Pointers

Programação (L.EIC009)

José Proença (FCUP) & João Bispo (FEUP) – *slides by Eduardo R. B.*

Outline

- Basic aspects
- Pointers and arrays
- Pointers and `struct` types
- `nullptr`: the null pointer.
- Pointer arithmetic
- Example functions - using pointers for array traversal

Basic aspects

Definition

A pointer variable `ptr` of type `T*` is declared as

```
T* ptr;
```

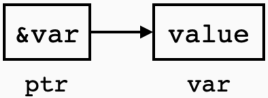
The domain of values for `T*` are memory addresses of values of type `T`.

The & and * operators

If `var` has type `T`, then `&var` has type `T*` and may be used to initialise `ptr` – `&` is called the **address operator**.

In turn, `*ptr` is a reference to the memory address pointed to by `ptr` – `*` is called the **dereferencing operator**.

```
T var;
T* ptr = &var;
*ptr = value; // <=> var = value;
```



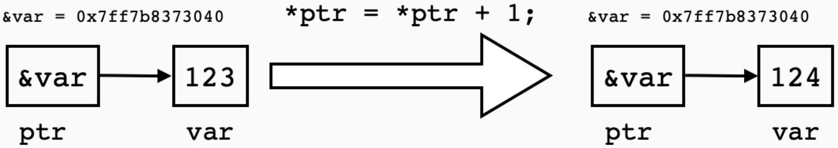
The & and * operators (cont.)

```cpp
int var = 123;
int *ptr = &var;
*ptr = *ptr + 1;
std::cout << ptr << ' '
     << *ptr << ' '
     << var;
```

```
0x7ff7b8373040 124 124
```

ptr holds a memory address (an integer), the address of variable var (&var) - typically this is is printed in hexadecimal format.

*ptr refers to var, hence *ptr = *ptr + 1; has the same effect as var = var + 1;. In this sense pointers behave the same as reference variables we discussed earlier.



Pointers can be used for call-by-reference semantics. Reference variables are more convenient, as they have less syntactic overhead (no need to use & or *).

```
// Using references
void get_min_max(int a, int b,
                 int& min, int& max) {
  min = a<b ? a : b; max = a>b ? a : b;
}
// Using pointers
void get_min_max(int a, int b,
                 int* min, int* max) {
  *min = a<b ? a : b; *max = a>b ? a : b;
}
```

```
int main() {
  int x = 200, y = 100, m, M;
  // call to ref. version
  get_min_max(x, y, m, M);
  // call to pointer version
  get_min_max(x, y, &m, &M);
  ...
}
```

Pointers vs. references (cont.)

Pointers came first (with C), references later (with C++).

References can be thought of as a special kind of pointer, "initialise-once" pointers. References tend to be more friendly to use, but their application is more limited.

Pointers are more expressive than references:

- Pointers can be **re-assigned**.
- **Arithmetic and relational operators** can be used with pointers, in particular when pointers refer to array positions.
- Pointers can be defined with **more than one level** of indirection, e.g., we can have arrays of pointers and pointers-to-pointers.
- And, as we will see in future classes, pointers are required to **deal with dynamically allocated memory**.

Pointers and arrays

```
void fill_with_zeros(int a[], int n) { ... }
```

is really **the same** as having

```
void fill_with_zeros(int* a, int n) { ... }
```

The compiler will complain if you define both:

```
error: redefinition of 'fill_with_zeros'
```

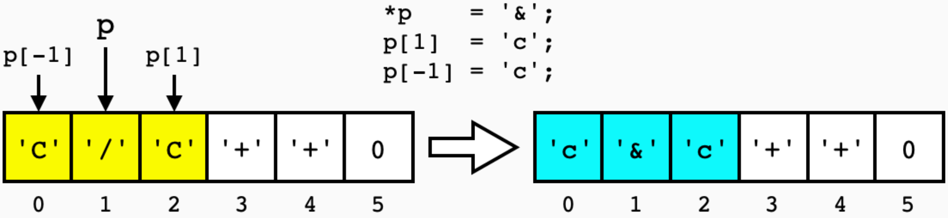
Thus, an array is passed by reference in the sense that a function gets the array's memory address as argument.

The first variant above is more clear in the sense that it explicitly suggests an array argument (`int a[]`), whereas the second variant could also be interpreted as receiving a reference to a plain `int` variable (as in the `get_min_max` example).

