is copy elision

```
1  struct Foo { ... };
2  Foo f() {
3    return Foo();
4  }
5  int main() {
6    // compiler was authorised to elude the copy
7    Foo foo = f();
8  }
```

New in C++17

The elision is guaranteed.

Guaranteed copy elision

Allows to write code not allowed with C++14 (would not compile)

One case where the guarantee is needed

```
1  struct Foo {
2    Foo() { ... }
3    Foo(const Foo &) = delete;
4    Foo(const Foo &&) = delete;
5  };
6  Foo f() {
7    return Foo();
8  }
9  int main() {
10   Foo foo = f();
11 }
```

[[fallthrough]]

7 C++17 features

- Nested namespace
- Copy elision
- [[fallthrough]]
- Structured Binding Declarations
- init-statements for if and switch
- new STL types

[[fallthrough]] attribute

C++14

```
1  switch (c) {
2    case 'a':
3      f(); // Warning emitted
4    case 'c':
5      h();
6  }
```

C++17

```
1  switch (c) {
2    case 'a':
3      f();
4      [[fallthrough]]; // Warning suppressed
5    case 'c':
6      h();
7  }
```

Structured Binding Declarations

7. C++17 features
   - Nested namespace
   - Copy elision
   - [[fallthrough]]
   - Structured Binding Declarations
   - init-statements for if and switch
   - new STL types

Structured Binding Declarations

Helps when using tuples as a return type.
Automatically creates variables and ties them.

C++14

```
1  int a = 0;
2  double b = 0.0;
3  long c = 0;
4  // a, b, c need to be declared first
5  std::tie(a, b, c) = tuple;
```

C++17

```
1  auto [ a, b, c ] = tuple;
```

init-statements for if and switch

init-statements for if and switch

Allows to simplify if and switch statements

C++14

```
1  auto val = GetValue();
2  if (condition(val)) {
3    // on success
4  } else {
5    // on false...
6  }
```

C++17

```
1  if (auto val = GetValue(); condition(val)) {
2    // on success
3  } else {
4    // on false...
5  }
```

val is visible only inside the if and else statements

new STL types

7. C++17 features
   - Nested namespace
   - Copy elision
   - [[fallthrough]]
   - Structured Binding Declarations
   - init-statements for if and switch
   - new STL types

Some new STL types

std::optional

- manages an optional contained value
- contextually converted to bool
- useful for the return value of a function that may fail

std::any

- a type-safe container for single values of any type
- the any_cast function provides type-safe access
- and throws std::bad_any_cast for bad access

std::variant

- a type-safe union
- std::get reads the value of the variant
- and throws std::bad_variant_access for bad accesses

Expert C++11/14/17

Perfect forwarding

The problem

Trying to write a generic wrapper function

```
1 template <typename T>
2 void wrapper(T arg) {
3   func(arg);
4 }
```

Example usage :

- emplace_back
- make_unique

Why is is not so simple ?

```
1  template <typename T>
2  void wrapper(T arg) {
3    func(arg);
4  }
```

What about references ?

what if func takes references to avoid copies ?

wrapper would force a copy and we fail to use references

Second try, second failure ?

```
1  template <typename T>
2  void wrapper(T &arg) {
3    func(arg);
4  }
5  wrapper(42);
6  // invalid initialization of
7  // non-const reference from
8  // an rvalue
```

const ref won't help : you may want to pass something non const and rvalue are not yet supported...

The solution : cover all cases

```
1  template <typename T>
2  void wrapper(T& arg) { func(arg); }
3
4  template <typename T>
5  void wrapper(const T& arg) { func(arg); }
6
7  template <typename T>
8  void wrapper(T&& arg) { func(arg); }
```

The new problem : scaling to n arguments

```
1  template <typename T1, typename T2>
2  void wrapper(T1& arg1, T2& arg2)
3  { func(arg1, arg2); }
4
5  template <typename T1, typename T2>
6  void wrapper(const T1& arg1, T2& arg2)
7  { func(arg1, arg2); }
8
9  template <typename T1, typename T2>
10 void wrapper(T1& arg1, const T2& arg2)
11 { func(arg1, arg2); }
12 ...
```

Exploding complexity

3^n complexity

you do not want to try n = 5...

