HEP C^{++} course

B. Gruber, S. Hageboeck, S. Ponce, C. Doglioni (minor edits)
caterina.doglioni@cern.ch

University of Manchester

January 11, 2024

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 it is already too long for the time we have
  - in this session we will only be able to go through the basics
  - if we have time after the exercises in the afternoon, we'll go through classes and OOP

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

Where to find latest version ?

- full sources at github.com/hsf-training/cpluspluscourse

More courses

The HSF Software Training Center

A set of course modules on more software engineering aspects prepared from within the HEP community

- Unix shell
- Python
- Version control (git, gitlab, github)
- ...

https://hepsoftwarefoundation.org/training/curriculum.html

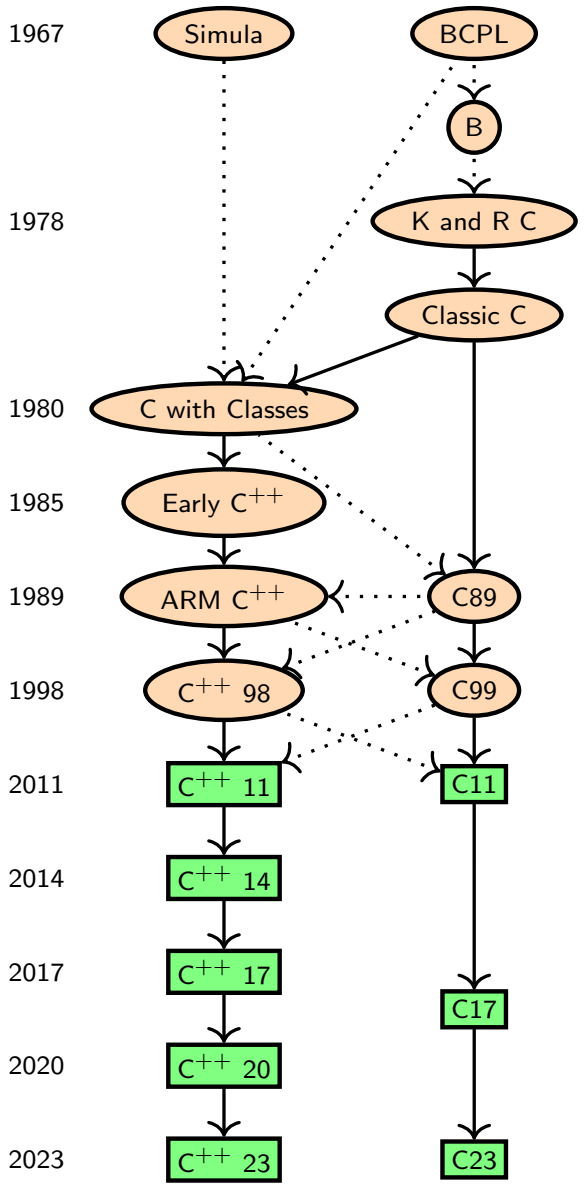
Outline

Detailed outline

History and goals

History

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 all C++ specs up to C++ 20 (only partially)
- Each slide will be marked with first spec introducing the feature

C++ 11, C++ 14, C++ 17, C++ 20, C++ 23, C++ 26...

Status

- A new C++ specification every 3 years
  - C++ 23 complete since 11th of Feb. 2023, awaiting ISO ballot
  - work on C++ 26 has begun
- Bringing each time a lot of goodies

C++ 11, C++ 14, C++ 17, C++ 20, C++ 23, C++ 26...

Status

- A new C++ specification every 3 years
  - C++ 23 complete since 11th of Feb. 2023, awaiting ISO ballot
  - work on C++ 26 has begun
- Bringing each time a lot of goodies

How to use C++ XX features

- Use a compatible compiler
- add `-std=c++xx` to compilation flags
- e.g. `-std=c++17`

C++	gcc	clang
11	≥4.8	≥3.3
14	≥4.9	≥3.4
17	≥7.3	≥5
20	>11	>12

Table: Minimum versions of gcc and clang for a given C++ version

Why we use it?

Why is C++ our language of choice?

Adapted to large projects

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

Why is C++ our language of choice?

Adapted to large projects

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

Fast

- compiled (unlike Java, C#, Python, ...)
- allows to go close to hardware when needed

Why is C++ our language of choice?

Adapted to large projects

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

Fast

- compiled (unlike Java, C#, Python, ...)
- allows to go close to hardware when needed

What we get

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

Language basics

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 until end of line
2  int i;
3
4  /* multiline comment
5   * in case we need to say more
6   */
7  double /* or something in between */ d;
8
9  /**
10  * Best choice : doxygen compatible comments
11  * \brief checks whether i is odd
12  * \param i input
13  * \return true if i is odd, otherwise false
14  * \see https://www.doxygen.nl/manual/docblocks.html
15  */
16 bool isOdd(int i);
```

Basic types(1)


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

Basic types(2)


```
1  int i = -123456;              // min 16, usually 32 bit
2  unsigned int i = 1234567; // min 16, usually 32 bit
3
4  long l = 0L                   // min 32 bit
5  unsigned long l = 0UL;    // min 32 bit
6
7  long long ll = 0LL;           // min 64 bit
8  unsigned long long l = 0ULL; // min 64 bit
9
10 float f = 1.23f;              // 32 (1+8+23) bit float
11 double d = 1.23E34;           // 64 (1+11+52) bit float
12 long double ld = 1.23E34L // min 64 bit float
```

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 !
```

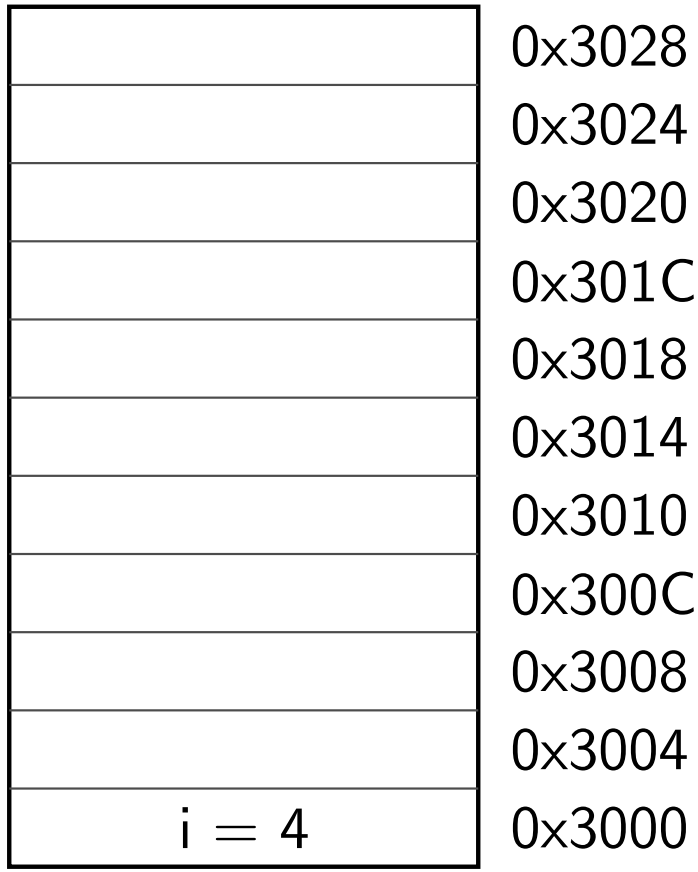
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```

