Modern C++ Programming

3. Basic Concepts II

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0

Agenda

- **Memory Management: Heap and Stack**
    - Heap allocation and memory leak
    - Stack memory
    - Stack 2D allocation
    - Initialization
    - Data/Bss memory segment
- **Storage Class Specifiers**
- **Pointers and References**
    - Pointers
    - Void Pointer
    - Address-of Operator
    - Pointer Arithmetic
    - Reference
- **sizeof Operator**

- **Other Keywords**
    - `const`, `constexpr`
    - `volatile`
    - `using`, `decltype`
- **Explicit Type Conversion**
- **Declaration and Definition**
- **Functions**
    - Call-by-value/pointer/reference
    - `inline`
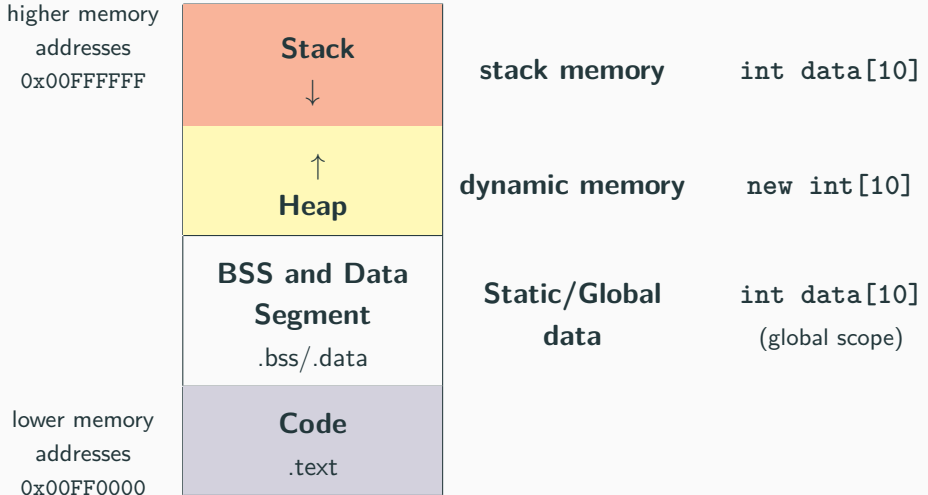   - Default parameters
    - Overloading
- **Unions and Bitfields**
- **Preprocessing**
    - Macro
    - Pragma

Memory Management: Heap and Stack

higher memory addresses 0x00FFFFFF	**Stack** ↓	**stack memory** `int data[10]`
	↑ **Heap**	**dynamic memory** `new int[10]`
	BSS and Data Segment .bss/.data	**Static/Global data** `int data[10]` (global scope)
lower memory addresses 0x00FF0000	**Code** .text	

Dynamic allocation on **heap** in C:

```c
int*  value = (int*) malloc(sizeof(int));       // allocate one int
int* array1 = (int*) malloc(10 * sizeof(int)); // allocate ten int
int* array2 = (int*) calloc(10 * sizeof(int)); // allocate ten int
                                                // and set to 0
free(array1); // free is the C method applied to deallocate  memory
```

The C++ operator `new` is applied to allocate a variable/object/etc. in heap memory:

```cpp
int* value  = new int;       // allocate one int
int* array1 = new int[10];   // allocate ten int
int* array2 = new int[10](); // allocate ten int and set to 0
delete value;    // delete is the C++ operator applied to deallocate
delete[] array1;
```

Note. For an array of objects:

- `delete/delete[]` recursively invoke destructor (see next lectures)

- `delete/delete[]` applied to `nullptr` has no effect (safe)

3/57

Definition (Memory Leak)

Dynamically allocated entity in heap memory is no longer used by the program, but still maintained overall its execution

An object is stored in memory but cannot be accessed by the running code

Problems:

- Illegal memory accesses
- Undefined values
- Additional memory consumption

Fundamental rules:

- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`

Common errors:

Pointer loss and wild pointer:

```
int main() {
  int* array = new int[10];
  array      = nullptr; // memory leak!! array cannot be deallocated!
  new int[10];          // legal code but memory leak!!
  int* ptr;             // wild pointer: Where will this pointer points?
  ...
}
```

Double delete:

```
int* array = new int[10];
delete[] array; // ok -> array: dangling pointer
delete[] array; // double free or corruption!!
// program aborted
```

Unless it is allocated in heap memory (i.e. `new`), then it is either in stack memory or CPU registers

Important:

Every object which resides in the stack is not valid outside the current scope!!

```cpp
int* wrongFunction() {
    int A[3] = {1, 2, 3};
    return A;
}

int main() {
   int* ptr = wrongFunction();
   cout << ptr[0]; // Illegal memory access!!
}
```

Important:

The organization of stack memory enables much higher performance. On the other hand, this memory space is limited!!