Reference collapsing in C++98

Reference to references

They are forbidden in C++

But still may happen

```
1  template <typename T>
2  void foo(T t) {
3    T& k = t;
4    ...
5  }
6  int ii = 4;
7  foo<int&>(ii);
```

Practically

all compilers were collapsing the 2 references

Reference collapsing in C++11

rvalues have been added

- what about int&& & ?
- and int && && ?

C++11 standardization

The rule is simple : & always wins

&& &, & &&, & & $\rightarrow$ &

&& && $\rightarrow$ &&

rvalue in type-deducing context

```
1  template <class T>
2  void func(T&& t) {}
```

In this context, && is not an rvalue
It means that the T type depends on the arguments passed to func

- if an lvalue of type U is given, T is deduced to U&
- if an rvalue, T is deduced to U

```
1  func(4);          // rvalue -> T is int
2  double d = 3.14;
3  func(d);          // lvalue -> T is double&
4  float f() {...}
5  func(f());        // rvalue -> T is float
6  int foo(int i) {
7    func(i);        // lvalue -> T is int&
8  }
```

std::remove_reference

Some template trickery removing reference from a type

```
1  template< class T >
2  struct remove_reference
3  {typedef T type;};
4
5  template< class T >
6  struct remove_reference<T&>
7  {typedef T type;};
8
9  template< class T >
10 struct remove_reference<T&&>
11 {typedef T type;};
```

If T is a reference type, T::type is the type referred to by T. Otherwise T::type is T.

std::forward

Another template trickery keeping references and mapping non reference types to rvalue references

```
1 template<class T>
2 T&& forward(typename std::remove_reference<T>::type& t)
3   noexcept {
4   return static_cast<T&&>(t);
5 }
```

How it works

- if T is int, if returns int &&
- if T is int&, it returns int& && ie int&

Perfect forwarding

Putting it all together

```
1 template <typename T>
2 void wrapper(T&& arg) {
3   func(forward<T>(arg));
4 }
```

How it works

- if we pass an rvalue to T (U&&)
  - arg will be of type U&&
  - func will be called with a U&&
- if we pass an lvalue to T (U&)
  - arg will be of type U&
  - func will be called with a U&

Real life example

```
1  template<typename T, typename... Args>
2  unique_ptr<T> make_unique(Args&&... args) {
3    return unique_ptr<T>
4      (new T(std::forward<Args>(args)...));
5  }
```

Variadic templates

Basic variadic template

The idea

- a template with an arbitrary number of parameters
- ... syntax as in good old printf
- using recursivity and specialization for stopping it

Practically

```
1  template<typename T, typename... Args>
2  T adder(T first, Args... args) {
3    return first + adder(args...);
4  }
5  template<typename T>
6  T adder(T v) {
7    return v;
8  }
9  long sum = adder(1, 2, 3, 8, 7);
```

A couple of remarks

About performance

- do not be afraid by recursivity
- everything is at compile time !
- unlike C style variadic functions

Why is it better than variadic functions

- it's more performant
- type safety is included
- it applies to everything, including objects

Variadic templated class

The tuple example

```
1  template <class... Ts> struct tuple {};
2
3  template <class T, class... Ts>
4  struct tuple<T, Ts...> : tuple<Ts...> {
5    tuple(T t, Ts... ts) :
6      tuple<Ts...>(ts...), m_tail(t) {}
7    T m_tail;
8  };
9
10 tuple<double, uint64_t, const char*>
11   t1(12.2, 42, "big");
```