Pointers

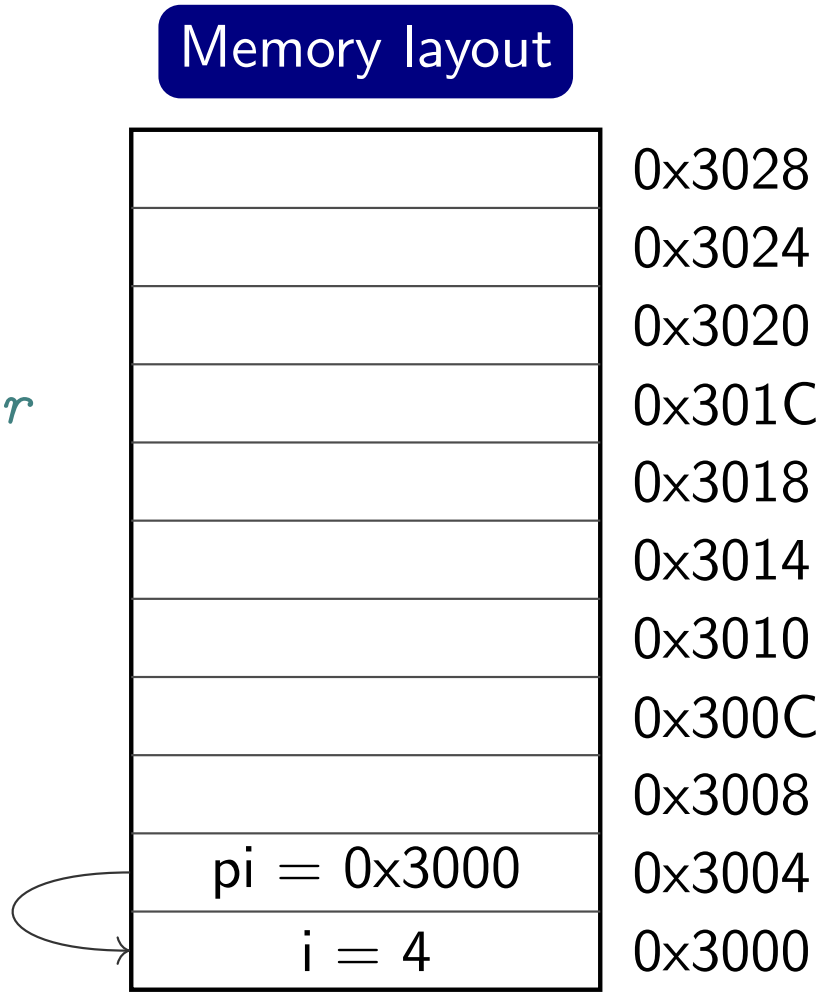

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```

Memory layout



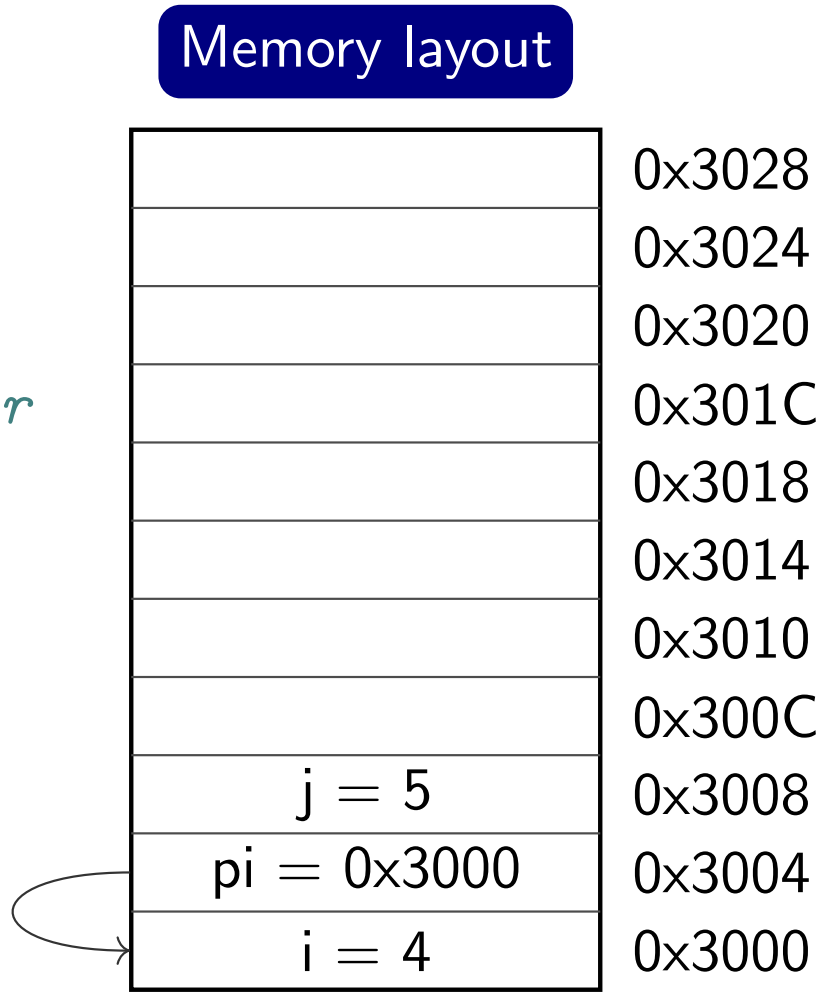
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



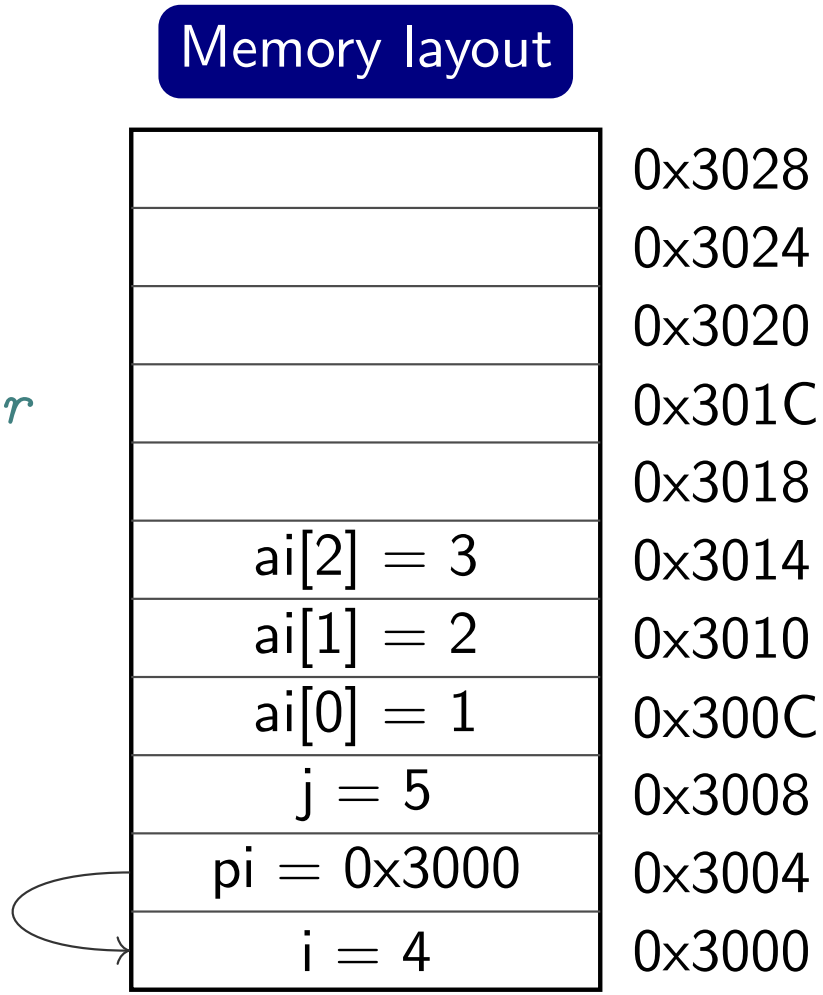
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



