```
constexpr bool cf() { return true; }  //    constexpr function returning true
          bool nf() { return true; }  // nonconstexpr function returning true

typedef bool (*Fp)(); // pointer to function taking no args. and returning bool

constexpr Fp cpcf = cf;  //    constexpr pointer to a    constexpr function
          Fp npcf = cf;  // nonconstexpr pointer to a    constexpr function
constexpr Fp cpnf = nf;  //    constexpr pointer to a nonconstexpr function
          Fp npnf = nf;  // nonconstexpr pointer to a nonconstexpr function
constexpr Fp cpz  = 0;   //    constexpr pointer having null pointer value

static_assert(cpcf == &cf, "");  // OK, reading a constexpr pointer
static_assert(npcf == &cf, "");  // Error, npcf is not a constexpr pointer.
static_assert(cpz  == 0,   "");  // OK, reading a constexpr pointer

static_assert(cpcf(),      "");  // OK, invoking a constexpr function through a
                                 //     constexpr pointer
static_assert(npcf(), "");       // Error, npcf is not a constexpr pointer.
static_assert(cpnf(), "");       // Error, can't invoke nonconstexpr function
static_assert(npnf(), "");       // Error, npnf is not a constexpr pointer.
static_assert(cpz(),  "");       // Error, 0 doesn't designate a function.
```

constexpr member functions

Member functions — including certain **special member functions**, such as *constructors* but not *destructors* — can be declared to be **constexpr**; see *Literal types defined* on page 278:

```
class Point1
{
    int d_x, d_y;  // two ordinary int data members
public:
    constexpr Point1(int x, int y) : d_x(x), d_y(y) { }  // OK, is constexpr

    constexpr int x()       { return d_x; }  // OK, is constexpr
              int y() const { return d_y; }  // OK, is not constexpr
};
```

Simple classes, such as Point1 above, having at least one **constexpr** constructor that is neither a *copy* nor a *move* constructor and ~~satisfies~~ all other requirements of being a literal type — see *Literal types defined* on page 278 — can be evaluated as part of constant expressions. However, it is only when combined with a means to access a data member at compile time (e.g., via a public data member, a **constexpr** accessor, or a free **constexpr friend** function) that an object of even literal type can contribute to a constant expression’s *value*:

```
int ax[Point1(5, 6).x()];  // OK, array of 5 ints
int ay[Point1(5, 6).y()];  // Error, accessor y is not declared constexpr.
```