SFINAE

8 Expert C++11/14/17
- Perfect forwarding
- Variadic templates
- SFINAE

Substitution Failure Is Not An Error

The main idea

- substitution replaces template parameters with the provided types or values
- if it leads to an invalid code, do not fail but try other overloads

Example

```
1  template <typename T>
2  void f(const T& t,
3         typename T::iterator* it = nullptr) { }
4  void f(...) { }   // ``sink'' function
5
6  f(1); // Calls void f(...)
```

decltype

The main idea

- gives the type of the of the expression it will evaluate
- at compile time

Example

```
1  struct A { double x; };
2  A a;
3  decltype(a.x) y;          // double
4  decltype((a.x)) z = y; // double& (lvalue)
5
6  template<typename T, typename U>
7  auto add(T t, U u) -> decltype(t + u);
8  // return type depends on template parameters
```

declval

The main idea

- gives you a "fake reference" to an object at compile time
- useful for types that cannot be easily constructed

Example

```
1  struct Default {
2    int foo() const { return 1; }
3  };
4  struct NonDefault {
5    NonDefault(int i) { }
6    int foo() const { return 1; }
7  };
8  decltype(Default().foo()) n1 = 1;      // int
9  decltype(NonDefault().foo()) n2 = n1; // error
10 decltype(std::declval<NonDefault>().foo()) n2 = n1;
```

true_type and false_type

The main idea

- encapsulate a constexpr boolean "true" and "false"
- can be inherited
- constexpr

Example

```
1  struct testStruct : std::true_type { };
2  constexpr bool testVar = testStruct();
3  bool test = testStruct::value; // true
```

Using SFINAE for instrospection

The main idea

- use a template specialization that may or may not create valid code
- use SFINAE to choose between them
- inherit from true/false_type

Example

```
1  template <typename T, typename = void>
2  struct hasFoo : std::false_type {};
3
4  template <typename T>
5  struct hasFoo<T, decltype(std::declval<T>().foo())>
6    : std::true_type {};
7
8  std::cout << hasFoo<MyType>::value << std::endl;
```

SFINAE and the STL

Lot's of useful stuff there

enable_if

```
1  template<bool B, class T = void>
2  struct enable_if {};
3  template<class T>
4  struct enable_if<true, T> { typedef T type; };
```

- If B is true, has a typedef `type` to type T
- otherwise, has no `type` typedef

is_*< T > (float/signed/object/final/abstract/...)

- Checks whether T is ...
- At compile time
- Has member `value`, as boolean telling whether it was

Gaudi usage example

```
1  constexpr struct deref_t {
2    template
3      <typename In,
4       typename = typename std::enable_if
5                  <!std::is_pointer<In>::value>::type>
6    In& operator()( In& in ) const { return in; }
7
8    template <typename In>
9    In& operator()( In* in ) const {
10     assert(in!=nullptr); return *in;
11   }
12 } deref {};
```

Back to variadic templated class

The tuple get method

```
1  template <class T, class... Ts>
2  struct elem_type_holder<0, tuple<T, Ts...>> {
3    typedef T type;
4  };
5
6  template <size_t k, class T, class... Ts>
7  struct elem_type_holder<k, tuple<T, Ts...>> {
8    typedef typename elem_type_holder
9       <k - 1, tuple<Ts...>>::type type;
10 };
```

Back to variadic templated class

The tuple get method

```
1  template <size_t k, class... Ts>
2  typename std::enable_if<k == 0,
3    typename elem_type_holder
4      <0, tuple<Ts...>>::type&>::type
5  get(tuple<Ts...>& t) {
6    return t.m_tail;
7  }
8  template <size_t k, class T, class... Ts>
9  typename std::enable_if<k != 0,
10   typename elem_type_holder
11     <k-1, tuple<Ts...>>::type&>::type
12 get(tuple<T, Ts...>& t) {
13   tuple<Ts...>& base = t;
14   return get<k - 1>(base);
15 }
```