The [] operator

We can initialise a pointer to refer to an array position. As with arrays, the [] operator can be used with pointers for index-based access.

```
char s[6] = "C/C++";
std::cout << s << '\n';
char* p = &s[1];
*p    = '&'; // <=> s[1] = '&';
p[1]  = 'c'; // <=> s[2] = 'c';
p[-1] = 'c'; // <=> s[0] = 'c';
std::cout << s << '\n'; // --> "c&c++"
```

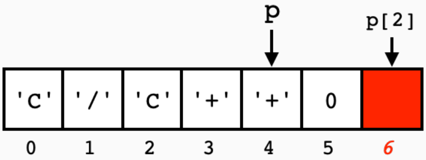


Buffer overflows!

Note that buffer overflows are possible using pointers!

```
char s[6] = "C/C++";
char* p = &s[4];
p[2] = 'X'; // <=> s[6] = 'X'; <-- BUFFER OVERFLOW!
```

```
==80384==ERROR: AddressSanitizer: stack-buffer-overflow
. . .
SUMMARY: AddressSanitizer: stack-buffer-overflow
pointer_buffer_overflow.cpp:4 in main
. . .
```



Arrays of pointers

U.PORTO

We can define arrays of pointers.

```
int a = 1, b = 2, c = 3;
int* iparr[] = { &a, &b, &c };
for (int i = 0; i < 3; i++) { *iparr[i] = 0; }
std::cout << a << ' ' << b << ' ' << c << '\n';

const char* sparr[] = { "Hello", "_", "C++" };
for (const char* s : sparr) std::cout << s;
std::cout << '\n';
```

```
0 0 0
Hello_C++
```

A note on string constants

U.PORTO

In the previous example:

```
const char* sparr[] = { "Hello", "world!", "2022" };
```

is an array of pointers, **not** a bi-dimensional array of type `char`.

String constants are placed in global read-only memory. Note that we can write

```
// const must be used
const char* s = "Hello world!";
```

which is different from using the string constant to initialise a `char` array.

```
char s[] = "Hello world!";
```

Arrays of pointers - another example

The `main` function can be declared as:

```
int main(int argc, char* argv[])
```

where `argc` is the number of arguments passed through the command line, and `argv` is an array of pointers to the arguments (C-strings).

```
// main_with_args.cpp
#include <iostream>

int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++)
    std::cout << "Arg. " << i << ": \"" << argv[i] << "\"\n";
  return 0;
}
```

Arrays of pointers - another example (cont.)

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++)
    std::cout << "Arg. " << i << ": \""
              << argv[i] << "\"\n";
. . .
```

```
  $ ./main_with_args C++ @ UP "March 21, 2022"
  Arg. 0: "./main_with_args"
  Arg. 1: "C++"
  Arg. 2: "@"
  Arg. 3: "UP"
  Arg. 4: "March 21, 2022"
```

Pointers and `struct` types

Definition

U.PORTO

Pointers work with struct types as well.

```
struct time_of_day {
  unsigned char h;
  unsigned char m;
};
time_of_day t { 12, 57 };
time_of_day* p = &t;
(*p).h = 13;
(*p).m = 58;      // t now contains { 13, 58 }
*p = { 14, 59 }; // t now contains { 14, 59 }
```

The -> operator

For `struct` pointers, the **pointer member operator** `->` can be used to access member fields: `p->member_field` is typically more readable than `(*p).member_field`.

```
time_of_day t { 12, 57 };
time_of_day* p = &t;
p -> h = 13; // <=> (*p).h = 13;
p -> m = 58; // <=> (*p).m = 58;
```

Pointers for member fields

U.PORTO

Pointers can also be defined over member fields of `struct` types:

```
time_of_day t { 12, 57 };
time_of_day* p = &t;
unsigned char* h = &(t.h);
unsigned char* m = &(p -> m);
*h = 13; // <=> t.h = 13;
*m = 58; // <=> t.m = 58;
```

`nullptr`: the null pointer

nullptr == 0

`nullptr` is a C++ keyword (introduced in C++ 11) that stands for the **null pointer**.

`nullptr` is used as a "points-to-nothing" value for pointers. Its value is address constant **0**!

```
int* p = nullptr;
std::cout << "p = " << p << '\n';
```

```
p = 0
```

It is also common to use `NULL` to denote the null pointer (since the early days of C).

As a “points-to-nothing” value for pointers, `nullptr` many times marks the end of a sequence. For example, `main` can also have a `envp` argument similar for environment variables that is `nullptr`-terminated.

```
#include <iostream>

int main(int argc, char *argv[], char *envp[]) {
  int i = 0;
  while (envp[i] != nullptr) {
    std::cout << envp[i] << '\n';
    i++;
  }
  return 0;
}
```

Example (cont.)

U. PORTO

```
int main(int argc,
         char *argv[],
         char *envp[]) {
  int i = 0;
  while (envp[i] != nullptr) {
    std::cout << envp[i] << '\n';
    i++;
  }
  . . .
}
```

(environment variables are printed with the format `VAR=VALUE`)

```
$ ./main_with_env_args
...
HOME=/Users/edrdo
SHELL=/bin/bash
...
```


`nullptr`: the null pointer

Access to memory using `nullptr` ?

`nullptr` must not be used to read/write memory! The semantics of C++ are undefined when this happens.

