

```

template <typename Type, int T> // We want to have a fix-sized stack as follows
class FixedStack { // A fixed stack of size N
public:
    typedef    int    sz_t;        // The type for counting elements
    FixedStack(): top_(0) {}
    sz_t  count() const { return top_;}
    bool  full()  const { return top_ == N;}
    bool  empty() const { return top_ == 0;}
    FixedStack& push(const Type& t)
{ if(full()) over_flow(); s[top_++] = t; return *this;}
    FixedStack& pop()
{ if (empty()) under_flow(); top_--; return *this;}
    Type&  top() { if (empty()) under_flow(); return s[count() - 1];}
    const Type& top() const { return const_cast<Stack&>(*this).top();}
private:
    static void under_flow();
    static void over_flow();
    Type s[N];
    sz_t top_;
};

```

```

// The above will blow-up the executable since every type
// and every size will create a different code.
class StackBase {          // The solution: Minimizes code bloat for templates
public:
    typedef    int    sz_t; // The type for size
    explicit StackBase(sz_t n): size_(set(n)), top_(0) {}
    sz_t  size()  const { return size_;}
    sz_t  count() const { return top_;}
    bool  full()  const { return count() == size();} // or top_ == size_
    bool  empty() const { return count() == 0;}
protected:
    static void under_flow();
    static void over_flow();
private:
    static sz_t set(sz_t n); // Checks if n is reasonable, etc.
    sz_t  size_,
    top_;
};

```

```

template <typename Type>
class Stack: public StackBase { // The "simple" class template
public:
    explicit Stack(sz_t n): StackBase(n), s(new Type[size()]) {}
    Stack(const Stack& src): StackBase(src.size()),
        a(src.copy_to(new Type[size()]))) {}
    ~Stack() { delete[] s;}
    Stack& operator=(const Stack& src);
    Stack& push(const Type& t);
    Stack& pop();
    Type& top() { if (empty()) under_flow(); return s[count() - 1];}
    const Type& top() const { return const_cast<Stack&>(*this).top();}
private:
    Type* copy_to(Type*) const; // Copies 's' to a target array
    Type *s;
};

```

```
                // A compile-time safe fixed-Stack
template <typename T, unsigned N>
class FixedStack: public Stack<T> {
public:
    FixedStack(): Stack<T>(N) {}
    // No need for Copy-Ctor and Dtor (and operator=)
    // However, operator= is problematic if not polymorphic - but used as one
};
```