It is $\approx 8MB$ on linux by default.

`sizeof` and memory allocation:

```cpp
int A[10];
int* B = new int[10];
cout << sizeof(A); // print sizeof(int) * 10 = 40
cout << sizeof(B); // print sizeof(int*) = 8  (64-bit)
```

```cpp
int* ptr[10];    // array of ten integer pointers
                 // read as (int*) ptr[10]
int (*ptr)[10];  // pointer to an array of ten integer
                 // equal to:
                 //     int a[10];
                 //     int* ptr = a;
```

Easy on stack:

```
int A[3][4];
```

Dynamic Memory 2D allocation/free:

```
int* A = new int*[3];
for (int i = 0; i < 3; i++)
    A[i] = new int[4];

for (int i = 0; i < 3; i++)
    delete[] A[i];
delete[] A;
```

Dynamic memory 2D allocation/free C++11:

```
auto A = new int[3][4];       // allocate 3 objects of size int[4]
int n = 3;                    // dynamic value
auto B = new int[n][4];       // ok
// auto C = new int[n][n];    // compile error
delete[] A;                   // same for B, C
```

Memory Management (Stack Array Initialization)

One dimension:

```cpp
int  A[3] = {1, 2, 3}; // explicit size
int  B[]  = {1, 2, 3}; // implicit size
char C[]  = "abcd";    // implicit size
int  C[3] = {1, 2};    // C[2] has undefined value
int  D[4] = {0}; // all values of D are initialized to 0
int  E[3] = {};  // all values of E are initialized to 0 (C++11)
```

Two dimensions:

```cpp
// int F[][]   = ...; // compile error
// int G[2][]  = ...; // compile error
int G[][2]  = { {1,2}, {3,4}, {5,6} }; // ok
int H[2][2] = { 1, 2, 3, 4 }; // ok
```

Memory Management (Data and BSS Segment)

```
int data[] = {1, 2, 3, 4}; // data segment memory
int big_data[1000000] = {}; // bss segment memory (zero-initialized)

int main() {
    int A[] = {1, 2, 3}; // stack memory
}
```

Data/Bss (Block Started by Symbol) are larger than stack memory
(max $\approx$ 1GB in general) but slower

Default Initialization

Default Initialization

Rules:

- An object with **dynamic** (heap) storage duration has <u>indeterminate</u> value

- An object whose initializer is an **empty set of parentheses** is <u>zero</u> or <u>default</u> initialized

- Objects with **static** or **thread storage** duration are <u>zero</u> or <u>default</u> initialized

Initialization

```cpp
int a1;                    // indeterminate
int* a2 = new int;         // indeterminate
int* a3 = new int();       // indeterminate
int* a4 = new int(4);      // allocate a single value equal to 4!!

int* b1 = new int[4]();    // allocate 4 elements zero-initiliazed
int* b2 = new int[4]{};    // indeterminate
int* b3 = new int[4]{1, 2}; // set first, second, indeterminate
                           // other values

int c1(4);                 // c1 = 4;
int c2 = int();            // zero-initiliazed
int c3(0);                 // zero-initiliazed
int c4 { 0 };              // zero-initiliazed
int c5 = { 0 };            // zero-initiliazed

static int d1;             // zero-initiliazed
thread_local int d2;       // zero-initiliazed
// int d3();               // d3 is a function
```

Pointers and References

Pointers and Pointer Dereferencing

Definition (Pointer)

A pointer is a value referring to a location in memory

Definition (Pointer Dereferencing)

Pointer dereferencing means obtaining the value stored in at the location refereed to the pointer

```cpp
int* ptr1 = new int;
*ptr1    = 4;      // deferencing (assignment)
int a     = *ptr1; // deferencing (get value)
```

Common errors:

```cpp
int *ptr1, ptr2; // one pointer and one integer!!
int *ptr1, *ptr2; // ok two pointers
```

void Pointer (Generic Pointer)

Instead of declaring different types of pointer variable it is possible to declare single pointer variable which can act as any pointer types