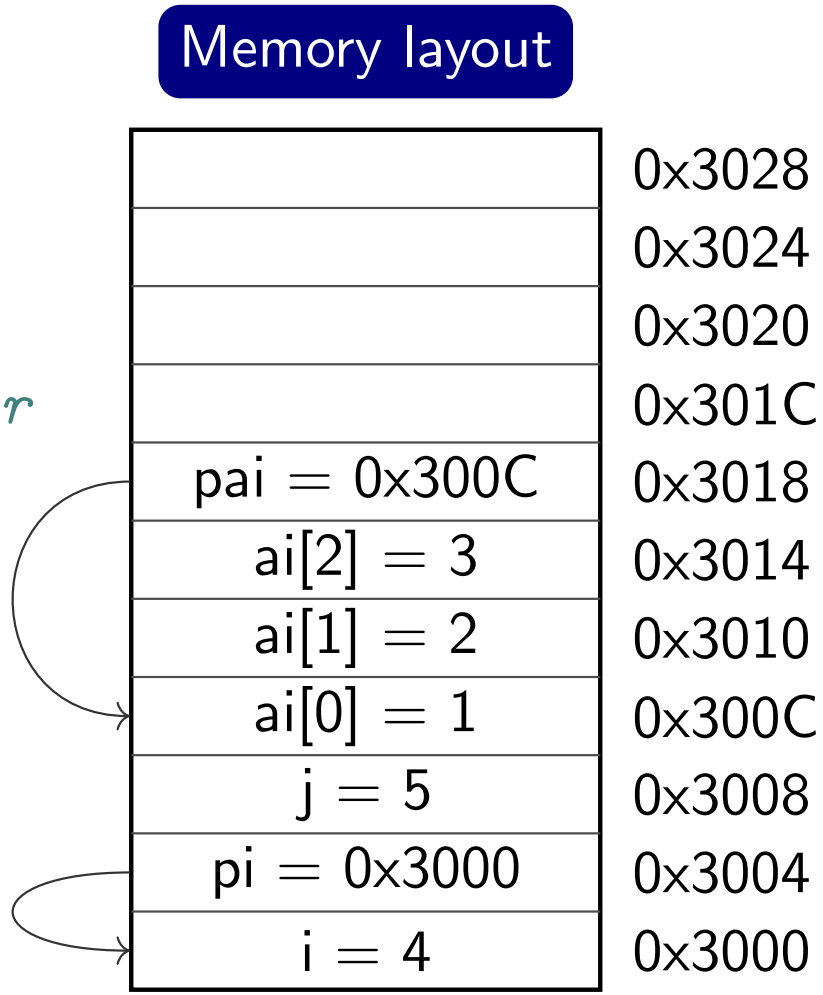
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



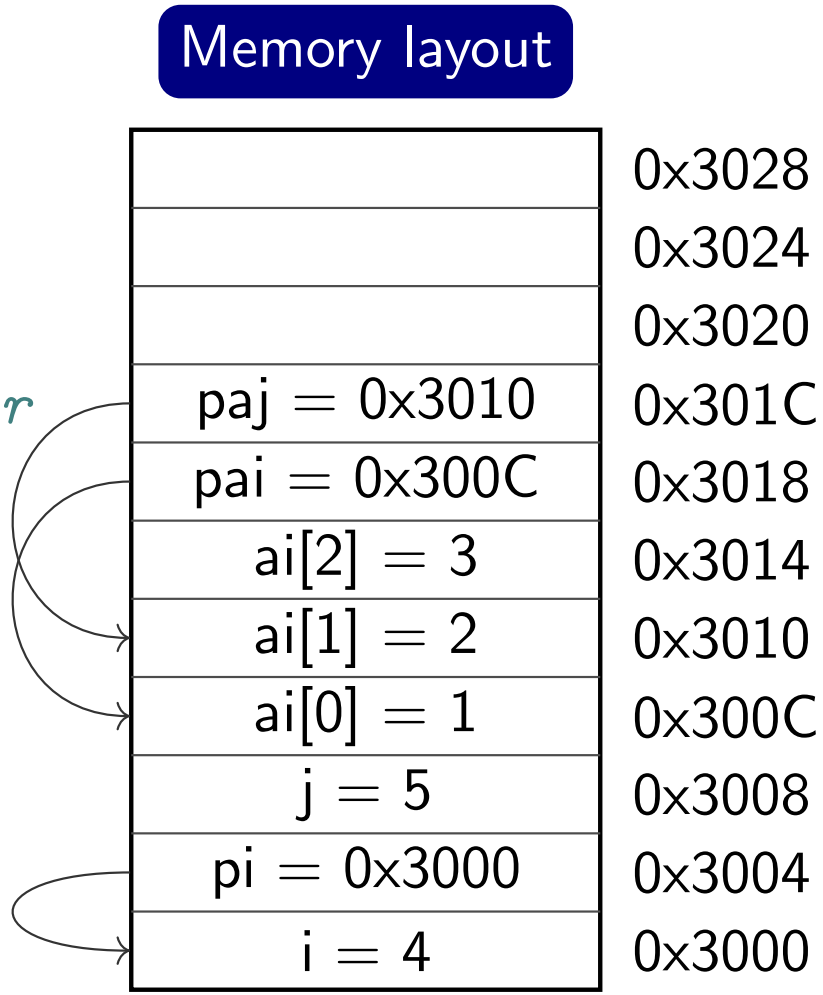
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



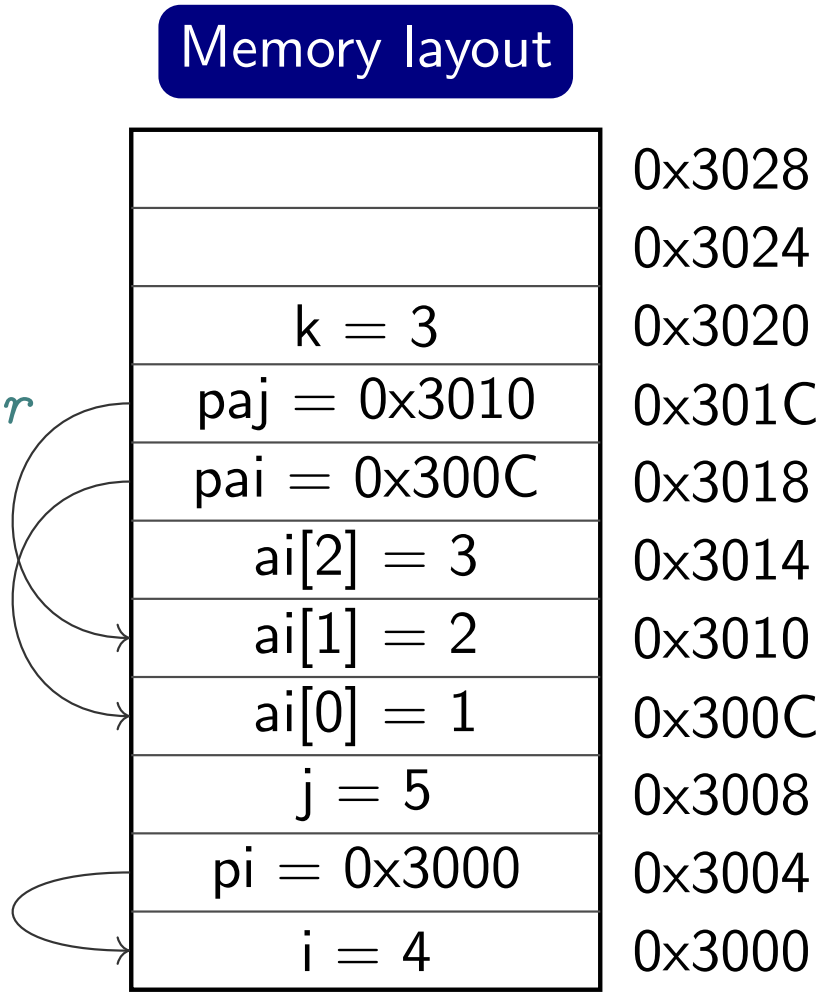
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