```
int* p = nullptr;
*p = 123;
```

UBSan and ASan signal this type of error.

```
null_pointer_access.cpp:4:6: runtime error:store to null pointer
of type 'int'
. . .
ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
```

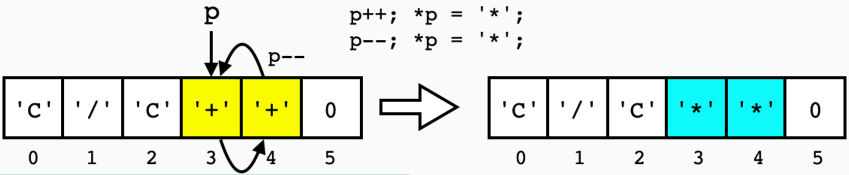
Programs typically crash in most cases like this … but compilers sometimes generate “surprising” code (as in other cases of undefined behavior).

Pointer arithmetic

Pointer arithmetic

Increment and decrement operators change the pointer by one position in memory:

```
char s[6] = "C/C++"; std::cout << s << '\n';
char* p = &s[3];
p++;        // p now points to s[4]
*p = '*'; // <=> s[4] = '*';
p--;        // p points to s[3] again
*p = '*'; // <=> s[3] = '*';
std::cout << s << '\n'; // --> "C/C**"
```

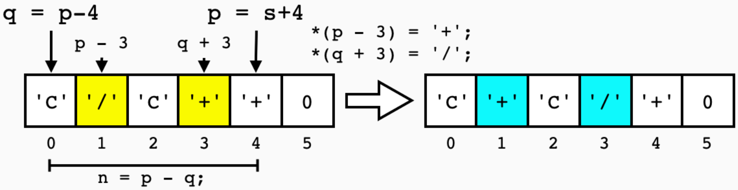


Pointer arithmetic (cont.)

More generally, + and − can be be used to define pointer values. Subtracting two pointers can tell us the offset/distance between them:

```
char s[6] = "C/C++"; std::cout<<s<<'\n';
char* p = s+4; // <=> p = &s[4];
char* q = p-4; // <=> q = &s[0];
*(p-3) = '+';  // <=> s[1] = '+';
*(q+3) = '/';  // <=> s[3] = '/';
```

```
std::cout<<s<<'\n';
  // '--> "C+C/+"
int n = p-q;
  // <=> n = &s[4] - &s[0];
std::cout<<n<<'\n'; // --> 4
```



Relational operators

Pointers can also be compared using relational operators ==, !=, <, <=, >, and >=.

```
char a[5]; char* p = a + 1; char* q = a + 2;
std::cout << boolalpha
 <<  "p == q ? " << (p == q) << " / p != q ? " << (p != q)
 << "\np < q ? " << (p < q)  << " / q < p ? "  << (q < p)
 << "\np > q ? " << (p > q)  << " / q > p ? "  << (q > p)
 << '\n';
```

```
p == q ? false / p != q ? true
p < q ? true / q < p ? false
p > q ? false / q > p ? true
```

Interpretation for `p < q`: `true` when memory address stored in `p` precedes (is lower than) the memory addresss stored in `q` (<=, >, and >= work similarly).

Example functions - using pointers for array traversal

The `length` function over C-strings we presented in previous classes is as follows:

```
int length(const char str[]) {
  int l = 0;
  while (str[l] != '\0') l++;
  return l;
}
```

We could express it alternatively using pointer arithmetic as:

```
int length(const char str[]) {
  const char* p = str;
  while (*p != '\0') p++;
  return p - str; // "distance" between str and p!
}
```

```
void copy(char dst[], const char src[]) {
  int i = 0;
  while (src[i] != '\0') { dst[i] = src[i]; i++; }
  dst[i] = '\0';
}
```

can be expressed alternatively as

```
void copy(char dst[], const char src[]) {
  char* p = dst;
  const char* q = src;
  while (*q != '\0') { *p = *q; p++; q++; }
  *p = '\0';
}
```

```
void reverse(int a[], int n) {
  int i = 0, j = n - 1;
  while (i < j) { // i != j would be incorrect (for even n)
    int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
    i++; j--;
}}
```

can be expressed alternatively as

```
void reverse(int a[], int n) {
  int* p = a;  int* q = a + n - 1;
  while (p < q) { // p != q would be incorrect (for even n)
    int tmp = *p; *p = *q; *q = tmp;
    p++; q--;
}}
```