- A `void*` can be assigned to another `void*`
- `void*` can be <u>compared</u> for equality and inequality
- A `void*` can be <u>explicitly</u> converted to another type.
- Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors

```cpp
cout << (sizeof(void*) == sizeof(int*));  // print true

int array[] = { 2, 3, 4 };
void* ptr = array;
cout << *array;         // print 2
// cout << *ptr;         // compile error
cout << *((int*) ptr); // print 2
// void* ptr2 = ptr + 2; // compile error
```

Address-of operator &

The **address-of operator** (&) returns the address of the variable

```cpp
int  a = 3;
int* b = &a; // address-of operator,
             // 'b' is equal to the address of 'a'
a++;
cout << *b;  // print 4;

int array[4]; // &array is a pointer to a array of size 4
int size1 = (&array)[1] - array;
int size2 = *(&array + 1) - array;
cout << size1;  // print 4
cout << size2;  // print 4
```

To not confuse with **Reference syntax:** `T& var = ...`

$1 + 1 \neq 2$: Pointer Arithmetic

Pointer syntax:

`ptr[i]` is equal to `*(ptr + i)`

Pointer arithmetic rule:

$address(ptr + i) = address(ptr) + (sizeof(T) * i)$

where `T` is the type of elements pointed by `ptr`

Example:

```
int array[4] = {1, 2, 3, 4};
cout << array[1];      // print 2
cout << *(array + 1);  // print 2
cout << array;         // print 0xFFFAFFF2
cout << array + 1;     // print 0xFFFAFFF6!!
```

`char arr[3] = "abc"`

value	address	
'a'	0x0	←arr[0]
'b'	0x1	←arr[1]
'c'	0x2	←arr[2]

`int arr[3] = {4,5,6}`

value	address	
	0x0	←arr[0]
4	0x1	
	0x2	
	0x3	
5	4	←arr[1]
	0x5	
	0x6	
	0x7	
	0x8	←arr[2]

Reference

Definition (Reference)

A variable **reference** is an **alias**, namely another name for an already existing variable. Both variable and variable reference can be applied to refer the value of the variable

References are safer then pointers:

- References **cannot have NULL** value. You must always be able to assume that a reference is connected to a legitimate piece of storage.

- References **cannot be changed**. Once a reference is initialized to an object, it cannot be changed to refer to another object.
  (Pointers can be pointed to another object at any time)

- References must be **initialized** when it is created.
  (Pointers can be initialized at any time)

- A pointer has its own memory address and size on the stack, reference shares the **same memory address** (with the original variable) but also takes up some space on the stack.

Reference (Examples)

Reference syntax: `T& var = ...`

```cpp
//int& d;    // reference. compile error!! no initilization
int   c = 2;
int& e = c; // reference. ok valid initialization
e++;        // increment `e'
cout << c;  // print 3
```

```cpp
int* b = &a; // pointer
int* c = &a; // pointer
b++;        // change the value of the pointer `b'
*c++;       // change the value `a'

int& c = a; // reference
c++;        // change the value `a'
```

References (Function Arguments)

References vs. pointer arguments:

```cpp
void f(int* value) {} // value may be a nullptr
void g(int& value) {} // value is never a nullptr

int a = 3;
f(&a);  // ok
g(a);   // ok
//g(3); // compile error "3" is not a reference of something
```

References can be use to indicate fixed size arrays:

```cpp
f(int (&array)[3]) {} // accepts only arrays of size 3

int A[3], B[4];
int* C = A;

f(A); // ok
f(B); // compile error!! B has size 4
f(C); // compile error!! C is a pointer
```

References (Arrays)

```cpp
int A[4];
int (&B)[4] = A;      // ok
int C[10][3];
int (&D)[10][3] = C; // ok

auto c = new int[3][4]; // type is int (*)[4]
// read as "pointer to arrays of 4 int"
// int (&d)[3][4] = c;  // compile error
// int (*e)[3]    = c;  // complier error
int (*f)[4] = c;        // ok
```

Reference:
[1] www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_
PointerReference.html

References and `struct`

- The dot (`.`) operator is applied to local objects and references
- The arrow operator (`->`) is used with a pointer to an object

```cpp
struct A {
    int x = 3;
};

A obj;

A* p = &obj; // pointer
p->x; // arrow syntax

A& ref = obj; // reference
std::cout << obj.x; // dot syntax
std::cout << ref.x; // dot syntax
```