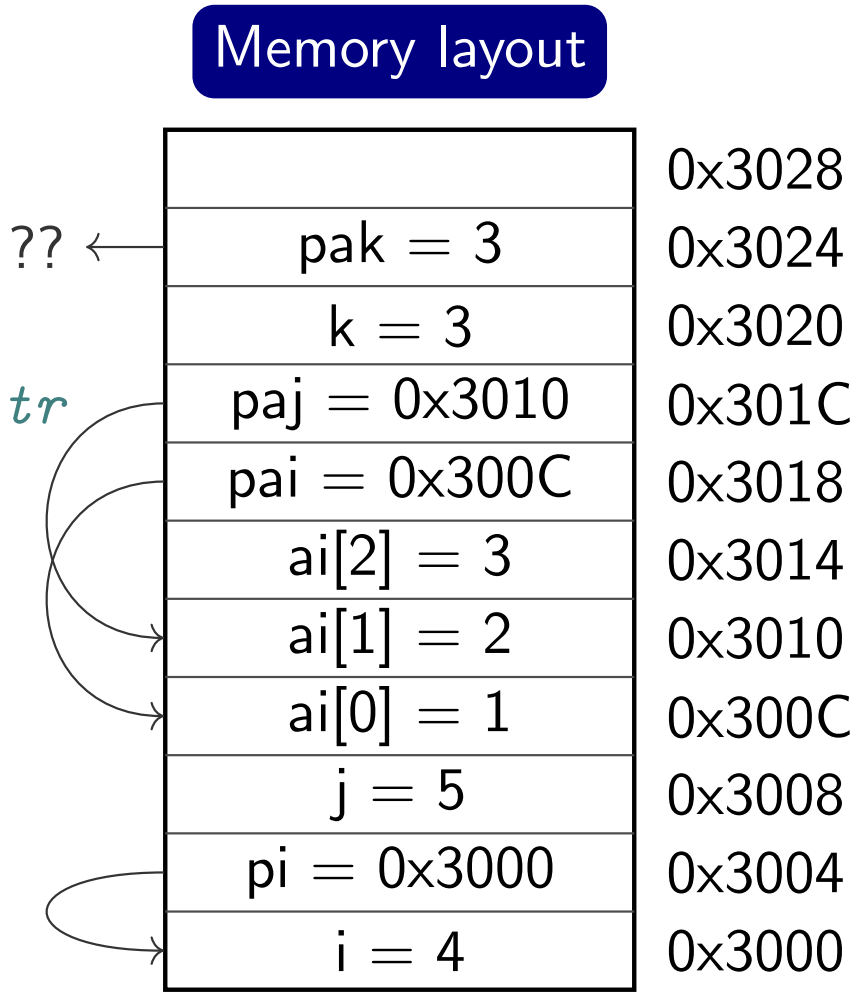
Pointers


```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



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; // decay to ptr
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



nullptr

A pointer to nothing

- if a pointer doesn't point to anything, set it to `nullptr`
  - useful to e.g. mark the end of a linked data structure
  - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C++ 11)
- triggers compilation error when assigned to integer

nullptr

A pointer to nothing

- if a pointer doesn't point to anything, set it to `nullptr`
  - useful to e.g. mark the end of a linked data structure
  - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C++ 11)
- triggers compilation error when assigned to integer

Example code

```
1  int* ip = nullptr;
2  int i = NULL;       // compiles, bug?
3  int i = nullptr;    // ERROR
```

Manual dynamic arrays using C++


```
1  #include <cstdlib>
2  #include <cstring>
3
4  // allocate array of 10 ints
5  int* ai = new int[10];   // uninitialized
6  int* ai = new int[10]{}; // zero-initialized
7
8  delete[] ai; // release array memory
9
10 // allocate a single int
11 int* pi = new int;
12 int* pi = new int{};
13 delete pi; // release scalar memory
```

Good practice: Don't use manual memory management

Use `std::vector` and friends or smart pointers

Scopes / namespaces

Scope

Definition

Portion of the source code where a given name is valid

Typically :

- simple block of code, within {}
- function, class, namespace
- the global scope, i.e. translation unit (.cpp file + all includes)

Example

```
1 { int a;
2   { int b;
3   } // end of b scope
4 } // end of a scope
```

Scope and lifetime of variables

Variable life time

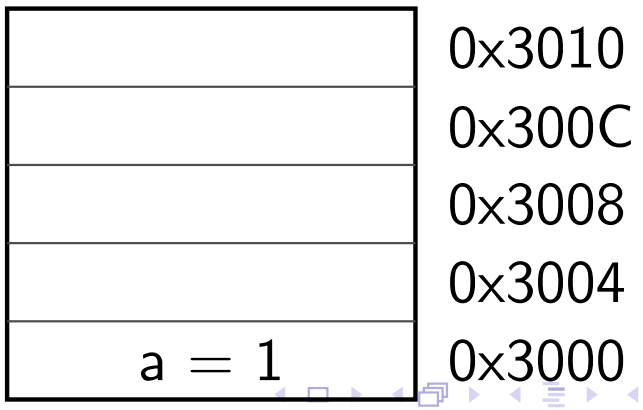
- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope (= they don't exist anymore outside it)

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

```
1  int a = 1;
2  {
3    int b[4];
4    b[0] = a;
5  }
6  // Doesn't compile here:
7  // b[1] = a + 1;
```

Memory layout



Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope (= they don't exist anymore outside it)

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

```
1  int a = 1;
2  {
3    int b[4];
4    b[0] = a;
5  }
6  // Doesn't compile here:
7  // b[1] = a + 1;
```

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = ?	0x3004
a = 1	0x3000

Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope (= they don't exist anymore outside it)

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

```
1  int a = 1;
2  {
3    int b[4];
4    b[0] = a;
5  }
6  // Doesn't compile here:
7  // b[1] = a + 1;
```

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = 1	0x3004
a = 1	0x3000

Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope (= they don't exist anymore outside it)

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

```
1  int a = 1;
2  {
3    int b[4];
4    b[0] = a;
5  }
6  // Doesn't compile here:
7  // b[1] = a + 1;
```

Memory layout

?	0x3010
?	0x300C
?	0x3008
1	0x3004
a = 1	0x3000

Namespaces

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

```
1  int a = 3;
2  namespace n {
3    int a = 4;
4    // not the same a!
5  }
6  namespace p {
7    int a = 5;
8  }
9  void f() {
10   n::a = 6;
11 }
```

```
11 ();
12  = 7;
13 uess:
14 hat is the value of the a's?
```

Hello World: also with namespaces!