Marrying C++and python

Writing a python module

How to build a python module around C++code

C++code : mandel.hpp

```
1   int mandel(const Complex &a);
```

Basic Module(1) : wrap your method

mandelModule.cpp

```
1  #include <Python.h>
2  #include "mandel.hpp"
3  static PyObject * mandel_wrapper(PyObject * self,
4                                   PyObject * args) {
5    // Parse Input
6    float r, i;
7    if (!PyArg_ParseTuple(args, "ff", &r, &i))
8      return NULL;
9    // Call C++ function
10   int result = mandel(Complex(r, i));
11   // Build returned objects
12   return Py_BuildValue("i", result);
13 }
```

Basic Module(2) : create the python module

mandelModule.cpp

```
1   // declare the modules' methods
2   static PyMethodDef MandelMethods[] = {
3       {"mandel", mandel_wrapper, METH_VARARGS,
4       "computes nb of iterations for mandelbrot set"},
5       {NULL, NULL, 0, NULL}
6   };
7   // initialize the module
8   PyMODINIT_FUNC initmandel(void) {
9     (void) Py_InitModule("mandel", MandelMethods);
10  }
```

Basic Module(3) : use it

mandel.py

```
1  from mandel import mandel
2  v = mandel(0.7, 1.2)
```

Marrying C++ and C

A question of mangling

Mangling

the act of converting the name of variable or function to a symbol name in the binary code

C versus C++ symbol names

- C uses bare function name
- C++ allows overloading of functions by taking the signature into account
- so C++ mangling has to contain signature

C mangling

Source : file.c

```
1  float sum(float a, float b);
2  int square(int a);
3  // won't compile : conflicting types for square
4  // float square(float a);
```

Binary symbols : file.o

```
# nm file.o
000000000000001a T square
0000000000000000 T sum
```

C++ mangling

Source : file.cpp

```
1  float sum(float a, float b);
2  int square(int a);
3  // ok, signature is different
4  float square(float a);
```

Binary symbols : file.o

```
# nm file.o
0000000000000000 T _Z3sumff
000000000000002a T _Z6squaref
000000000000001a T _Z6squarei
```

Forcing C mangling in C++

extern "C"

These functions will use C mangling :

```
1 extern "C" {
2   float sum(float a, float b);
3   int square(int a);
4 }
```

Forcing C mangling in C++

extern "C"

These functions will use C mangling :

```
1 extern "C" {
2   float sum(float a, float b);
3   int square(int a);
4 }
```

You can now call these C++functions from C code

Forcing C mangling in C++

extern "C"

These functions will use C mangling :

```
1 extern "C" {
2   float sum(float a, float b);
3   int square(int a);
4 }
```

You can now call these C++functions from C code

Limitations

- no C++types should go out
- no exceptions either (use noexcept here)
- member functions cannot be used
  - they need to be wrapped one by one

Using the ctypes module

The ctypes python module

From the documentation

- provides C compatible data types
- allows calling functions in DLLs or shared libraries
- can be used to wrap these libraries in pure Python

ctypes : usage example

C++code : mandel.hpp

```
1  int mandel(const Complex &a);
```

"C" code : mandel_cwrapper.hpp

```
1  extern "C" {
2    int mandel(float r, float i) {
3      return mandel(Complex(r, i));
4    };
5  }
```

calling the mandel library

```
1  from ctypes import *
2  libmandel = CDLL('libmandelc.so')
3  v = libmandel.mandel(c_float(0.3), c_float(1.2))
```

Marrying C++ and python

Exercise Time

- go to code/python
- look at the original python code mandel.py
- time it
- look at the code in mandel.hpp/cpp
- look at the python module mandel_module.cpp
- compile and modify mandel.py to use it
- see the gain in time
- look at the C wrapper in mandel_cwrapper.cpp
- modify mandel.py to use libmandelc directly with ctypes

This is the end

Questions ?

http://cern.ch/sponce/C++Course