`sizeof` Operator

The **`sizeof`** is a compile-time operator that determines the size, in bytes, of a variable or data type

- `sizeof` returns a value of type size_t

- `sizeof` (incomplete type) produces compile error

- `sizeof` (bitfield) produces compile error

- `sizeof` (anything) never returns 0 except for array of size 0

- `sizeof(char)` always returns 1

- When applied to structures it also takes into account padding

- When applied to a reference, the result is the size of the referenced type

```
sizeof(int);    // 4
sizeof(int*);   // 8 in a 64-bit OS
sizeof(void*)   // 8 in a 64-bit OS
sizeof(size_t)  // 8 in a 64-bit OS

char  a;
char& b = a;
sizeof(&a);     // 8 in a 64-bit OS (pointer)
sizeof(b);      // 1 sizeof(char)

struct A {};
sizeof(A);      // 1 : sizeof never return 0

A array1[10];
sizeof(array1); // 1 : array of empty structures

int array2[0];
sizeof(array2); // 0
```

```
struct B {
    int  x;
    char y;
};

struct C : B { // C extends B
    short z;
};

sizeof(B);     // 8 : 4 + 1 (+ 3) (padding)
sizeof(C);     // 12 : sizeof(B) + 2 (+ 2) (padding)

int array[4]
sizeof(array)  // 16: 4 elements of 4 bytes
sizeof(array) / sizeof(int);   // 4 elements
```

Other C++ Keywords

> **Definition (const keyword)**
>
> It indicates objects never changing value after their initialization
> (they must be initialized when declared)

Evaluated at compile-time

```
int size = 3;
int A[size] = {1, 2, 3}; // Technically possible (size is dynamic)
                         // But NOT approved by the C++ standard.

const int SIZE = 3;
// SIZE = 4;             // compile error!!
int B[SIZE] = {1, 2, 3}; // ok

const int size2 = size;
int B[size2] = {1, 2, 3}; // BAD programming!! size is not const
// (some compilers allow variable size stack array -> dangerous!!)
```

Constness rules:

- `int* → const int*`

- `const int* ↛ int*`

```
int g(const int* array) { // the values of array cannot be
    ...                    // modified
}
```

```
void f(const int) {  // pass-by-value combined with 'const'
    ...              // note: it is not useful because the value
}                    //       is copied
```

- `int*` pointer to int
    - The value of the pointer <u>can</u> be modified
    - The elements refereed by the pointer <u>can</u> be modified

- `const int*` pointer to const int. Read as `(const int)*`
    - The value of the pointer <u>can</u> be modified
    - The elements refereed by the pointer <u>cannot</u> be modified

- `int *const` const pointer to int
    - The value of the pointer <u>cannot</u> be modified
    - The elements refereed by the pointer <u>can</u> be modified

- `const int *const` const pointer to const int
    - The value of the pointer <u>cannot</u> be modified
    - The elements refereed by the pointer <u>cannot</u> be modified

Note: `const int *` is equal to `int const*`

Tip: pointer types should be read from right to left

constexpr keyword

> **Definition (constexpr)**
>
> C++11 guarantees compile-time evaluation of an expression as
> long as **all** its arguments are constant

- `const` guarantees the value of a variable to be fixed overall the execution of the program
- `constexpr` tells the compiler that the expression results is at compile-time. **constexpr** value implies **const** for variables
- C++11: `constexpr` must contain exactly one **return** statement and it must not contain loops or switch
- C++14: `constexpr` has no restrictions

```cpp
constexpr int square(int value) {
    return value * value;
}
const int v1 = square(3); // compile-time evaluation
int a = 3;
const int v2 = square(a); // run-time evaluation
```

Type Alias Keyword (`using` and `decltype`)

- In C++11, the `using` keyword has the same semantics of `typedef` specifier (alias-declaration), but with better syntax

```cpp
typedef int distance_t;    // equal to:
using distance_t = int;
```

- In C++11, `decltype` captures the type of an object or an expression

```cpp
int a = 3;
delctype(a) b = 5;           // 'b' is int
delctype(2.0f) c = 3.0f;     // 'c' is float
delctype(a + 2.0f) d = 3.0f; // 'd' is float
delctype(f(a)) e = ...;      // 'e' depends on f(a)
```

Explicit Type Conversion

Old style cast `(type) value`

C++11 cast:

- `static_cast` does compile-time, not run-time checking of the types involved. In many situations, this can make it the safest type of cast, as it provides the least room for accidental/unsafe conversions between various types.