```
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 }
```

Using namespace directives

Avoid "using namespace" directives

- Make all members of a namespace visible in current scope
- Risk of name clashes or ambiguities

```
1 using namespace std;
2 cout << "We can print now\n"; // uses std::cout
```

Never use in headers at global scope!

```
1 #include "PoorlyWritten.h" // using namespace std;
2 struct array { ... };
3 array a;  // Error: name clash with std::array
```

What to do instead

- Qualify names: `std::vector`, `std::cout`, ...
- Put things that belong together in the same namespace
- Use *using declarations* in local scopes: `using std::cout;`

Class and enum 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 = 78.5f;
9
10 Individual teacher = {
11   45, 67.0f
12 };
```

```
14 Individual *ptr = &student;
15 ptr->age = 25;
16 // same as: (*ptr).age = 25;
```

Note: other C-like structures exist (`union`, `enums`) but we won't cover them here.

References

References

References

- References allow for direct access to another object
- They can be used as shortcuts / better readability
- They can be declared `const` to allow only read access

Example:

```
1  int i = 2;
2  int &iref = i; // access to i
3  iref = 3;      // i is now 3
4
5  // const reference to a member:
6  struct A { int x; int y; } a;
7  const int &x = a.x; // direct read access to A's x
8  x = 4;              // doesn't compile
9  a.x = 4;            // fine
```

Pointers vs References

Specificities of reference

- Natural syntax
- Cannot be `nullptr`
- Must be assigned when defined, cannot be reassigned
- References to temporary objects must be `const`

Advantages of pointers

- Can be `nullptr`
- Can be initialized after declaration, can be reassigned

Pointers vs References

Specificities of reference

- Natural syntax
- Cannot be `nullptr`
- Must be assigned when defined, cannot be reassigned
- References to temporary objects must be `const`

Advantages of pointers

- Can be `nullptr`
- Can be initialized after declaration, can be reassigned

Good practice: References

- Prefer using references instead of pointers
- Mark references `const` to prevent modification

We will see an example in practice in the afternoon session

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 return
12 void log(char* msg) {
13   std::cout << msg;
14 }
15
16 // no parameter
17 void hello() {
18   std::cout << "Hello World";
19 }
```

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 return
12 void log(char* msg) {
13   std::cout << msg;
14 }
15
16 // no parameter
17 void hello() {
18   std::cout << "Hello World";
19 }
```

Functions and references to returned values

```
1  int result = square(2);
2  int & temp = square(2);          // Not allowed
3  int const & temp2 = square(2); // OK
```

Function default arguments

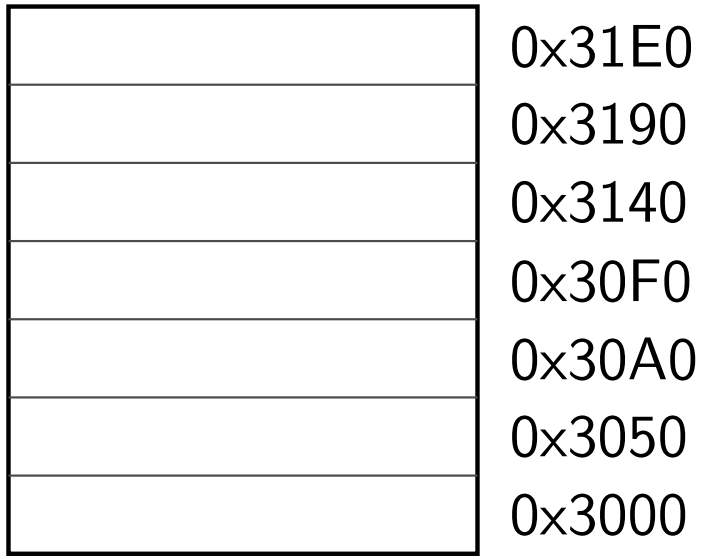

```
1  // must be the trailing
2  // argument
3  int add(int a,
4          int b = 2) {
5    return a + b;
6  }
7  // add(1) == 3
8  // add(3,4) == 7
9
```

```
11 // multiple default
12 // arguments are possible
13 int add(int a = 2,
14         int b = 2) {
15   return a + b;
16 }
17 // add() == 4
18 // add(3) == 5
```

Functions: parameters are passed by value

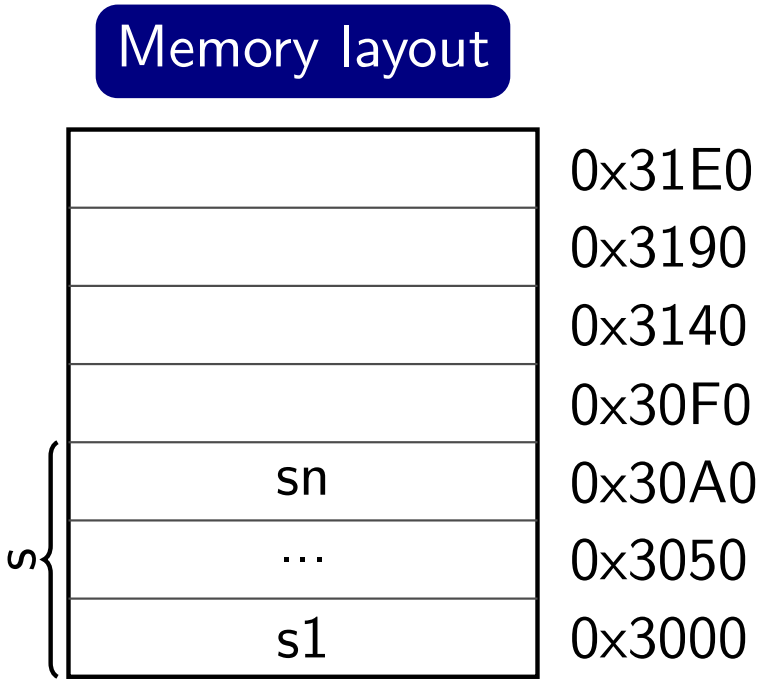

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6     ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12    ...
13 }
14 printRef(s); // no copy
```

Memory layout



Functions: parameters are passed by value

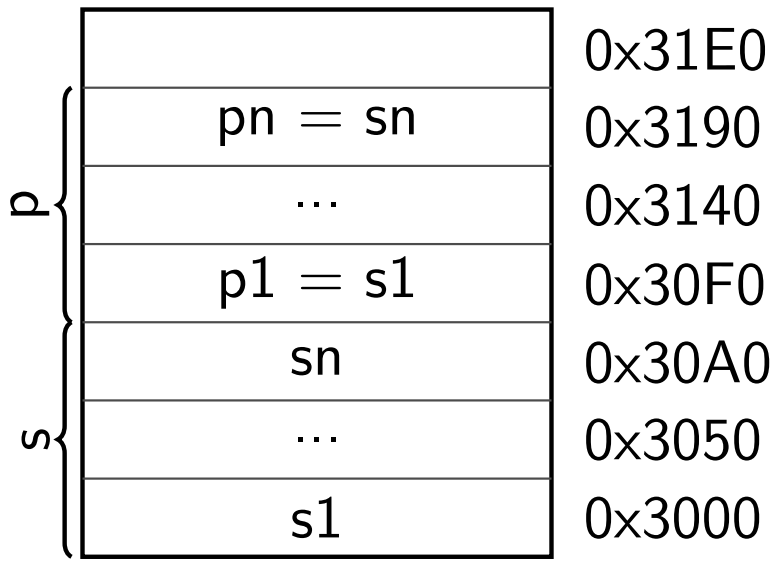

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6     ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12    ...
13 }
14 printRef(s); // no copy
```



