

# POINTERS, REFERENCES, UNDEFINED BEHAVIOUR



# PROGRAMMA

1. Undefined Behaviour; wat?
2. Undefined Behaviour; het echte gevaar
3. Unspecified Behaviour; huh?
4. Pointers
5. References
6. Smart Pointers
7. Verder lezen/kijken/luisteren

# UNDEFINED BEHAVIOUR; WAT?

Wat is de uitkomst van het volgende stukje code?

```
1 int func(int k){  
2     return k+10;  
3 }  
4  
5 int x = INT_MAX;  
6 std::cout << func(x) << std::endl;
```

## ZIJSTAPJE; C-STYLE ARRAY VOOR STRINGS

```
1 char[] t = "Hallo";  
2  
3 char* t = "Hallo";  
4  
5 void printSomething(char* p){printf("p: %s",p);}
```

# UNDEFINED BEHAVIOUR; WAT?

Wat is de uitkomst van het volgende stukje code?

```
1 char t[] = "Hallo";  
2 t[6] = '!';
```

# UNDEFINED BEHAVIOUR; WAT?

- Elke taal heeft **beschreven** gedrag
- Soms kan code wel compileren, maar is **gedrag niet beschreven**; Undefined
- Gedrag van code is **niet te voorspellen**; Undefined Behaviour
- (Uitkomst van) code onbruikbaar?

# UNDEFINED BEHAVIOUR; HET ECHE GEVAAR

- UB *ergens* in je programma maakt het hele programma undefined
- Volgens de standaard: UB komt niet voor, dus mogen het negeren

Bijvoorbeeld: Signed Integer Overflow

```
1          //| x86 ASM
2 bool f(int i) {          //| f:
3     return i+1 > i;      //| mov $0x1, %eax
4 }                      //| retq
5                      //|
6 bool g(int i) {          //| g:
7     if (i == INT_MAX) return false; //| mov $0x1, %eax
8     else return f(i);    //| retq
9 }                      //|
```

# UNDEFINED BEHAVIOUR; HET ECHE GEVAAR

- UB *ergens* in je programma maakt het hele programma undefined
- Volgens de standaard: UB komt niet voor, dus mogen het negeren

Bijvoorbeeld: Signed Integer Overflow

```

1 void read_from_network(int size) {
2     // size > size+1 is impossible since signed
3     // overflow is impossible. Optimize it out!
4     // if (size > size+1) errx(1, "packet too big");
5
6     char *buf = malloc(size+1);
7     if (buf == NULL)
8         errx(1, "out of memory");
9
10    read(fd, buf, size);
11    // ... error checking on read.
12
13    buf[size] = 0;
14    process_packet(buf);
15    free(buf);
16 }
```



# UNDEFINED BEHAVIOUR; HET ECHTE GEVAAR

- Probleem: Security Audits zien dit snel over het hoofd
- Hoe check je dan wel op signed integer overflow?
- Vóóordat je iets aanpast

```
1  int a = <something>;  
2  int x = <something>;  
3  if (x > 0 && a > INT_MAX - x) { ... } // `a + x` would overflow
```

In het algemeen:

- Ja, je code werkt nu misschien
- Maar morgen nog? Op andere architectuur? Andere compiler?

# UNDEFINED BEHAVIOUR; WAT?

Voorbeelden:

- Signed integer overflow
- Out of bounds memory access
- Zero division
- Gebruik van ongeïnitieerde variabele
- Dereferencing null ptr
- Gebruik van objecten na lifetime
- En meer... Helaas...
- Oplossingen zijn specifiek per geval

# UNSPECIFIED BEHAVIOUR; HUH?

En ook; Implementation-defined behaviour. Verschil?

Unspecified behaviour:

- **Hóeft niet** gedocumenteerd te worden
- Order of Evaluation

```
1  #include <iostream>
2  void f(int a, int b, int c){
3      printf("%d, %d, %d", a,b,c);
4  }
5
6  int main(){
7      int x = 12;
8      f(x, x++, x+20);
9  }
```

- String Literal Comparison

```
1  char[] t = "Hallo";
2
3  std::cout << t == "Hallo" << std::endl;
```

Implementation defined behaviour:

- **Moet wel** gedocumenteerd worden
- Waarde van `std::sizeof(int)`;
  - 32- of 64-bit
- Uit hoeveel bits een byte bestaat
  - Mééstal 8, maar niet altijd!