- `reinterpret_cast`
  `reinterpret_cast<T*>(v)` equal to `(T*) v`
  `reinterpret_cast<T&>(v)` equal to `*((T*) &v)`

- `const_cast` may be used to cast away (remove) constness or volatility.

Static cast vs. old style cast:

```cpp
char a[] = {1, 2, 3, 4};
int* b = (int*) a; // ok
cout << b[0];      // print 67305985 not 1!!
int* c = static_cast<int*>(a); // compile error!! unsafe conversion
```

Const cast:

```cpp
const int a = 5;
const_cast<int>(a) = 3; // ok
```

Reinterpret cast: (bit-level conversion)

```cpp
float b = 3.0f;
// bit representation of b: 01000000010000000000000000000000
int c = reinterpret_cast<int&>(b);
// bit representation of c: 01000000010000000000000000000000
int a[3][4];   // array reshaping example
int (&b)[2][6] = reinterpret_cast<int (&)[2][6]>(a);
int (*c)[6]    = reinterpret_cast<int (*)[6]>(a);
```

Narrowing Conversion

C++11 provides protection against **narrowing**, i.e. it against assigning a numeric value to a numeric type not capable of holding that value

```cpp
int main() {
    int a1 = 36.6;     // ok
//  int a2 = { 36.6 }; // compile error

    float b1 = 36.6;   // ok
//  float b2 = 36.6;   // compile error

    char c1 = 512;     // ok
//  char c2 = { 512 }; // compile error
}
```

Declaration and Definition

Declaration/Definition

Definition (declaration/prototype)

A **declaration** (or prototype) of an entity is an identifier describing its type

A declaration is what the compiler and the linker needs to accept references to that identifier

Definition (definition/implementation)

An entity **definition** is the implementation of a declaration

Declaration/Definition (incomplete type)

A declaration without a concrete implementation is an incomplete type (as void)

C++ Entities (class, functions, etc.) can be declared multiple times (with the same signature)

```cpp
struct A;   // declaration 1
struct A;   // declaration 2 (ok)

struct B {  // declaration and definition
    int b;
//  A x;    // incomplete type
};

struct A {  // definition
    char c;
}
```

Functions

Definition (Signature)

Type signature defines the inputs and outputs for a function. A type signature includes the number of arguments, the types of arguments and the order of the arguments contained by a function

Definition (Function Parameter [formal])

A parameter is the variable which is part of the method's signature

Definition (Function Argument [actual])

An argument is the actual value (instance) of the variable that gets passed to the function

```cpp
int f(int a, char* b);    // function declaration
                          // signature:  (int, char*)
                          // parameters: int a, char* b

int f(int a, char*) {     // function definition
    ...                   // b can be omitted if not used
}

// char f(int a, char* b); // compile error!! same signature

// int f(const int a, char* b); // invalid declaration!
                                 // const int == int
int f(int a, const char* b);    // ok

int main() {
    f(3, "abc"); // function arguments: 3, "abc"
                 // error: "f" call f(int, const char*)
}                // which is not defined
```

Call-by-Value

Definition (Call-by-value)

Values are copies and assigned to input arguments of the method

Advantages:

- Changes made to the parameter inside the function have no effect on the argument

Disadvantages:

- Performance penalty if the copied arguments are large (e.g. a structure with a large array)

When to use:

- Small objects that do not need to be change

When not to use:

- Arrays: arrays decay into pointers, when passed to a function
- Large objects

Call-by-Pointer

Definition (Call-by-pointer)

The addresses of variables are copies and assigned to input arguments of the method

Inside the function, the address is used to access the actual argument used in the call

Advantages:

- Allows a function to change the value of the argument
- Copy of the argument is not made (fast)

Disadvantages:

- The argument may be `null`
- Dereferencing a pointer is slower than accessing a value directly

When to use:

- When passing built-in arrays (use `const` if read-only)

When not to use:

- Small objects

Call-by-Reference

Definition (Call-by-reference)

The reference of variables are copies and assigned to input arguments of the method

Advantages:

- Allows a function to change the value of the argument

- Copy of the argument is not made (fast)

- References must be initialized (no null pointer)

- Avoid implicit conversion

When to use:

- Structs or Classes (use `const` if read-only)

