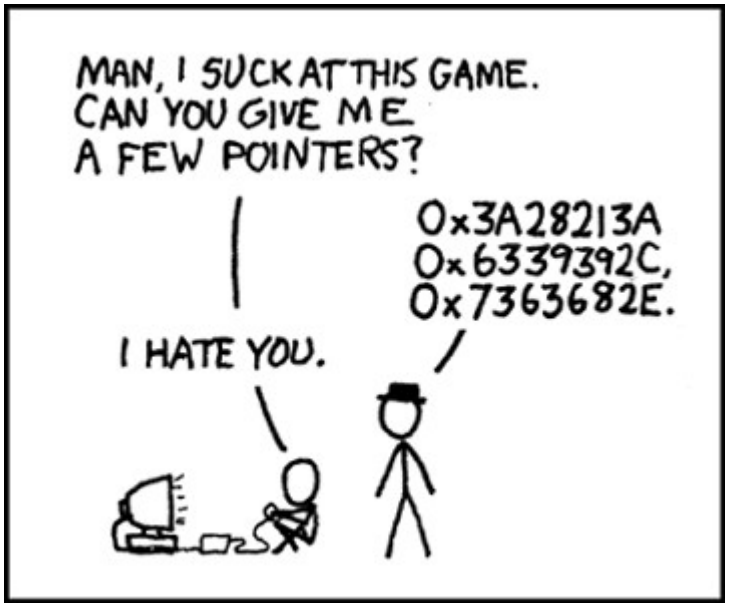
C++ Pointers (Part II)



Krishna Kumar

Every computer, at the unreachable memory address 0x-1, stores a secret. I found it, and it is that all humans ar-- SEGMENTATION FAULT
XKCD (https://xkcd.com/138/)

Pointers (recap)

- Pointer values are memory addresses
- A pointer **does not** know the **number of elements** that it's pointing to.
- A pointer ***does*** know the **type of the object** that it's pointing to
- **Stack:** Stores local data, return addresses, used for parameter passing
    - Local variables
    - Cleared when out of scope
- **Heap:** You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data
    - Accessed using *new* and *delete*
    - *Memory leaks – manual delete of used memory*

Void*

- **void*** means “pointer to some memory that the compiler doesn't know the type of”
- We use **void*** when we want to transmit an address between pieces of code that really don't know each other's types – so the programmer has to know
- The primary use for void* is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions
- There are no objects of type **void**
  - **void v;** *// error*
  - **void f();** *// f() returns nothing – f() does **not** return an object of type **void***
- Any pointer to object can be assigned to a **void***
  - **int* pi = new int;**
  - **double* pd = new double[10];**
  - **void* pv1 = pi;**
  - **void* pv2 = pd;**

void* (cont...)

- To use a **void*** we must tell the compiler what it points to

```
void f(void* pv) {
    void* pv2 = pv;          // copying is ok (copying is what void*s are for)
    double* pd = pv;         // error: can't implicitly convert void* to double*
    *pv = 7;                 // error: you can't dereference a void*
    ++pv;                    // error: can't increment void*
                             // (the size of the object pointed to is unknown)
    pv[2] = 9;               // error: you can't subscript a void*
    int* pi = static_cast<int*>(pv);    // ok: explicit conversion
                                        // Unsafe!!
    // …
}
```

- A **static_cast** can be used to explicitly convert to a pointer to object type
  - "**static_cast**" is a deliberately ugly name for an ugly (and dangerous) operation – use it only when absolutely necessary

Warnings

- There are some serious *gotchas* when using casts on pointers, so try to avoid them
  - They are ***far*** too complicated to cover in this course
  - There is precisely ***one*** safe use, to get back to the type you started with – and a compiler can't check you got it right

```
myclass object();

myclass* myptr = object.addr();

void* rawptr = myptr;

. . .

myclass* newptr = static_cast<myclass*>(rawptr);
```

Never write it like C

- You will often see C-style casts in C++ code derived from C – or by C programmers writing “C++”
  - They look like **`(newtype)expression`**
- But ***don't*** write them
  - They are extremely hard to spot in non-trivial code
  - They are even ***less*** safe than **`reinterpret_cast`**

void*

- **void*** is the closest C++ has to a plain machine address
- Functions using void* pointers typically exist at the very lowest level of the system, where real hardware resources are manipulated.

  void* my_alloc(size_t n); // allocate n bytes
  //from my special heap
- **Avoid using it if you possibly can**
  - It bypasses essentially all type checks
  - Occurrences of void*s at higher levels of the system should be viewed with great suspicion because they are likely indicators of design errors.
- Pointers to functions and pointers to members cannot be assigned to void*s.