Functions: parameters are passed by value

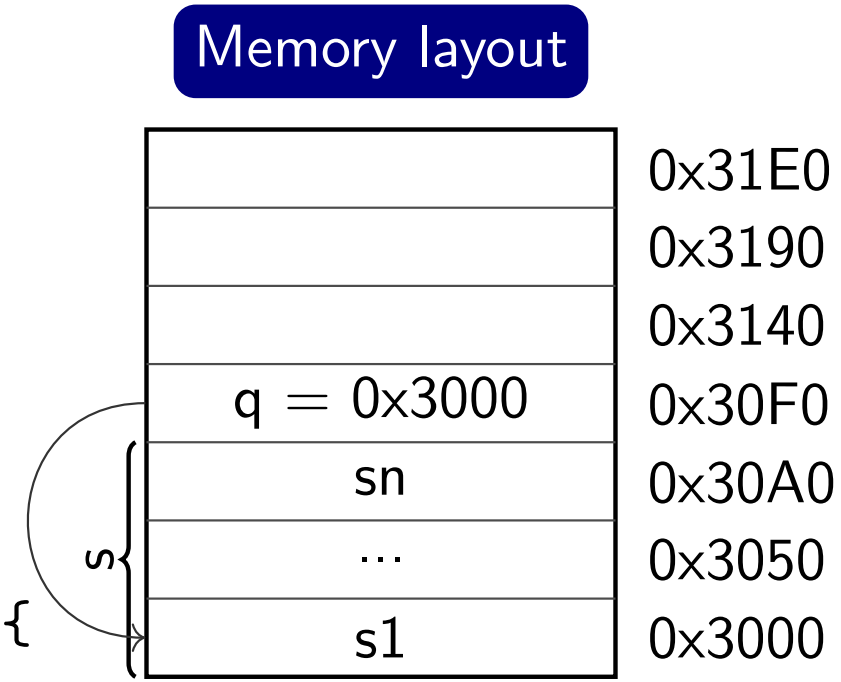

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6     ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12    ...
13 }
14 printRef(s); // no copy
```

Memory layout



Functions: parameters are passed by value

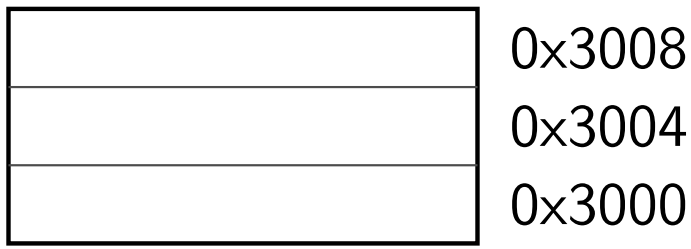

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6     ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12    ...
13 }
14 printRef(s); // no copy
```



Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

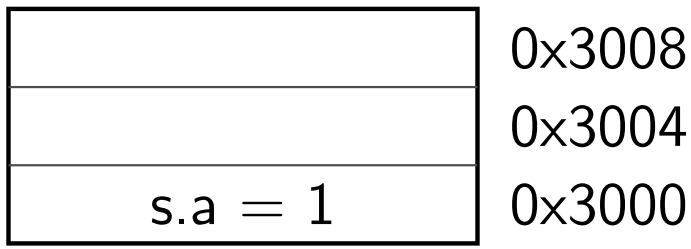
Memory layout



Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

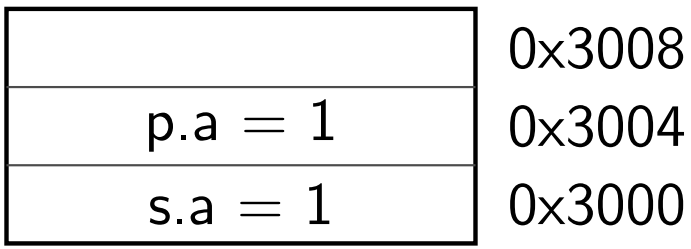
Memory layout



Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

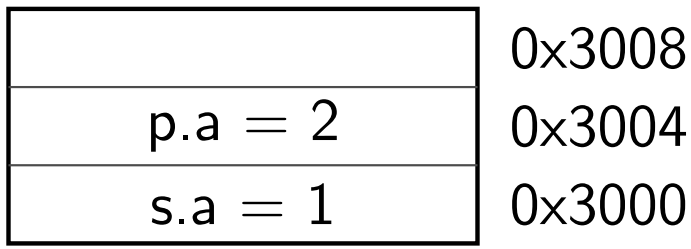
Memory layout



Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

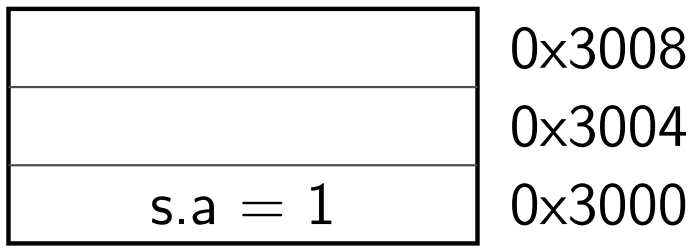
Memory layout



Functions: pass by value or reference?

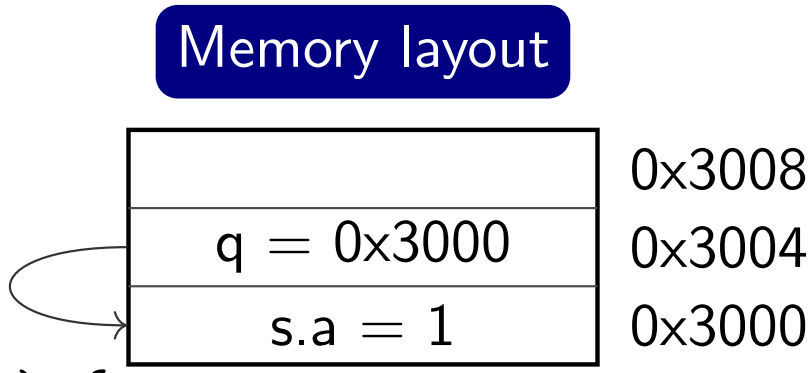

```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

Memory layout



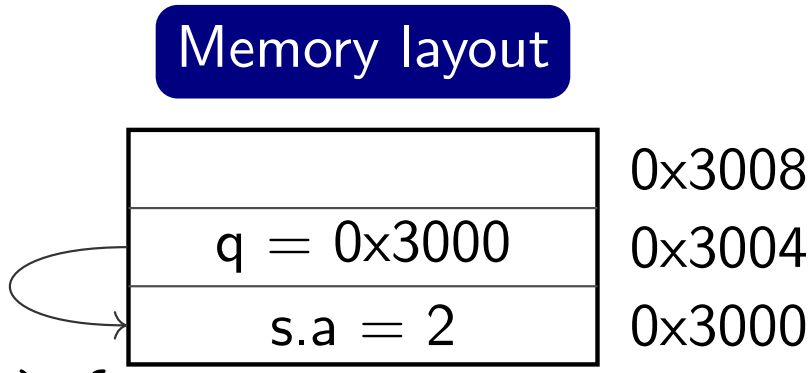
Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```



Functions: pass by value or reference?

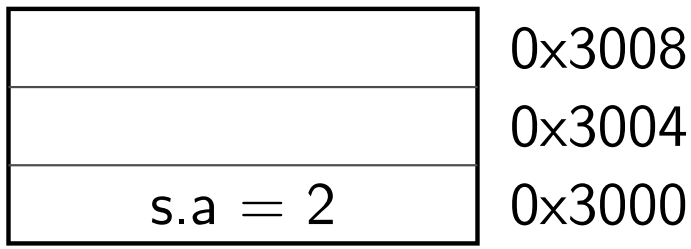