Examples

```cpp
struct MyStruct {
    int field;
};

void f1(int a);        // call by value
void f2(int& a);       // call by reference
void f3(const int& a); // call by const reference
void f4(MyStruct& a);  // requires a.field to access

void f5(int* a);       // call by pointer
void f6(const int* a); // call by const pointer
void f7(MyStruct* a);  // requires a->field to access
//----------------------------------------------------
char c = 'a';
f1('a');    // ok, pass-by-value
// f2('a'); // compile error!! pass-by-reference
```

`inline` Function Declaration

Definition (inline)

`inline` specifier is a hint for the compiler. The code of the function can be copied where it is called (inlining)

```
inline void f(int a) { ... }
```

- It is just a hint. The compiler can ignore the hint (`inline` increases the compiler heuristic threshold)

- The compiled code is larger because the `inline` function is expanded in-place for every function call

GCC/Clang extensions allow to force inline functions:

```
inline __attribute__((always_inline)) void f(int a) { ... }
```

Function Default Parameters

Definition (Default/Optional parameter)

A **default parameter** is a function parameter that has a default value provided to it.
If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value

- All default parameters must be the rightmost parameters
- Default parameters can only be declared once
- Default parameters can improve compile time because they avoid defining other overloaded functions

```cpp
void f(int a, int b = 20);
// void g(int a = 10, int b); // compile error

void f(int a, int b) { ... }  // default value of "b" already set

int main() {
    f(5);  // b is 20
}
```

Function Overloading (Ambiguous Matches)

An **overloaded declaration** is a declaration with the same name as a previously declared identifier (in the same scope), except that both declarations have different arguments and different definition (implementation)

Overload resolution rules:

- An exact match
- A promotion (e.g. char to int)
- A standard type conversion (e.g. between float and int)
- A constructor or user-defined type conversion

```cpp
void f(int a);
void f(float value);


void g(int a);


void h(int a);
void h(int a, int b = 0);
```

```cpp
  f(0);    // ok
// f('a'); // ambiguous matches, compile error
  f(2.3f); // ok
// f(2.3); // ambiguous matches, compile error


  g(2.3);  // ok, standard type conversion


// h(3);   // ambiguous matches, compile error
```

Functor (Function as Argument)

Definition (Functor)

Functors, or **function object**, are objects that can be treated as parameters

Function type:
$<$return_type$>$(*[function_name])($<$arg_type1$>$, $<$arg_type2$>$, ...)

```cpp
int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}
int add(int a, int b) { //type: int (*)(int, int)
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}

cout << eval(4, 3, add); // print 7
cout << eval(4, 3, sub); // print 1
```

Union and Bitfield

Union

Definition (Union)

A **unions** is a special data type that allows to store different data types in the same memory location

- The union is only as big as necessary to hold its largest data member
- The union is a kind of *"overlapping"* storage

```cpp
union A {
    int  x;
    char y;
}; // sizeof(A): 4

A a;
a.x = 1023;      // bits: 00..01111111111
a.y = 0;         // bits: 00..01100000000
std::cout << a.x; // print 512 + 256 = 768
```

Bitfield

Definition (Bitfield)

A `bitfield` is variable of a structure with a predefined bit width.
A bitfield can hold bits instead byte

```
struct S1 {
    int b1 : 10;   // range [0, 1023]
    int b2 : 10;   // range [0, 1023]
    int b3 : 8;    // range [0, 255]
};

struct S2 {
    int b1 : 10;
    int    : 0;    // force start at bit 32
    int b2 : 10;
};
```

Preprocessing

Preprocessor macros are evil:

Do not use macro expansion!!

...or use as little as possible

- Macros can't be debugged
- Macro expansions can have strange side effects
- Macros have no namespace or scope

Preprocessor Macros:

- **#if** <condition>, **#elif** <condition>, **#else**, **#endif**
- **#if defined(...)** equal to **#ifdef** ...
- **#if !defined(...)** equal to **#ifndef** ...
- **#define** <macro> multi lines \
- **#undef** <macro> (every macro should be undefined for safety reasons)

Commonly useful macros:

- __LINE__ Integer value representing the current line in the source code file being compiled

- __FILE__ A string literal containing the presumed name of the source file being compiled

- __DATE__ A string literal in the form "MMM DD YYYY" containing the date in which the compilation process began

- __TIME__ A string literal in the form "hh:mm:ss" containing the time at which the compilation process began