# PROOF IS IN THE PUDDING;

```
1 #include <iostream>
2 void f(int a, int b, int c){
3     printf("%d, %d, %d", a,b,c);
4 }
5
6 int main(){
7     int x = 12;
8     f(x, x++, x+20);
9 }
```



12, 12, 33

**<BR>**

# POINTERS

Variabele voor het wijzen naar de plek van een object in het geheugen

Bijzonder: `nullptr` ; pointer die (nog) nergens naar wijst

Pointers kunnen wijzen naar verschillende soorten geheugen:

- **Stack-geheugen**: Voor lokale variabelen.
- **Heap-geheugen**: Voor dynamisch gealloceerde objecten.
  - `malloc` & `free` in C
  - `new` & `delete` in C++

Wanneer dynamisch geheugen?

- Niet altijd duidelijk **hoeveel geheugen** je nodig hebt @ compile time
- Hele grote objecten wil je **niet in de stack** plaatsen
- Data structuren zonder **upper bound**

# POINTERS

Pointers kunnen worden gebruikt voor:

- **Pointer arithmetic:** Navigeren door arrays of geheugenblokken.

```
1  int a[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
2  int len = sizeof(a)/sizeof(int);
3  int *x = a;
4  int i = 0;
5
6  for(i = 0; i < len; i++){
7      printf("Address of subscript %d = %d Value = %d\n", i, x, *x);
8      x++;
9  }
```

- **Function pointers:** Functies als argumenten doorgeven.

```
1  void loop(unsigned int count, void(*func)()) {
2      for (unsigned int i = 0; i < count; i++)
3          func();
4  }
```

# POINTERS

Gevaren van pointers:

- UB met **pointer arithmetic**: Non-array types, out of bounds memory access.
- **Memory leaks**: Geheugen dat niet wordt vrijgemaakt.
- **Null pointers**: Dereferencing kan leiden tot crashes.
- **Dangling pointers**: Wijzen naar vrijgemaakt geheugen.

```
1  Class *object = new Class();
2  Class *object2 = object;
3
4  delete object;
5  object = nullptr;
6  // Waar wijst object2 heen?
```

```
1  Object *method() {
2      Object object;
3      return &object;
4  }
5
6  Object *object2 = method();
7  // object bestaat niet meer, dus waar wijst object2 naar?
```



# REFERENCES

- Een **reference** is een alias voor een bestaande variabele.
- Wordt gebruikt om een variabele door te geven zonder een kopie te maken.
- **Altijd gebonden aan een bestaand object**; kan niet `null` zijn zoals een pointer.

Voordelen van references:

- Makkelijker te gebruiken dan pointers.
- Geen risico op `null` of dangling references (tenzij via UB).
- Ideaal voor **pass-by-reference** in functies.

Gebruik van references:

- **Functieargumenten**: Vermijd kopiëren van grote objecten.
- **Returnwaarden**: Geef een alias terug naar een bestaand object.
- **Const references**: Voor veilige toegang zonder wijzigingen.

# SMART POINTERS

- References zijn dus **niet altijd** een vervanging voor pointers
  - Voor dynamic memory management bieden ze geen oplossing
- Pointers zijn **té veelzijdig**:
  - Single object <> Arrays
  - Ownership <> non-ownership
- In plaats hiervan kun je **Smart Pointers** gebruiken:
  - ... Maar is te veel voor nu, en een les op zich
  - Sorry :(

# VERDER LEZEN/KIJKEN/LUISTEREN

Undefined behaviour:

Undefined Behavior in C++: What Every Programmer Should Know and Fear; Fedor Pikus

Back To Basics: Undefined Behavior; Ansel Sermersheim & Barbara Geller

Undefined Behavior is Not an Error; Barbara Geller & Ansel Sermersheim

Pointers & References:

What is the Difference Between a Pointer and a Reference C++

Back to Basics: C++ Smart Pointers; David Olsen

– E inde –