```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```



Functions: pass by value or reference?


```
1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5    p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11   q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

Memory layout



Pass by value, reference or pointer

Different ways to pass arguments to a function

- By default, arguments are passed by value (= copy)
  good for small types, e.g. numbers
- Use references for parameters to avoid copies
  good for large types, e.g. objects
- Use `const` for safety and readability whenever possible

Pass by value, reference or pointer

Different ways to pass arguments to a function

- By default, arguments are passed by value (= copy)
  good for small types, e.g. numbers
- Use references for parameters to avoid copies
  good for large types, e.g. objects
- Use `const` for safety and readability whenever possible

Syntax

```
1 struct T {...}; T a;
2 void fVal(T value);          fVal(a);   // by value
3 void fRef(const T &value);   fRef(a);   // by reference
4 void fPtr(const T *value);   fPtr(&a);  // by pointer
5 void fWrite(T &value);       fWrite(a); // non-const ref
```

Overloading

Overloading

- We can have multiple functions with the same name
  - Must have different parameter lists
  - A different return type alone is not allowed
  - Form a so-called "overload set"
- Default arguments can cause ambiguities

```
1  int sum(int b);                  // 1
2  int sum(int b, int c);           // 2, ok, overload
3  // float sum(int b, int c); // disallowed
4  sum(42); // calls 1
5  sum(42, 43); // calls 2
6  int sum(int b, int c, int d = 4); // 3, overload
7  sum(42, 43, 44); // calls 3
8  sum(42, 43);      // error: ambiguous, 2 or 3
```

Functions

Exercise: Functions

Familiarise yourself with pass by value / pass by reference.

- Go to `exercises/functions`
- Look at `functions.cpp`
- Compile it (`make`) and run the program (`./functions`)
- Work on the tasks that you find in `functions.cpp`

Functions: good practices

Good practice: Write readable functions

- Keep functions short
- Do one logical thing (single-responsibility principle)
- Use expressive names
- Document non-trivial functions

Example: Good

```
1  /// Count number of dilepton events in data.
2  /// \param d Dataset to search.
3  unsigned int countDileptons(Data &d) {
4    selectEventsWithMuons(d);
5    selectEventsWithElectrons(d);
6    return d.size();
7  }
```

Functions: good practices

Example: don't! Everything in one long function

```
unsigned int runJob() {
  // Step 1: data
  Data data;
  data.resize(123456);
  data.fill(...);

  // Step 2: muons
  for (....) {
    if (...) {
      data.erase(...);
    }
  }
  // Step 3: electrons
  for (....) {
    if (...) {
      data.erase(...);
    }
  }

  // Step 4: dileptons
  int counter = 0;
  for (....) {
    if (...) {
      counter++;
    }
  }

  return counter;
}
```

Operators

Operators(1)

Binary and Assignment Operators

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

Operators(1)

Binary and Assignment Operators

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

Increment / Decrement Operators

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

Operators(1)

Binary and Assignment Operators

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

Increment / Decrement Operators

Use wisely

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

Operators(2)

Logical Operators

```
1  bool a = true;
2  bool b = false;
3  bool c = a && b;     // false
4  bool d = a || b;     // true
5  bool e = !d;         // false
```

Operators(3)

Comparison Operators

```
1  bool a = (3 == 3);  // true
2  bool b = (3 != 3);  // false
3  bool c = (4 <  4);  // false
4  bool d = (4 <= 4);  // true
5  bool e = (4 >  4);  // false
6  bool f = (4 >= 4);  // true
7  auto g = (5 <=> 5); // C++20 (later)
```

Operators(3)

Comparison Operators

```
1  bool a = (3 == 3);  // true
2  bool b = (3 != 3);  // false
3  bool c = (4 <  4);  // false
4  bool d = (4 <= 4);  // true
5  bool e = (4 >  4);  // false
6  bool f = (4 >= 4);  // true
7  auto g = (5 <=> 5); // C++20 (later)
```

Precedences

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

Details can be found on cppreference

Operators(3)

Comparison Operators

```
1  bool a = (3 == 3);  // true
2  bool b = (3 != 3);  // false
3  bool c = (4 <  4);  // false
4  bool d = (4 <= 4);  // true
5  bool e = (4 >  4);  // false
6  bool f = (4 >= 4);  // true
7  auto g = (5 <=> 5); // C++20 (later)
```

Precedences

Avoid

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

Details can be found on cppreference

Operators(3)

Comparison Operators

```
1  bool a = (3 == 3);  // true
2  bool b = (3 != 3);  // false
3  bool c = (4 <  4);  // false
4  bool d = (4 <= 4);  // true
5  bool e = (4 >  4);  // false
6  bool f = (4 >= 4);  // true
7  auto g = (5 <=> 5); // C++20 (later)
```

Precedences

Avoid - use parentheses

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

Details can be found on cppreference

Control structures

2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Auto keyword
- Headers and interfaces

Control structures: if

if syntax

```
1  if (condition1) {
2    Statement1; Statement2;
3  } else if (condition2)
4    OnlyOneStatement;
5  else {
6    Statement3;
7    Statement4;
8  }
```

- The `else` and `else if` clauses are optional
- The `else if` clause can be repeated
- Braces are optional if there is a single statement

Control structures: if

Practical example

```
1  int collatz(int a) {
2    if (a <= 0) {
3      std::cout << "not supported\n";
4      return 0;
5    } else if (a == 1) {
6      return 1;
7    } else if (a%2 == 0) {
8      return collatz(a/2);
9    } else {
10     return collatz(3*a+1);
11   }
12 }
```

Control structures: conditional operator

Syntax

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

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Control structures: conditional operator

Syntax

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

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Practical example

```
1   const int charge = isLepton ? -1 : 0;
```

Control structures: conditional operator

Syntax

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

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Practical example

```
1  const int charge = isLepton ? -1 : 0;
```

Do not abuse it

```
1  int collatz(int a) {
2    return a==1 ? 1 : collatz(a%2==0 ? a/2 : 3*a+1);
3  }
```

- Explicit `if`s are generally easier to read
- Use the ternary operator with short conditions and expressions
- Avoid nesting

Control structures: switch

Syntax

```
1  switch(identifier) {
2    case c1 : statements1; break;
3    case c2 : statements2; break;
4    case c3 : statements3; break;
5    ...
6    default : statementsn; break;
7  }
```

- The `break` statement is not mandatory but...
- Cases are entry points, not independent pieces
- Execution "falls through" to the next case without a `break`!
- The `default` case may be omitted

Control structures: switch

Syntax

```
1  switch(identifier) {
2    case c1 : statements1; break;
3    case c2 : statements2; break;
4    case c3 : statements3; break;
5    ...
6    default : statementsn; break;
7  }
```

- The `break` statement is not mandatory but...
- Cases are entry points, not independent pieces
- Execution "falls through" to the next case without a `break`!
- The `default` case may be omitted

Use break

Avoid `switch` statements with fall-through cases

Control structures: switch

Practical example