- **Select code depending on the C/C++ version**
  - `#if defined(__cplusplus)` C++ code
  - `#if __cplusplus == 199711L` ISO C++ 1998/2003
  - `#if __cplusplus == 201103L` ISO C++ 2011

- **Select code depending on the compiler**
  - `#if defined(__GNUG__)` The compiler is gcc/g++
  - `#if defined(__clang__)` The compiler is clang/clang++

- **Select code depending on the operation system or environment**
  - `#if defined(_WIN64)` OS is Windows 64-bit
  - `#if defined(__linux__)` OS is Linux
  - `#if defined(__APPLE__)` OS is Mac OS
  - `#if defined(__MINGW32__)` OS is MinGW 32-bit
  - ...and many others

49/57

Do not define macro in header file and before includes!!

Example:

```cpp
#include <iostream>

#define value     // very dangerous!!
#include <big_lib>
using namespace std;

int main() {
    cout << f(4); // should print 7, but it prints always 3
}
```

big_lib.hpp:

```cpp
int f(int value) {  // 'value' disapear
    return value + 3;
}
```

Use parenthesis in macro definition!!

Example:

```cpp
#include <iostream>
using namespace std;

#define SUB1(a, b)  a - b      // wrong
#define SUB2(a, b) (a - b)     // wrong
#define SUB3(a, b) ((a) - (b)) // correct

int main() {
    cout << (5 * SUB1(2, 1)); // print 9 not 5!!
    cout << SUB2(3+3, 2+2);   // print 6 not 2!!
    cout << SUB3(3+3, 2+2);   // print 2
}
```

Macro make hard to find compile errors!!

Example:

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: #define F(a) {              \
 5:         ...                \
 6:         ...                \
 7:         return v;
 8:
 9: int main() {
10:     F(3);      // compile error at line 5!!
11: }
```

- **In which line is the error??!**

In C++11, a **variadic macro** is a special macro accepting a varying number of arguments (separated by comma)

Each occurrence of the special identifier `__VA_ARGS__` in the macro replacement list is replaced by the passed arguments

Example:

```
#define PRINT3(...) \
printf("list: %d %d %d\n", __VA_ARGS__);

#define PRINT4(...) \
printf("list: %d %d %d %d\n", __VA_ARGS__);

PRINT3(1, 2, 3)
PRINT4(1, 2, 3, 4)
```

When macros are necessary:

- **Conditional compiling**: different architectures, compiler features, etc.
- **Mixing different languages**: code generation (example: asm assembly)
- **Complex name replacing**: see template programming

Otherwise, prefer const and constexpr, specially for constant values and functions

```cpp
#define SIZE 3 // replaced with
const int SIZE = 3;

#define SUB(a, b) ((a) - (b)) // replaced with
constexpr int sub(int a, int b) {
    return a - b;
}
```

Most important `pragmas`:

- `#pragma once` In C++11, it indicates that a (header) file is only to be parsed once, even if it is (directly or indirectly) included multiple times in the same source file.
  It is an alternative of the standard include guard:

  ```
  #ifndef FILENAME_H
  #define FILENAME_H
  ...code...
  #endif /* FILENAME_H */
  ```

- `#pragma unroll` Applied immediately before a for loop, it replicates his body to eliminates branches. Unrolling enables aggressive instruction scheduling (supported by Intel/Ibm/Clang compilers).

- `#pragma message "text"` Display informational messages at compile time. (every time this instruction is compiled).

- `_Pragma`(<command>) (C++11)
  It is an operator (like `sizeof`), and can be embedded in a macro (ex. `#define`)

```
#define MY_LOOP      \
_Pragma(unroll)      \
for(i=0;i<10;i++)    \
    cout<<"c";
```

- `#error "text"` The directive emits a user-specified error message at compile time when the compiler parse the related instruction.

Find the size offset of a field inside a structure:

```cpp
#define FIELD_OFFSET(structure, field)                       \
    reinterpret_cast<size_t>(
        &((reinterpret_cast<structure*>(0))->field) )
```

Get the size of an arbitrary type without using `sizeof`

```cpp
#define MY_SIZE(type, ret)                                   \
    { type x; ret = reinterpret_cast<char*>(&x + 1) -        \
                    reinterpret_cast<char*>(&x); }
```

```cpp
struct A {
    int   a;
    float b;
};

std::cout << FIELD_OFFSET(A, b); // print 4
int size;
MY_SIZE(A, size); // size = 8
```