```
1  enum class Lang { French, German, English, Other };
2  Lang language = ...;
3  switch (language) {
4    case Lang::French:
5      std::cout << "Bonjour";
6      break;
7    case Lang::German:
8      std::cout << "Guten Tag";
9      break;
10   case Lang::English:
11     std::cout << "Good morning";
12     break;
13   default:
14     std::cout << "I do not speak your language";
15 }
```

[[fallthrough]] attribute

New compiler warning

Since C++ 17, compilers are encouraged to warn on fall-through

C++ 17

```
1   switch (c) {
2     case 'a':
3       f();     // Warning emitted
4     case 'b': // Warning probably suppressed
5     case 'c':
6       g();
7       [[fallthrough]]; // Warning suppressed
8     case 'd':
9       h();
10  }
```

Init-statements for if and switch

C++ 17

Purpose

Allows to limit variable scope in `if` and `switch` statements

C++ 17

```
1  if (Value val = GetValue(); condition(val)) {
2    f(val); // ok
3  } else
4    g(val); // ok
5  h(val);   // error, no `val` in scope here
```

Init-statements for if and switch

C++ 17

Purpose

Allows to limit variable scope in `if` and `switch` statements

C++ 17

```
1  if (Value val = GetValue(); condition(val)) {
2    f(val); // ok
3  } else
4    g(val); // ok
5  h(val);   // error, no `val` in scope here
```

C++ 98

Don't confuse with a variable declaration as condition:

```
7  if (Value* val = GetValuePtr())
8    f(*val);
```

Control structures: for loop

for loop syntax

```
1 for(initializations; condition; increments) {
2   statements;
3 }
```

- Initializations and increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Control structures: for loop

for loop syntax

```
1  for(initializations; condition; increments) {
2    statements;
3  }
```

- Initializations and increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Practical example

```
4  for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
5    std::cout << i << "^2 is " << j << '\n';
6  }
```

Control structures: for loop

for loop syntax

```
1 for(initializations; condition; increments) {
2   statements;
3 }
```

- Initializations and increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Practical example

```
4 for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
5   std::cout << i << "^2 is " << j << '\n';
6 }
```

Good practice: Don't abuse the `for` syntax

- The `for` loop head should fit in 1-3 lines

Range-based loops

Reason of being

- Simplifies loops over "ranges" tremendously
- Especially with STL containers and ranges

Syntax

```
1  for ( type iteration_variable : range ) {
2    // body using iteration_variable
3  }
```

Example code

```
4  int v[4] = {1,2,3,4};
5  int sum = 0;
6  for (int a : v) { sum += a; }
```

Init-statements for range-based loops

C++ 20

Purpose

Allows to limit variable scope in range-based loops

C++ 17

```
1  std::array data = {"hello", ",", "world"};
2  std::size_t i = 0;
3  for (auto& d : data) {
4    std::cout << i++ << ' ' << d << '\n';
5  }
```

C++ 20

```
6  for (std::size_t i = 0; auto const & d : data) {
7    std::cout << i++ << ' ' << d << '\n';
8  }
```

Control structures: while loop

while loop syntax

```
1  while(condition) {
2    statements;
3  }
4
5  do {
6    statements;
7  } while(condition);
```

- Braces are optional if the body is a single statement

Control structures: while loop

while loop syntax

```
1  while(condition) {
2    statements;
3  }
4
5  do {
6    statements;
7  } while(condition);
```

- Braces are optional if the body is a single statement

Bad example

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

Control structures: jump statements

break	Exits the loop and continues after it
continue	Goes immediately to next loop iteration
return	Exits the current function
goto	Can jump anywhere inside a function, avoid!

Control structures: jump statements

break	Exits the loop and continues after it
continue	Goes immediately to next loop iteration
return	Exits the current function
goto	Can jump anywhere inside a function, avoid!

Bad example

```
1  while (1) {
2    if (n == 1) break;
3    if (0 == n%2) {
4      std::cout << n << '\n';
5      n /= 2;
6      continue;
7    }
8    n = 3 * n + 1;
9  }
```

Control structures

Exercise: Control structures

Familiarise yourself with different kinds of control structures.
Re-implement them in different ways.

- Go to `exercises/control`
- Look at `control.cpp`
- Compile it (`make`) and run the program (`./control`)
- Work on the tasks that you find in `README.md`

Auto keyword

Auto keyword

Reason of being

- Many type declarations are redundant
- They are often a source for compiler warnings and errors
- Using auto prevents unwanted/unnecessary type conversions

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

Auto keyword

Reason of being

- Many type declarations are redundant
- They are often a source for compiler warnings and errors
- Using auto prevents unwanted/unnecessary type conversions

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

Practical usage

```
1  std::vector<int> v;
2  auto a = v[3];
3  const auto b = v.size(); // std::size_t
4  int sum{0};
5  for (auto n : v) { sum += n; }
```

Loops, references, auto

Exercise: Loops, references, auto

Familiarise yourself with range-based for loops and references

- Go to `exercises/loopsRefsAuto`
- Look at `loopsRefsAuto.cpp`
- Compile it (`make`) and run the program (`./loopsRefsAuto`)
- Work on the tasks that you find in `loopsRefsAuto.cpp`

Headers and interfaces

Headers and interfaces

Interface

Set of declarations defining some functionality

- Put in a so-called "header file"
- The implementation exists somewhere else

Header: hello.hpp

```
void printHello();
```

Usage: myfile.cpp

```
1 #include "hello.hpp"
2 int main() {
3   printHello();
4 }
```

Preprocessor


```
1  // file inclusion
2  #include "hello.hpp"
3  // macro constants and function-style macros
4  #define MY_GOLDEN_NUMBER 1746
5  #define CHECK_GOLDEN(x) if ((x) != MY_GOLDEN_NUMBER) \
6    std::cerr << #x " was not the golden number\n";
7  // compile time or platform specific configuration
8  #if defined(USE64BITS) || defined(__GNUG__)
9    using myint = std::uint64_t;
10 #elif
11   using myint = std::uint32_t;
12 #endif
```

Preprocessor


```
1  // file inclusion
2  #include "hello.hpp"
3  // macro constants and function-style macros
4  #define MY_GOLDEN_NUMBER 1746
5  #define CHECK_GOLDEN(x) if ((x) != MY_GOLDEN_NUMBER) \
6    std::cerr << #x " was not the golden number\n";
7  // compile time or platform specific configuration
8  #if defined(USE64BITS) || defined(__GNUG__)
9    using myint = std::uint64_t;
10 #elif
11   using myint = std::uint32_t;
12 #endif
```

Good practice: Use preprocessor only in very restricted cases

- Conditional inclusion of headers
- Customization for specific compilers/platforms

Books

A Tour of C++, Third Edition
Bjarne Stroustrup, Addison-Wesley, Sep 2022
ISBN-13: 978-0136816485

Effective Modern C++
Scott Meyers, O'Reilly Media, Nov 2014
ISBN-13: 978-1-491-90399-5

C++ Templates - The Complete Guide, 2nd Edition
David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor
ISBN-13: 978-0-321-71412-1

C++ Best Practices, 2nd Edition
Jason Turner
https://leanpub.com/cppbestpractices

Clean Architecture
Robert C. Martin, Pearson, Sep 2017
ISBN-13: 978-0-13-449416-6

The Art of UNIX Programming
Eric S. Raymond, Addison-Wesley, Sep 2002
ISBN-13: 978-0131429017

Introduction to Algorithms, 4th Edition
T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Apr 2022
ISBN-13: 978-0262046305